

0 Preface

0.0 Introduction

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

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

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

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

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

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

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

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

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

—End of Introduction

0.1 Current Edition

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

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

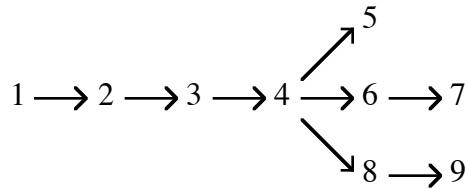
Lecture slides and solutions to exercises are available to course instructors from the author.

—End of Current Edition

0.2 Quick Tour

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

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



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

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

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

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

End of Quick Tour

0.3 Acknowledgements

For inspiration and guidance I thank Working Group 2.3 (Programming Methodology) of the International Federation for Information Processing, particularly Edsger Dijkstra, David Gries, Tony Hoare, Jim Horning, Cliff Jones, Bill McKeeman, 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 Benet Devereux, Lorene Gupta, Peter Kanareitsev, Yannis Kassios, Victor Kwan, Albert Lai, Chris Lengauer, Andrew Malton, Theo Norvell, Rich Paige, Dimi Paun, Hugh Redelmeier, Alan Rosenthal, Anya Taffliovich, Justin Ward, and Robert Will. For their critical and helpful reading of the first draft I am most grateful to Wim Hesselink, Jim Horning, and Jan van de Snepscheut. For good ideas I thank Ralph Back, Wim Feijen, Netty van Gasteren, Nicolas Halbwachs, Gilles Kahn, Leslie Lamport, Alain Martin, Joe Morris, Martin Rem, Pierre-Yves Schobbens, Mary Shaw, Bob Tennent, and Jan Tijmen Udding. For reading the draft and suggesting improvements I thank Jules Desharnais, Andy Gravell, Ali Mili, Bernhard Möller, Helmut Partsch, Jørgen Steensgaard-Madsen, and Norbert Völker. I thank my classes for finding errors.

End of Acknowledgements

End of Preface

1 Basic Theories

1.0 Boolean Theory

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

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

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

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

	\top	\perp
\neg	\perp	\top

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

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

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

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

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

The following truth table shows how the classification of boolean expressions formed with two-operand operators can be obtained from the classification of the operands. Above the horizontal line, the pair $\top\top$ means that both operands are theorems; the pair $\top\perp$ 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 \perp means that the result is an antitheorem.

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

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

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

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

	$\top\top\top$	$\top\top\perp$	$\top\perp\top$	$\top\perp\perp$	$\perp\top\top$	$\perp\top\perp$	$\perp\perp\top$	$\perp\perp\perp$
if then else	\top	\top	\perp	\perp	\top	\perp	\top	\perp

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

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

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

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

- We sometimes have to insert parentheses around expressions that are replacing variables in order to maintain the precedence of operators. In the example of the preceding paragraph, we replaced a conjunct x with an implication $\perp \Rightarrow \neg(\perp \vee \top)$; since conjunction comes before implication in the precedence table, we had to enclose the implication in parentheses. We also replaced a conjunct y with a disjunction $\perp \vee \top$, so we had to enclose the disjunction in parentheses.
- When the same variable occurs more than once in an expression, it must be replaced by the same expression at each occurrence. From $x \wedge x$ we can obtain $\top \wedge \top$, but not $\top \wedge \perp$. However, different variables may be replaced by the same or different expressions. From $x\wedge y$ we can obtain both $\top \wedge \top$ and $\top \wedge \perp$.

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

1.0.0 Axioms and Proof Rules

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

Axiom Rule If a boolean expression is an axiom, then it is a theorem. If a boolean expression is an anti-axiom, then it is an antitheorem.

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

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

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

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

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

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

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

The boolean expression

$$\top \vee x$$

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

$$x \vee \neg x$$

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

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

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

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

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

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

1.0.1 Expression and Proof Format

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

$$a \wedge b \vee c$$

$$a \wedge b \vee c$$

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

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

$$(\textit{first part}$$

$$\wedge \textit{second part})$$

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

$$\textit{first part}$$

$$= \textit{second part}$$

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

A proof is a boolean expression that is clearly a theorem. One form of proof is a continuing equation with hints.

$$\textit{expression0} \qquad \text{hint 0}$$

$$= \textit{expression1} \qquad \text{hint 1}$$

$$= \textit{expression2} \qquad \text{hint 2}$$

$$= \textit{expression3}$$

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

$$\textit{expression0} = \textit{expression1}$$

$$\wedge \textit{expression1} = \textit{expression2}$$

$$\wedge \textit{expression2} = \textit{expression3}$$

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 $\textit{expression0} = \textit{expression1}$ is a theorem. The “hint 1” is supposed to make it clear that $\textit{expression1} = \textit{expression2}$ is a theorem. And so on. By the transitivity of $=$, this proof proves the theorem $\textit{expression0} = \textit{expression3}$.

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 correctness is not a matter of opinion.

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

$$a \wedge b \Rightarrow c \equiv a \Rightarrow (b \Rightarrow c)$$

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

$$\begin{aligned} & a \wedge b \Rightarrow c && \text{Material Implication} \\ \equiv & \neg(a \wedge b) \vee c && \text{Duality} \\ \equiv & \neg a \vee \neg b \vee c && \text{Material Implication} \\ \equiv & a \Rightarrow \neg b \vee c && \text{Material Implication} \\ \equiv & a \Rightarrow (b \Rightarrow c) \end{aligned}$$

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

$$a \wedge b \Rightarrow c \equiv \neg(a \wedge b) \vee c$$

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

$$a \Rightarrow b \equiv \neg a \vee b$$

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

Here is the proof again, in a different form.

$$\begin{aligned} & (a \wedge b \Rightarrow c \equiv a \Rightarrow (b \Rightarrow c)) && \text{Material Implication, 3 times} \\ \equiv & (\neg(a \wedge b) \vee c \equiv \neg a \vee (\neg b \vee c)) && \text{Duality} \\ \equiv & (\neg a \vee \neg b \vee c \equiv \neg a \vee \neg b \vee c) && \text{Reflexivity of } \equiv \\ \equiv & \top \end{aligned}$$

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

$$\begin{aligned} & (a \Rightarrow b \equiv a \wedge b) = a && \text{Associative Law for } = \\ \equiv & (a \Rightarrow b \equiv (a \wedge b \equiv a)) && \text{a Law of Inclusion} \\ \equiv & \top \end{aligned}$$

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

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

$$\begin{aligned} & \text{expression0} && \text{short hint} \\ \equiv & \text{expression1} && \text{and now a very long hint, written just as this is written,} \\ & && \text{on as many lines as necessary, followed by} \\ \equiv & \text{expression2} \end{aligned}$$

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

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

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

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

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

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

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

which can easily be rearranged to give the desired theorem.

The implication operator is reflexive $a \Rightarrow a$, antisymmetric $(a \Rightarrow b) \wedge (b \Rightarrow a) = (a = b)$, 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 ”. The words “stronger” and “weaker” may have come from a philosophical origin; we ignore any meaning they may have other than the boolean order, in which \perp is stronger than \top . For clarity and to avoid philosophical discussion, it would be better to say “falsier” rather than “stronger”, and to say “truer” rather than “weaker”, but we use the standard terms.

The Monotonic Law $a \Rightarrow b \Rightarrow c \wedge a \Rightarrow c \wedge b$ can be read (a little carelessly) as follows: if a is weakened to b , then $c \wedge a$ is weakened to $c \wedge b$. (To be more careful, we should say “weakened or equal”.) If we weaken a , then we weaken $c \wedge a$. Or, the other way round, if we strengthen b , then we strengthen $c \wedge b$. 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 $a \Rightarrow b \Rightarrow (b \Rightarrow c) \Rightarrow (a \Rightarrow c)$ says that whatever happens to an antecedent (weaken or strengthen), the opposite happens to the implication. We say that implication is antimonotonic in its antecedent.

Here are the monotonic and antimonotonic properties of boolean 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** is monotonic in b and monotonic in c

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

$$\begin{aligned} & \neg(a \wedge \neg(avb)) && \text{use the Law of Generalization} \\ \Leftarrow & \neg(a \wedge \neg a) && \text{now use the Law of Noncontradiction} \\ = & \top \end{aligned}$$

We thus prove that $\neg(a \wedge \neg(avb)) \Leftarrow \top$, and by an identity law, that is the same as proving $\neg(a \wedge \neg(avb))$. In other words, $\neg(a \wedge \neg(avb))$ is weaker than or equal to \top , and since there is

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

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

$$\begin{array}{l} a \wedge \neg(avb) \\ \Rightarrow a \wedge \neg a \\ = \perp \end{array} \quad \begin{array}{l} \text{use the Law of Generalization} \\ \text{now use the Law of Noncontradiction} \end{array}$$

This is called “proof by contradiction”. It proves $a \wedge \neg(avb) \Rightarrow \perp$, which is the same as proving $\neg(a \wedge \neg(avb))$. Any proof by contradiction can be converted to a proof by simplification to \top at the cost of one \neg sign per line.

End of Monotonicity and Antimonotonicity

1.0.3 Context

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

$$\begin{array}{l} \text{assumption} \\ \Rightarrow (\text{expression0} \\ = \text{expression1} \\ = \text{expression2} \\ = \text{expression3}) \end{array}$$

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

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

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

$$\text{assumption} \Rightarrow \text{expression0}$$

If the subproof is proving \perp , then the whole proof is proving

$$\text{assumption} \Rightarrow \perp$$

which is equal to $\neg\text{assumption}$. Once again, this is “proof by contradiction”.

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

$$\begin{array}{l} \text{if possibility} \\ \text{then (first subproof} \\ \quad \text{assuming possibility} \\ \quad \text{as a local axiom)} \\ \text{else (second subproof} \\ \quad \text{assuming } \neg\text{possibility} \\ \quad \text{as a local axiom)} \end{array}$$

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

$$\text{if possibility then something else anotherthing}$$

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

Consider a step in a proof that looks like this:

$$\begin{array}{l} \text{expression0} \wedge \text{expression1} \\ = \text{expression0} \wedge \text{expression2} \end{array}$$

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

$$\begin{aligned} & expression0 \wedge expression1 \\ = & expression2 \wedge expression1 \end{aligned}$$

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

$$\begin{aligned} & expression0 \wedge expression1 \\ = & expression2 \wedge expression3 \end{aligned}$$

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

$$\begin{aligned} & a \wedge a && \text{assume first } a \text{ to simplify second } a \\ = & a \wedge \top \end{aligned}$$

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

$$\begin{aligned} & a \wedge a && \text{assume second } a \text{ to simplify first } a \\ = & \top \wedge a \end{aligned}$$

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

$$\begin{aligned} & a \wedge a && \text{this step is wrong} \\ = & \top \wedge \top \end{aligned}$$

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

Here is a list of context rules for proof.

In $expression0 \wedge expression1$, when changing $expression0$, we can assume $expression1$.

In $expression0 \wedge expression1$, when changing $expression1$, we can assume $expression0$.

In $expression0 \vee expression1$, when changing $expression0$, we can assume $\neg expression1$.

In $expression0 \vee expression1$, when changing $expression1$, we can assume $\neg expression0$.

In $expression0 \Rightarrow expression1$, when changing $expression0$, we can assume $\neg expression1$.

In $expression0 \Rightarrow expression1$, when changing $expression1$, we can assume $expression0$.

In $expression0 \Leftarrow expression1$, when changing $expression0$, we can assume $expression1$.

In $expression0 \Leftarrow expression1$, when changing $expression1$, we can assume $\neg expression0$.

In **if** $expression0$ **then** $expression1$ **else** $expression2$, when changing $expression0$, we can assume $expression1 \neq expression2$.

In **if** $expression0$ **then** $expression1$ **else** $expression2$, when changing $expression1$, we can assume $expression0$.

In **if** $expression0$ **then** $expression1$ **else** $expression2$, when changing $expression2$, we can assume $\neg expression0$.

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

$$\begin{aligned} & \neg(a \wedge \neg(avb)) && \text{assume } a \text{ to simplify } \neg(avb) \\ = & \neg(a \wedge \neg(\top \vee b)) && \text{Symmetry Law and Base Law for } \vee \\ = & \neg(a \wedge \neg \top) && \text{Truth Table for } \neg \\ = & \neg(a \wedge \perp) && \text{Base Law for } \wedge \\ = & \neg \perp && \text{Boolean Axiom, or Truth Table for } \neg \\ = & \top \end{aligned}$$

1.0.4 Formalization

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

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

The best translation may not be a one-for-one substitution of symbols for words. The same word in different places may be translated to different symbols, and different words may be translated to the same symbol. The words “and”, “also”, “but”, “yet”, “however”, and “moreover” might all be translated as \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 Boolean Theory

1.1 Number Theory

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

a sequence of one or more decimal digits	
∞	“infinity”
$-x$	“minus x ”
$x + y$	“ x plus y ”
$x - y$	“ x minus y ”
$x \times y$	“ x times y ”
x / y	“ x divided by y ”
x^y	“ x to the power y ”
if a then x else y	

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

$x < y$	“ x is less than y ”
$x \leq y$	“ x is less than or equal to y ”
$x > y$	“ x is greater than y ”
$x \geq y$	“ x is greater than or equal to y ”
$x = y$	“ x equals y ”, “ x is equal to y ”
$x \neq y$	“ x differs from y ”, “ x is unequal to y ”

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

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

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

—End of Number Theory

1.2 Character Theory

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

—End of Character Theory

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

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

—End of Basic Theories

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

2 Basic Data Structures

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

unpacked, unindexed:	<u>bunch</u>
packaged, unindexed:	<u>set</u>
unpacked, indexed:	<u>string</u>
packaged, indexed:	<u>list</u>

2.0 Bunch Theory

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

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

From bunches A and B we can form the bunches

A, B “ A union B ”

$A \cap B$ “ A intersection B ”

and the number

$\#A$ “size of A ”, “cardinality of A ”

and the boolean

$A \in B$ “ A is in B ”, “ A is included in B ”

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

$$\#2 = 1$$

$$\#(0, 2, 5, 9) = 4$$

Here are three quick examples of bunch inclusion.

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

$$2: 2$$

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

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

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

$$x: y = x=y \quad \text{elementary axiom}$$

$$x: A, B = x: A \vee x: B \quad \text{compound axiom}$$

$$A, A = A \quad \text{idempotence}$$

$$A, B = B, A \quad \text{symmetry}$$

$A,(B,C) = (A,B),C$	associativity
$A'A = A$	idempotence
$A'B = B'A$	symmetry
$A'(B'C) = (A'B)'C$	associativity
$A,B: C = A: C \wedge B: C$	antidistributivity
$A: B'C = A: B \wedge A: C$	distributivity
$A: A,B$	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
$\phi x = 1$	size
$\phi(A, B) + \phi(A'B) = \phi A + \phi B$	size
$\neg x: A \Rightarrow \phi(A'x) = 0$	size
$A: B \Rightarrow \phi A \leq \phi B$	size

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

$A,(A'B) = A$	absorption
$A'(A,B) = A$	absorption
$A: B \Rightarrow C,A: C,B$	monotonicity
$A: B \Rightarrow C'A: C'B$	monotonicity
$A: B = A,B = B = A = A'B$	inclusion
$A,(B,C) = (A,B),(A,C)$	distributivity
$A,(B'C) = (A,B)'(A,C)$	distributivity
$A'(B,C) = (A'B), (A'C)$	distributivity
$A'(B'C) = (A'B)'(A'C)$	distributivity
$A: B \wedge C: D \Rightarrow A,C: B,D$	conflation
$A: B \wedge C: D \Rightarrow A'C: B'D$	conflation

Here are several bunches that we will find useful:

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

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

$$\begin{aligned} \neg \text{null} &= \text{null} \\ \neg(A, B) &= \neg A, \neg B \\ A + \text{null} &= \text{null} + A = \text{null} \\ (A, B) + (C, D) &= A + C, A + D, B + C, B + D \end{aligned}$$

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

We define the empty bunch, null , with the axioms

$$\begin{aligned} \text{null}: A \\ \phi A = 0 \quad \equiv \quad A = \text{null} \end{aligned}$$

This gives us three more laws:

$$\begin{aligned} A, \text{null} &= A && \text{identity} \\ A \phi \text{null} &= \text{null} && \text{base} \\ \phi \text{null} &= 0 && \text{size} \end{aligned}$$

The bunch bool is defined by the axiom

$$\text{bool} = \top, \perp$$

The bunch nat is defined by the two axioms

$$\begin{aligned} 0, \text{nat}+1: \text{nat} &&& \text{construction} \\ 0, B+1: B \Rightarrow \text{nat}: B &&& \text{induction} \end{aligned}$$

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.

$$\begin{aligned} \text{int} &= \text{nat}, \neg \text{nat} \\ \text{rat} &= \text{int}/(\text{nat}+1) \\ \text{xnat} &= \text{nat}, \infty \\ \text{xint} &= -\infty, \text{int}, \infty \\ \text{xrat} &= -\infty, \text{rat}, \infty \end{aligned}$$

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

$$\text{xreal} = -\infty, \text{real}, \infty$$

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

We also use the notation

$$x,..y \quad \text{“ } x \text{ to } y \text{” (not “ } x \text{ through } y \text{”)}$$

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

$$i: x,..y \quad \equiv \quad x \leq i < y$$

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

$$\begin{aligned} 0,.. \infty &= \text{nat} \\ 5,..5 &= \text{null} \\ \phi(x,..y) &= y-x \end{aligned}$$

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

2.1 Set Theory

optional

Let A be any bunch (anything). Then

$\{A\}$ “set containing A ”

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

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

$\sim S$ “contents of S ”

is its contents. For example,

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

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

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

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

$\{A\} \neq A$	structure
$\{\sim S\} = S$	set formation
$\sim\{A\} = A$	“contents”
$\$\{A\} = \phi A$	“size”, “cardinality”
$A \in \{B\} = A: B$	“elements”
$\{A\} \subseteq \{B\} = A: B$	“subset”
$\{A\}: \{B\} = A: B$	“power”
$\{A\} \cup \{B\} = \{A, B\}$	“union”
$\{A\} \cap \{B\} = \{A \text{ ' } B\}$	“intersection”
$\{A\} = \{B\} = A = B$	“equation”

—End of Set Theory

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

2.2 String Theory

The simplest string is

nil the empty string

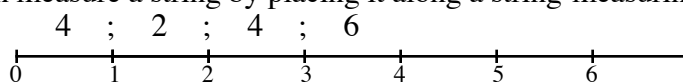
Any number, character, boolean, set, (and later also list and function) is a one-item string, or item. For example, the number 2 is a one-item string, or item. A nonempty bunch of items is also an item. Strings are catenated (joined) together by semicolons to make longer strings. For example,

$4; 2; 4; 6$

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

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

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



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

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

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

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

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

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

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

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

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

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

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

$$(3; 5; 9)\langle 2 \rangle 8 = 3; 5; 8$$

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

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

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

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

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

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

nil	the empty string
i	an item
$S;T$	“ S catenate T ”
S_T	“ S sub T ”
$n*S$	“ n copies of S ”
$S\langle n \rangle i$	“ S but at n there's i ”
are strings,	
$*S$	“copies of S ”
is a bunch of strings, and	
$\leftrightarrow S$	“length of S ”

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

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

$$\begin{array}{ll}
 nil; S = S; nil = S & \text{identity} \\
 S; (T; U) = (S; T); U & \text{associativity} \\
 \Leftrightarrow nil = 0 & \text{base} \\
 \Leftrightarrow i = 1 & \text{base} \\
 \Leftrightarrow (S; T) = \Leftrightarrow S + \Leftrightarrow T & \\
 S_{nil} = nil & \\
 \Leftrightarrow S < \infty \Rightarrow (S; i; T) \Leftrightarrow S = i & \\
 S_T; U = S_T; S_U & \\
 S_{(T; U)} = (S_T)_U & \\
 0^* S = nil & \\
 (n+1)^* S = n^* S; S & \\
 \Leftrightarrow S < \infty \Rightarrow S; i; T \triangleleft \Leftrightarrow S \triangleright j = S; j; T & \\
 \Leftrightarrow S < \infty \Rightarrow nil \leq S < S; i; T & \\
 \Leftrightarrow S < \infty \Rightarrow (i < j = S; i; T < S; j; U) & \\
 \Leftrightarrow S < \infty \Rightarrow (i=j = S; i; T = S; j; T) &
 \end{array}$$

We also use the notation

$$x;..y \quad \text{“ } x \text{ to } y \text{” (same pronunciation as } x..y \text{)}$$

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

$$\Leftrightarrow (x;..y) = y-x$$

Here are the axioms.

$$\begin{array}{l}
 x;..x = nil \\
 x;..x+1 = x \\
 (x;..y) ; (y;..z) = x;..z
 \end{array}$$

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

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

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

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

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

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

Perform the indexing and see what you get.

String catenation distributes over bunch union:

$$\begin{array}{l}
 A; null; B = null \\
 (A; B); (C; D) = A; C, A; D, B; C, B; D
 \end{array}$$

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

$$0; 1; 2: nat; 1; (0;..10)$$

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

$$0; 1; 2: nat; 1; (0;..10): 3*nat: *nat$$

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

$$\begin{aligned} \text{null} * A &= \text{null} \\ (A, B) * C &= A * C, B * C \end{aligned}$$

Using this left-distributivity, we define the one-operand $*$ by the axiom

$$*A = \text{nat} * A$$

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

—End of String Theory

2.3 List Theory

A list is a packaged string. For example,

$$[0; 1; 2]$$

is a list of three items. List brackets $[]$ distribute over bunch union.

$$\begin{aligned} [\text{null}] &= \text{null} \\ [A, B] &= [A], [B] \end{aligned}$$

Because $0: \text{nat}$ and $1: 1$ and $2: 0, \dots, 10$ we can say

$$[0; 1; 2]: [\text{nat}; 1; (0, \dots, 10)]$$

On the left of the colon we have a list of integers; on the right we have a list of bunches, or equivalently, a bunch of lists. A list is an element (elementary bunch) just when all its items are elements; $[0; 1; 2]$ is an element, but $[\text{nat}; 1; (0, \dots, 10)]$ is not. Progressing to larger bunches,

$$[0; 1; 2]: [\text{nat}; 1; (0, \dots, 10)]: [3 * \text{nat}]: [* \text{nat}]$$

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 ”
LM	“ LM ” or “ L composed with M ”
L^+M	“ L catenate M ”
$n \rightarrow i \mid L$	“ n maps to i otherwise L ”

are lists,

$\sim L$	“contents of L ”
----------	--------------------

is a string,

$\#L$	“length of L ”
-------	------------------

is an extended natural number, and

$L n$	“ $L n$ ” or “ L at index n ”
-------	-----------------------------------

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

The contents of a list is the string of items it contains.

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

The length of a list is the number of items it contains.

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

List indexes, like string indexes, start at 0. An item can be selected from a list by juxtaposing (sitting next to each other) a list and an index.

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

A list of indexes gives a list of selected items. For example,

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

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

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

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

$$2 \rightarrow 22 \mid [10; \dots; 15] = [10; 11; 22; 13; 14]$$

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

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

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

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

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

$[S] \neq S$	structure
$[\sim L] = L$	list formation
$\sim[S] = S$	contents
$\#[S] = \leftrightarrow S$	length
$[S] + [T] = [S; T]$	catenation
$[S] n = S_n$	indexing
$[S] [T] = [S_T]$	composition
$n \rightarrow i \mid [S] = [S \langle n \rangle i]$	modification
$[S] = [T] \iff S = T$	equation
$[S] < [T] \iff S < T$	order

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

$(L M) n = L (M n)$	
$(L M) N = L (M N)$	associativity
$L (M + N) = L M + L N$	distributivity

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

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

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

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

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

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

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

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

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

2.3.0 Multidimensional Structures

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

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

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

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

which can then be indexed again to give a number.

$$A\ 1\ 2 = 2$$

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

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

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

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

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

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

$$L@nil = L$$

$$L@n = L\ n$$

$$L@(S;T) = L@S@T$$

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

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

We generalize the notation $S \rightarrow i \mid L$ 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,

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

End of Multidimensional Structures

End of List Theory

End of Basic Data Structures

3 Function Theory

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

A variable is a name that is introduced for the purpose of instantiation (replacing it). For example, the law $x \times 1 = x$ 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 π and the axiom $3.14 < \pi < 3.15$, but we do not mean that every number is between 3.14 and 3.15. Similarly we might introduce the name i and the axiom $i^2 = -1$ and we do not want to instantiate i .

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

3.0 Functions

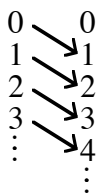
Let v be a name, let D be a bunch of items (possibly using previously introduced names but not using v), and let b be any expression (possibly using previously introduced names and possibly using v). Then

$\langle v: D \rightarrow b \rangle$ “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,

$\langle n: \text{nat} \rightarrow n+1 \rangle$

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



If f is a function, then

Δf “domain of f ”

is its domain. The Domain Axiom is

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

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

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

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

The substitution must replace every occurrence of v with w .

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

$$fx \quad \text{“} f \text{ applied to } x \text{” or “} f \text{ of } x \text{”}$$

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

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

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

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

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

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

$$max = \langle x: xrat \rightarrow \langle y: xrat \rightarrow \text{if } x \geq y \text{ then } x \text{ else } y \rangle \rangle$$

$$min = \langle x: xrat \rightarrow \langle y: xrat \rightarrow \text{if } x \leq y \text{ then } x \text{ else } y \rangle \rangle$$

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

$$max\ 3 = \langle y: xrat \rightarrow \text{if } 3 \geq y \text{ then } 3 \text{ else } y \rangle$$

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

$$max\ 3\ 5 = 5$$

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

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

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

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

$$divides = \langle n: nat+1 \rightarrow \langle i: int \rightarrow i/n: int \rangle \rangle$$

$$divides\ 2 = even$$

$$divides\ 2\ 3 = \perp$$

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

$$f|g \quad \text{“} f \text{ otherwise } g \text{”, “the selective union of } f \text{ and } g \text{”}$$

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

$$\Delta(f|g) = \Delta f, \Delta g$$

$$(f|g)\ x = \text{if } x: \Delta f \text{ then } f\ x \text{ else } g\ x$$

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

3.0.0 Abbreviated Function Notations

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

$$\langle x, y: x \text{ rat} \rightarrow \text{if } x \geq y \text{ then } x \text{ else } y \rangle$$

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

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

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

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

$$x+3$$

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

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

which introduces two variables, would have the same abbreviation as

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

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

End of Abbreviated Function Notations

3.0.1 Scope and Substitution

A variable is 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.

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

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

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

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

$$\langle y \rightarrow x \quad y \quad \langle x \rightarrow x \quad y \quad \rangle \quad x \quad y \quad \rangle x$$

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

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

by renaming, and then substitution gives

$$= \langle x \quad x \quad \langle z \rightarrow z \quad x \quad \rangle \quad x \quad x \quad \rangle$$

The Application Axiom (for element $x: D$)

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

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

$$\langle v: D \rightarrow b \rangle = \langle w: D \rightarrow \langle v: D \rightarrow b \rangle w \rangle$$

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

$$f = \langle w: D \rightarrow f w \rangle$$

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

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

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

End of Scope and Substitution

End of Functions

3.1 Quantifiers

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

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

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

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

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

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

$$\forall x: rat \cdot \forall y: rat \cdot x = y+1 \Rightarrow x > y$$

we can write

$$\forall x, y: rat \cdot x = y+1 \Rightarrow x > y$$

and in place of

$$\Sigma n: 0, \dots, 10 \cdot \Sigma m: 0, \dots, 10 \cdot n \times m$$

we can write

$$\Sigma n, m: 0, \dots, 10 \cdot n \times m$$

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

$$\forall v: null \cdot b = \top$$

$$\forall v: x \cdot b = \langle v: x \rightarrow b \rangle x$$

$$\forall v: A, B \cdot b = (\forall v: A \cdot b) \wedge (\forall v: B \cdot b)$$

$$\exists v: null \cdot b = \perp$$

$$\exists v: x \cdot b = \langle v: x \rightarrow b \rangle x$$

$$\exists v: A, B \cdot b = (\exists v: A \cdot b) \vee (\exists v: B \cdot b)$$

$$\Sigma v: null \cdot n = 0$$

$$\Sigma v: x \cdot n = \langle v: x \rightarrow n \rangle x$$

$$(\Sigma v: A, B \cdot n) + (\Sigma v: A' B' \cdot n) = (\Sigma v: A \cdot n) + (\Sigma v: B \cdot n)$$

$$\Pi v: null \cdot n = 1$$

$$\Pi v: x \cdot n = \langle v: x \rightarrow n \rangle x$$

$$(\Pi v: A, B \cdot n) \times (\Pi v: A' B' \cdot n) = (\Pi v: A \cdot n) \times (\Pi v: B \cdot n)$$

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

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

$$MIN\ v: null \cdot n = \infty$$

$$MIN\ v: x \cdot n = \langle v: x \rightarrow n \rangle x$$

$$MIN\ v: A, B \cdot n = \min (MIN\ v: A \cdot n) (MIN\ v: B \cdot n)$$

$$\begin{aligned} \text{MAX } v: \text{null} \cdot n &= -\infty \\ \text{MAX } v: x \cdot n &= \langle v: x \rightarrow n \rangle x \\ \text{MAX } v: A, B \cdot n &= \max (\text{MAX } v: A \cdot n) (\text{MAX } v: B \cdot n) \end{aligned}$$

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

$$\begin{aligned} \S v: \text{null} \cdot b &= \text{null} \\ \S v: x \cdot b &= \mathbf{if} \langle v: x \rightarrow b \rangle x \mathbf{ then } x \mathbf{ else } \text{null} \\ \S v: A, B \cdot b &= (\S v: A \cdot b), (\S v: B \cdot b) \end{aligned}$$

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

$$\S i: \text{int} \cdot i^2 = 4 = -2, 2 \quad \text{“those } i \text{ in } \text{int} \text{ such that ...”}$$

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

$$\S n: \text{nat} \cdot n < 3 = 0, \dots, 3$$

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

$$\begin{aligned} \forall v: D \cdot b \implies \langle v: D \rightarrow b \rangle x \\ \langle v: D \rightarrow b \rangle x \implies \exists v: D \cdot b \end{aligned}$$

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

$$\forall f \implies f x \implies \exists f$$

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

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

$$\begin{aligned} \forall v: D \cdot v = x \implies b &= \langle v: D \rightarrow b \rangle x \\ \exists v: D \cdot v = x \wedge b &= \langle v: D \rightarrow b \rangle x \end{aligned}$$

For instance, in the universal quantification $\forall n: \text{nat} \cdot n = 3 \implies n < 10$, 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, $\exists n: \text{nat} \cdot n = 3 \wedge n < 10$ 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:

$$\begin{aligned} &\forall x: \Delta P \cdot x = y \implies P x \\ = &\exists x: \Delta P \cdot x = y \wedge P x \\ = &\langle x: \Delta P \rightarrow P x \rangle y \\ = &P y \end{aligned}$$

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

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

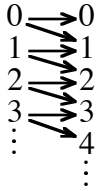
even though 0 is never a result of the function $\langle n: \text{nat} \rightarrow 1/(n+1) \rangle$.

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,

$\langle n: \text{nat} \rightarrow n, n+1 \rangle$
maps each natural number to two natural numbers.



Application works as usual:

$$\langle n: \text{nat} \rightarrow n, n+1 \rangle 3 = 3, 4$$

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 $\langle n: \text{nat} \rightarrow 0, \dots, n \rangle$ is both partial and nondeterministic.

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

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

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

$$f \text{ null} = \text{null}$$

$$f(A, B) = fA, fB$$

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

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

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

$$\begin{aligned} & \text{double}(2, 3) && \text{this step is right} \\ &= \text{double } 2, \text{double } 3 \\ &= 4, 6 \end{aligned}$$

but we cannot say

$$\begin{aligned} & \text{double}(2, 3) && \text{this step is wrong} \\ &= (2, 3) + (2, 3) \\ &= 4, 5, 6 \end{aligned}$$

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

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

$$\begin{aligned} & \langle n: \text{nat} \rightarrow n \times 2 \rangle (2, 3) && \text{this step is ok} \\ &= (2, 3) \times 2 \\ &= 4, 6 \end{aligned}$$

3.2.0 Function Inclusion and Equality

optional

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

$$f: g = \Delta g: \Delta f \wedge \forall x: \Delta g \cdot fx: gx$$

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

$$f = g = \Delta f = \Delta g \wedge \forall x: \Delta f \cdot fx = gx$$

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

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

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

$$\begin{aligned} max: xrat \rightarrow xrat \rightarrow xrat \\ min: xrat \rightarrow xrat \rightarrow xrat \\ even: int \rightarrow bool \\ odd: int \rightarrow bool \\ divides: (nat+1) \rightarrow int \rightarrow bool \end{aligned}$$

And, more generally,

$$f: A \rightarrow B = A: \Delta f \wedge fA: B$$

End of Function Inclusion and Equality

We earlier defined suc by the axiom

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

This equation can be written instead as

$$\Delta suc = nat \wedge \forall n: nat \cdot suc n = n+1$$

We could have defined suc by the weaker axiom

$$nat: \Delta suc \wedge \forall n: nat \cdot suc n = n+1$$

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 max , min , $even$, odd , and $divides$.

3.2.1 Higher-Order Functions

optional

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

$$\langle f: (A \rightarrow B) \rightarrow \dots f \dots \rangle g$$

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

$$\langle f: \{ (A \rightarrow B) \rightarrow \dots \sim f \dots \} \{ g \} \rangle$$

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

Here is a predicate whose parameter is a function.

$$\langle f: ((0, \dots, 10) \rightarrow int) \rightarrow \forall n: 0, \dots, 10 \cdot even(f n) \rangle$$

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

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

End of Higher-Order Functions

3.2.2 Function Composition

optional

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

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

$$(gf) x = g(fx)$$

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

$$\Delta(even\ suc) = \S x: \Delta suc \cdot suc\ x: \Delta even = \S x: nat \cdot x+1: int = nat$$

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

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

$$\begin{aligned} h f x g y & \text{ juxtaposition is left-associative} \\ = (((h f) x) g) y & \text{ use function composition on } h f \\ = ((h(fx)) g) y & \text{ use function composition on } (h(fx)) g \\ = (h(fx))(g y) & \text{ drop superfluous parentheses} \\ = h(fx)(g y) \end{aligned}$$

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

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

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

We can compose *double* with *suc*.

$$\begin{aligned} (double\ suc) 3 & \text{ use composition} \\ = double(suc\ 3) & \text{ apply } double \text{ to } suc\ 3 \\ = suc\ 3 + suc\ 3 \end{aligned}$$

From *double* we can form a new function

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

which can be applied to *suc*

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

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

$$\begin{aligned} (double\ suc) 3 & \text{ they “apply” } double \text{ to } suc \\ (suc + suc) 3 & \text{ then distribute } 3 \text{ to all occurrences of } suc \\ suc\ 3 + suc\ 3 & \text{ and get the right answer.} \end{aligned}$$

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

Like application, composition distributes over bunch union.

$$\begin{aligned} f(g, h) &= fg, fh \\ (f, g) h &= fh, gh \end{aligned}$$

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

$$(-f) x = -(f x)$$

In particular,

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

Similarly if p is a predicate, then

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

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

$$\neg even = odd$$

We can write the Duality Laws this way:

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

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

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

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

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

End of Function Composition

End of Function Fine Points

3.3 List as Function

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

$$-[3; 5; 2] = [-3; -5; -2]$$

$$suc [3; 5; 2] = [4; 6; 3]$$

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

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

just as we defined it for lists.

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

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

End of List as Function

3.4 Limits and Reals

optional

Let $f: \text{nat} \rightarrow \text{rat}$ so that $f_0; f_1; f_2; \dots$ is a sequence of rationals. The limit of the function (limit of the sequence) is expressed as $LIM f$. For example,

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

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

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

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

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

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

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

In general,

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

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

Now we can define the extended real numbers.

$$x: \text{xreal} = \exists f: \text{nat} \rightarrow \text{rat} \cdot x = LIM f$$

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

$$r: \text{real} = r: \text{xreal} \wedge -\infty < r < \infty$$

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

Let $p: \text{nat} \rightarrow \text{bool}$ so that p is a predicate and $p_0; p_1; p_2; \dots$ is a sequence of booleans. The limit of predicate p is defined by the axiom

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

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

$$\exists m \cdot \forall i \cdot i \geq m \Rightarrow p_i \Rightarrow LIM p$$

$$\exists m \cdot \forall i \cdot i \geq m \Rightarrow \neg p_i \Rightarrow \neg LIM p$$

For any particular assignment of values to (nonlocal) variables, the first implication says that $LIM p$ is \top if there is a point in the sequence $p_0 p_1 p_2 \dots$ past which p is always \top , and the second implication says that $LIM p$ is \perp if there is a point in the sequence past which p is always \perp .

For example,

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

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

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

End of Limits and Reals

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

$$\exists n: \text{nat} \cdot x = 2 \times n$$

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

End of Function Theory

4 Program Theory

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

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

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

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

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

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

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

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

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

4.0 Specifications

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

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

Specification S is <u>unsatisfiable</u> for prestate σ :	$\wp(\S\sigma' \cdot S) < 1$
Specification S is <u>satisfiable</u> for prestate σ :	$\wp(\S\sigma' \cdot S) \geq 1$
Specification S is <u>deterministic</u> for prestate σ :	$\wp(\S\sigma' \cdot S) \leq 1$
Specification S is <u>nondeterministic</u> for prestate σ :	$\wp(\S\sigma' \cdot S) > 1$

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 <u>implementable</u> :	$\forall\sigma \cdot \exists\sigma' \cdot S$
---	--

For convenience, we prefer to write specifications in the initial values x, y, \dots and final values x', y', \dots 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

$$\begin{aligned} & \forall x, y \cdot \exists x', y' \cdot x' = x+1 \wedge y' = y && \text{One-Point Law twice} \\ = & \forall x, y \cdot \top && \text{Identity Law twice} \\ = & \top \end{aligned}$$

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

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

$$x' > x$$

This specification is satisfied by a computation that increases x by any amount; it may leave y unchanged or may change it to any integer. This specification is nondeterministic for each initial state. Computer behavior satisfying the earlier specification $x' = x+1 \wedge y' = y$ 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 \perp , which is not satisfied by any computer behavior. But \perp 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; 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 we don't care what would happen if we did. That's what this specification says: for negative x any result is satisfactory. It allows an implementer to provide an error indication when x is initially negative. If we want a particular error indication, we can strengthen the specification to say so. We can describe the acceptable inputs as $x \geq 0$, but not the computer behavior. 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”, and a computing constant is also known as a “state constant”. A state variable x corresponds to two mathematical variables x and x' . A state constant is a single mathematical variable; it is there for instantiation, and it does not change state.

4.0.0 Specification Notations

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

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

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

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

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 boolean expressions, and they can be combined using any operators of Boolean 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** where b is a boolean expression of the initial state; it can be implemented by a computer that evaluates b , and then, depending on the value of b , behaves according to either S or R . The \vee and **if then else** 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 dependent 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 dependent composition is pronounced “dot”. This is not the same as the raised dot used in the abbreviated form of quantification.) Dependent composition is defined as follows.

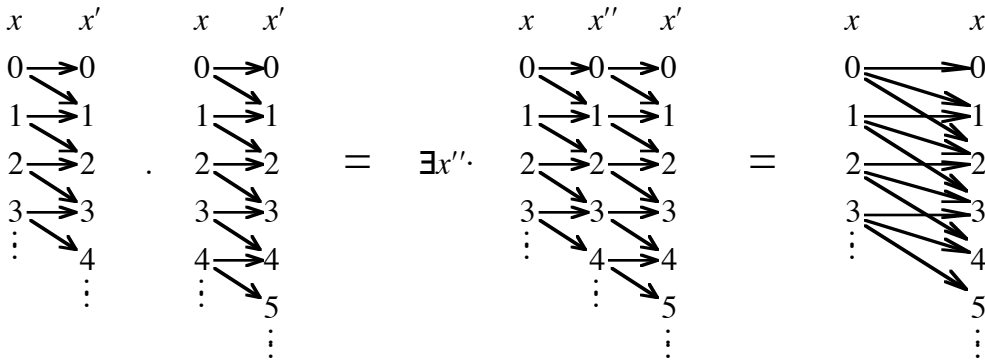
$$\begin{aligned} S.R &= \exists \sigma'' \cdot \langle \sigma' \rightarrow S \rangle \sigma'' \wedge \langle \sigma \rightarrow R \rangle \sigma'' \\ &= \exists x'', y'', \dots \langle x', y', \dots \rightarrow S \rangle x'' y'' \dots \wedge \langle x, y, \dots \rightarrow R \rangle x'' y'' \dots \\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

Here's 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.

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

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

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



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

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

—End of Specification Notations

4.0.1 Specification Laws

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

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

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

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

$$(\mathbf{if\ } b \mathbf{\ then\ } P \mathbf{\ else\ } Q). R = \mathbf{if\ } b \mathbf{\ then\ } (P. R) \mathbf{\ else\ } (Q. R) \quad \text{Distributive Law}$$

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

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

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

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

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

$$= y' > x'$$

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

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

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

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

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

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

(d) $x := 1. x \geq 1 \Rightarrow \exists x. y' = 2 \times x$

In $x \geq 1 \Rightarrow \exists x. y' = 2 \times x$, 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.

$$= 1 \geq 1 \Rightarrow \exists x. y' = 2 \times x$$

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

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

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 .

$$= x := y. x \geq 1 \Rightarrow \exists k. y' = x \times k$$

$$= y \geq 1 \Rightarrow \exists k. y' = y \times k$$

(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 .

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

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

(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).

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

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

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

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

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

(j) $x := 1. \text{if } y > x \text{ then } x := x + 1 \text{ else } x := y$

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

$$= \text{if } y > 1 \text{ then } x := 2 \text{ else } x := y$$

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

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

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

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

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

$$= x' > 2$$

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

—End of Specification Laws

4.0.2 Refinement

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

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

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

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

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

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

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.

$$\begin{aligned} x' > x &\Leftarrow x' = x + 1 \wedge y' = y \\ x' = x + 1 \wedge y' = y &\Leftarrow x := x + 1 \\ x' \leq x &\Leftarrow \text{if } x = 0 \text{ then } x' = x \text{ else } x' < x \\ x' > y' > x &\Leftarrow y := x + 1. \quad x := y + 1 \end{aligned}$$

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

End of Refinement

4.0.3 Conditions

optional

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

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

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

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

$$\begin{aligned} & \text{(the exact precondition for } x' > 5 \text{ to be refined by } x := x + 1 \text{)} \\ = & \quad \forall x'. x' > 5 \Leftarrow (x := x + 1) \\ = & \quad \forall x'. x' > 5 \Leftarrow x' = x + 1 && \text{One-Point Law} \\ = & \quad x + 1 > 5 \\ = & \quad x > 4 \end{aligned}$$

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

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

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

$$\begin{aligned} & \text{(the exact postcondition for } x > 4 \text{ to be refined by } x := x + 1 \text{)} \\ = & \quad \forall x. x > 4 \Leftarrow (x := x + 1) \\ = & \quad \forall x. x > 4 \Leftarrow x' = x + 1 && \text{One-Point Law} \\ = & \quad x' - 1 > 4 \\ = & \quad x' > 5 \end{aligned}$$

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

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

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

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

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

$$\begin{aligned} & \text{(the exact precondition for } abs\ x' > abs\ x \text{ to be refined by } x := x^2 \text{)} \\ = & \forall x'. abs\ x' > abs\ x \Leftarrow x' = x^2 && \text{One-Point Law} \\ = & abs\ (x^2) > abs\ x && \text{by the arithmetic properties of } abs\ x \text{ and } x^2 \\ = & x \neq -1 \wedge x \neq 0 \wedge x \neq 1 \end{aligned}$$

$$\begin{aligned} & \text{(the exact postcondition for } abs\ x' > abs\ x \text{ to be refined by } x := x^2 \text{)} \\ = & \forall x. abs\ x' > abs\ x \Leftarrow x' = x^2 && \text{after several steps including domain splitting and} \\ & \text{variable change and using the arithmetic properties of } abs\ x \text{ and } x^2 \\ = & x' \neq 0 \wedge x' \neq 1 \end{aligned}$$

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

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

$$\begin{aligned} C \wedge (P.Q) & \Leftarrow C \wedge P.Q \\ C \Rightarrow (P.Q) & \Leftarrow C \Rightarrow P.Q \\ (P.Q) \wedge C' & \Leftarrow P.Q \wedge C' \\ (P.Q) \Leftarrow C' & \Leftarrow P.Q \Leftarrow C' \\ P.C \wedge Q & \Leftarrow P \wedge C'.Q \\ P.Q & \Leftarrow P \wedge C'.C \Rightarrow Q \end{aligned}$$

Precondition Law:

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

Postcondition Law:

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

—End of Conditions

4.0.4 Programs

A program is a description or specification of computer behavior. A computer executes a program by behaving according to the program, by satisfying the program. People often confuse programs with computer behavior. They talk about what a program “does”; of course it just sits there on the page or screen; it is the computer that does something. They ask whether a program “terminates”; of course it does; it is the 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 disk head may crash, a compiler may have a bug, or a resource may become exhausted (stack overflow, number overflow), to mention a few. Then the difference between a program and the computer behavior is obvious.

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

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

For the “implemented expressions” referred to in (b) and (c), we take booleans, numbers, characters, and lists, with all their operators. We omit bunches, sets, and strings because we have lists, and we omit functions and quantifiers because they are harder to implement. All these notations, and others, are still welcome in specifications.

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

Here is an example refinement in one integer variable x .

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

The problem is $x \geq 0 \Rightarrow x' = 0$. The solution is **if** $x=0$ **then** ok **else** $(x := x-1. x \geq 0 \Rightarrow x' = 0)$. 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. A computer executes it by behaving according to the solution, and whenever the problem is encountered again, the behavior is again according to the solution.

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

$$\begin{aligned}
 & \text{if } x=0 \text{ then } ok \text{ else } (x := x-1. x \geq 0 \Rightarrow x' = 0) && \text{Replace } ok; \text{ Substitution Law} \\
 = & \text{if } x=0 \text{ then } x' = x \text{ else } x-1 \geq 0 \Rightarrow x' = 0 && \text{use context } x=0 \text{ to modify the } \mathbf{then}\text{-part} \\
 & \text{and use context } x \neq 0 \text{ and } x: \mathit{int} \text{ to modify the } \mathbf{else}\text{-part} \\
 = & \text{if } x=0 \text{ then } x \geq 0 \Rightarrow x' = 0 \text{ else } x \geq 0 \Rightarrow x' = 0 && \text{Case Idempotence} \\
 = & x \geq 0 \Rightarrow x' = 0
 \end{aligned}$$

—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 $A \Leftarrow \text{if } b \text{ then } C \text{ else } D$ and $C \Leftarrow E$ and $D \Leftarrow F$ are theorems,
then $A \Leftarrow \text{if } b \text{ then } E \text{ else } F$ is a theorem.

If $A \Leftarrow B.C$ and $B \Leftarrow D$ and $C \Leftarrow E$ are theorems, then $A \Leftarrow D.E$ is a theorem.

If $A \Leftarrow B$ and $B \Leftarrow C$ are theorems, then $A \Leftarrow C$ is a theorem.

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

Refinement by Parts (monotonicity, conflation)

If $A \Leftarrow \text{if } b \text{ then } C \text{ else } D$ and $E \Leftarrow \text{if } b \text{ then } F \text{ else } G$ are theorems,
then $A \wedge E \Leftarrow \text{if } b \text{ then } C \wedge F \text{ else } D \wedge G$ is a theorem.

If $A \Leftarrow B.C$ and $D \Leftarrow E.F$ are theorems, then $A \wedge D \Leftarrow B \wedge E.C \wedge F$ is a theorem.

If $A \Leftarrow B$ and $C \Leftarrow D$ are theorems, then $A \wedge C \Leftarrow B \wedge D$ is a theorem.

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

Refinement by Cases

$P \Leftarrow \text{if } b \text{ then } Q \text{ else } R$ is a theorem if and only if
 $P \Leftarrow b \wedge Q$ and $P \Leftarrow \neg b \wedge R$ are theorems.

As an example of Refinement by Cases, we can prove

$$x' \leq x \Leftarrow \text{if } x=0 \text{ then } x'=x \text{ else } x' < x$$

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

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

$$s := \Sigma L$$

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

$$s' = \Sigma L$$

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

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

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

$$s' = s + \Sigma L [n;..#L] \Leftarrow \begin{array}{l} \text{if } n = \#L \text{ then } n = \#L \Rightarrow s' = s + \Sigma L [n;..#L] \\ \text{else } n \neq \#L \Rightarrow s' = s + \Sigma L [n;..#L] \end{array}$$

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.

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

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

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

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

also uses the notation $n;..#L$, which is defined only for $n \leq \#L$. 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 an identifier. For example, the summation program looks like

$$\begin{aligned} A &\Leftarrow s:=0. n:=0. B \\ B &\Leftarrow \text{if } n=\#L \text{ then } C \text{ else } D \\ C &\Leftarrow ok \\ D &\Leftarrow s:=s+Ln. n:=n+1. B \end{aligned}$$

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

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

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

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

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

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

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

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

—End of List Summation

4.1.2 Binary Exponentiation

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

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

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 very worried that these superfluous assignments might in some way make the result wrong. With the theory, we only need to prove these six refinements, and we are confident that execution will not give us a wrong answer.

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

```
A ← if x=0 then B else C
B ← y:= 1. x:= 3
C ← D. E
D ← F. A
E ← y:= 2xy. x:= 5
F ← x:= x-1. y:= 7
```

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

```
A ← if x=0 then (y:= 1. x:= 3) else (x:= x-1. y:= 7. A. y:= 2xy. x:= 5)
```

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

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

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

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

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

—End of Binary Exponentiation

—End of Program Development

4.2 Time

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

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

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

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

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

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

- Replace each assignment $x := e$ by
 $t := t +$ (the time to evaluate and store e). $x := e$
- Replace each conditional **if** b **then** P **else** Q by
 $t := t +$ (the time to evaluate b and branch). **if** b **then** P **else** Q
- Replace each call P by
 $t := t +$ (the time for the call and return). P
 For a call that is implemented “in-line”, this time will be zero. For a call that is executed last in a refinement solution, it may be just the time for a branch. Sometimes it will be the time required to push a return address onto a stack and branch, plus the time to pop the return address and branch back.
- Each refined specification can include time. For example, let f be a function of the initial state σ . Then
 $t' = t + f\sigma$
 specifies that $f\sigma$ is the execution time,
 $t' \leq t + f\sigma$
 specifies that $f\sigma$ is an upper bound on the execution time, and
 $t' \geq t + f\sigma$
 specifies that $f\sigma$ is a lower bound on the execution time.

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

In Subsection 4.0.4 we considered an example of the form

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

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

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

This refinement is a theorem when

$$P = \mathbf{if} \ x \geq 0 \ \mathbf{then} \ x'=0 \ \wedge \ t' = t+3 \times x+1 \ \mathbf{else} \ t' = \infty$$

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

The same refinement

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

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

$$P = x'=0$$

$$P = \mathbf{if} \ x \geq 0 \ \mathbf{then} \ t'=t+3 \times x+1 \ \mathbf{else} \ t' = \infty$$

$$P = x'=0 \ \wedge \ \mathbf{if} \ x \geq 0 \ \mathbf{then} \ t'=t+3 \times x+1 \ \mathbf{else} \ t' = \infty$$

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

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

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

Three implementable specifications for which this is a theorem are

$$Q = t' = \infty$$

$$Q = x' = 2 \wedge t' = \infty$$

$$Q = x' = 3 \wedge t' = \infty$$

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 $xint$, and

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

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

In the recursive measure, our earlier example becomes

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

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

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

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

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

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

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

End of Recursive Time

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

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

where x is an integer variable, and

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

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

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

This exercise uses the functions div (divide and round down) and \log (binary logarithm). 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:

$$\begin{aligned}
x'=1 &\Leftarrow \text{if } x=1 \text{ then ok else } (x:= \text{div } x \ 2. \ t:= t+1. \ x'=1) \\
x \geq 1 &\Rightarrow t' \leq t + \log x \Leftarrow \text{if } x=1 \text{ then ok} \\
&\quad \text{else } (x:= \text{div } x \ 2. \ t:= t+1. \ x \geq 1 \Rightarrow t' \leq t + \log x) \\
x < 1 &\Rightarrow t' = \infty \Leftarrow \text{if } x=1 \text{ then ok else } (x:= \text{div } x \ 2. \ t:= t+1. \ x < 1 \Rightarrow t' = \infty)
\end{aligned}$$

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

$$\begin{aligned}
x'=1 &\Leftarrow \text{if } x=1 \text{ then } x'=x \wedge t'=t \text{ else } x'=1 \\
x \geq 1 &\Rightarrow t' \leq t + \log x \Leftarrow \text{if } x=1 \text{ then } x'=x \wedge t'=t \\
&\quad \text{else } \text{div } x \ 2 \geq 1 \Rightarrow t' \leq t + 1 + \log (\text{div } x \ 2) \\
x < 1 &\Rightarrow t' = \infty \Leftarrow \text{if } x=1 \text{ then } x'=x \wedge t'=t \text{ else } \text{div } x \ 2 < 1 \Rightarrow t' = \infty
\end{aligned}$$

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

$$\begin{aligned}
x'=1 &\Leftarrow x=1 \wedge x'=x \wedge t'=t \\
x'=1 &\Leftarrow x \neq 1 \wedge x'=1
\end{aligned}$$

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

$$\begin{aligned}
x < 1 &\Rightarrow t' = \infty \Leftarrow x=1 \wedge x'=x \wedge t'=t \\
x < 1 &\Rightarrow t' = \infty \Leftarrow x \neq 1 \wedge (\text{div } x \ 2 < 1 \Rightarrow t' = \infty)
\end{aligned}$$

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

$$\begin{aligned}
&(x'=1 \Leftarrow x=1 \wedge x'=x \wedge t'=t) && \text{by transitivity and specialization} \\
= &\top
\end{aligned}$$

Next,

$$\begin{aligned}
&(x'=1 \Leftarrow x \neq 1 \wedge x'=1) && \text{by specialization} \\
= &\top
\end{aligned}$$

Next,

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

Next comes the hardest one of the six.

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

The next one is easier.

$$\begin{aligned}
 & (x < 1 \Rightarrow t' = \infty \Leftarrow x = 1 \wedge x' = x \wedge t' = t) && \text{Law of Portation} \\
 = & t' = \infty \Leftarrow x < 1 \wedge x = 1 \wedge x' = x \wedge t' = t && \text{Put } x < 1 \wedge x = 1 \text{ together, and first Base Law} \\
 = & t' = \infty \Leftarrow \perp && \text{last Base Law} \\
 = & \top
 \end{aligned}$$

And finally,

$$\begin{aligned}
 & (x < 1 \Rightarrow t' = \infty \Leftarrow x \neq 1 \wedge (\text{div } x \ 2 < 1 \Rightarrow t' = \infty)) && \text{Law of Portation} \\
 = & t' = \infty \Leftarrow x < 1 \wedge (\text{div } x \ 2 < 1 \Rightarrow t' = \infty) && \text{Discharge} \\
 = & t' = \infty \Leftarrow x < 1 \wedge t' = \infty && \text{Specialization} \\
 = & \top
 \end{aligned}$$

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 dependent (sequential) composition following an infinite loop. Then the computation does start at time ∞ (in other words, it never starts), and we cannot expect it to stop before it starts. An implementable specification must be satisfiable with nondecreasing time for all initial states, even for initial time ∞ .

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

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

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 accurately reflected in the theory, which allows the same refinement constructions for (c) as for (a).

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

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

is not a theorem. We refine

$$x'=2 \wedge t' \leq t+1 \Leftarrow x:=2$$

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

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

—End of Termination

4.2.3 Soundness and Completeness

optional

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

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

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

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

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

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

$$P \Leftarrow Q$$

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

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

—End of Soundness and Completeness

4.2.4 Linear Search

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

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

$$\neg x: L(0,..h') \wedge (Lh'=x \vee h'=\#L) \wedge t' \leq t+\#L$$

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:

$$\neg x: L(0,..h') \wedge (Lh'=x \vee h'=\#L) \Leftarrow$$

$$h:=0. h \leq \#L \Rightarrow \neg x: L(h,..h') \wedge (Lh'=x \vee h'=\#L)$$

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

$$h \leq \#L \Rightarrow \neg x: L(h, ..h') \wedge (Lh'=x \vee h'=\#L) \Leftarrow$$

$$\text{if } h=\#L \text{ then ok else } h<\#L \Rightarrow \neg x: L(h, ..h') \wedge (Lh'=x \vee h'=\#L)$$

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

$$h < \#L \Rightarrow \neg x: L(h, ..h') \wedge (Lh'=x \vee h'=\#L) \Leftarrow$$

$$\text{if } Lh=x \text{ then ok else } (h:=h+1. h \leq \#L \Rightarrow \neg x: L(h, ..h') \wedge (Lh'=x \vee h'=\#L))$$

Now for the timing:

$$t' \leq t + \#L \Leftarrow h:=0. h \leq \#L \Rightarrow t' \leq t + \#L - h$$

$$h \leq \#L \Rightarrow t' \leq t + \#L - h \Leftarrow \text{if } h=\#L \text{ then ok else } h < \#L \Rightarrow t' \leq t + \#L - h$$

$$h < \#L \Rightarrow t' \leq t + \#L - h \Leftarrow \text{if } Lh=x \text{ then ok}$$

$$\text{else } (h:=h+1. t:=t+1. h \leq \#L \Rightarrow t' \leq t + \#L - h)$$

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

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

$$\neg x: L(0, ..h') \wedge (Lh'=x \vee h'=\#L) \wedge t' \leq t + \#L \Leftarrow$$

$$h:=0. h \leq \#L \Rightarrow \neg x: L(h, ..h') \wedge (Lh'=x \vee h'=\#L) \wedge t' \leq t + \#L - h$$

$$h \leq \#L \Rightarrow \neg x: L(h, ..h') \wedge (Lh'=x \vee h'=\#L) \wedge t' \leq t + \#L - h \Leftarrow$$

$$\text{if } h = \#L \text{ then ok}$$

$$\text{else if } Lh = x \text{ then ok}$$

$$\text{else } (h:=h+1. t:=t+1. h \leq \#L \Rightarrow \neg x: L(h, ..h') \wedge (Lh'=x \vee h'=\#L) \wedge t' \leq t + \#L - h)$$

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

$$\neg x: L(0, ..h') \wedge (Lh'=x \vee h'=\#L) \wedge t' \leq t + \#L \Leftarrow$$

$$h:=0. h < \#L \Rightarrow \neg x: L(h, ..h') \wedge (Lh'=x \vee h'=\#L) \wedge t' \leq t + \#L - h$$

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

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

$$\neg x: L(0, ..h') \wedge (Lh'=x \vee h'=\#L) \Leftarrow L:=L+[x]. h:=0. Q$$

$$Q \Leftarrow \text{if } Lh=x \text{ then ok else } (h:=h+1. Q)$$

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

4.2.5 Binary Search

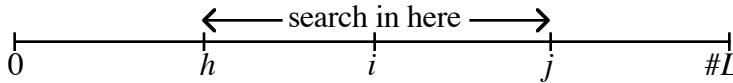
Exercise 154: Write a program to find a given item in a given nonempty sorted list. The execution time must be logarithmic in the length of the list. The strategy is to identify which half of the list contains the item if it occurs at all, then which quarter, then which eighth, and so on.

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

$$x: L(0, \dots, \#L) = p' \Rightarrow Lh' = x$$

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

$$R = (x: L(h, \dots, j) = p' \Rightarrow Lh' = x)$$



We can now solve the problem.

$$(x: L(0, \dots, \#L) = p' \Rightarrow Lh' = x) \Leftarrow h := 0. j := \#L. h < j \Rightarrow R$$

$$h < j \Rightarrow R \Leftarrow \text{if } j - h = 1 \text{ then } p := Lh = x \text{ else } j - h \geq 2 \Rightarrow R$$

$$j - h \geq 2 \Rightarrow R \Leftarrow \begin{array}{l} j - h \geq 2 \Rightarrow h' = h < i' < j = j' \\ \text{if } Li \leq x \text{ then } h := i \text{ else } j := i. \\ h < j \Rightarrow R \end{array}$$

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 $h; \dots, j$ into halves. To obtain the best execution time on average, we should choose i so it splits the segment $h; \dots, j$ 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 $h; \dots, j$ into two segments of equal size.

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

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

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

$$T \Leftarrow h := 0. j := \#L. U$$

$$U \Leftarrow \text{if } j - h = 1 \text{ then } p := Lh = x \text{ else } V$$

$$V \Leftarrow \begin{array}{l} i := \text{div}(h + j) 2. \\ \text{if } Li \leq x \text{ then } h := i \text{ else } j := i. \\ t := t + 1. U \end{array}$$

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

$$\begin{array}{rcl} \#L & = & 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 14\ 15\ 16\ 17\ 18\ \dots \\ t'-t & = & 0\ 1\ 2\ 2\ 3\ 3\ 3\ 3\ 4\ 4\ 4\ 4\ 4\ 4\ 4\ 5\ 5\ \dots \end{array}$$

from which we define

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

where ceil is the function that rounds up.

We can identify three levels of care in programming. At the lowest level, one writes programs without bothering to write clear specifications and refinements. At the next level, one writes clear and precise specifications and refinements as we have just done for binary search; with practice, one can quickly see the correctness of the refinements without bothering to write formal proofs. At the highest level of care, one proves each refinement formally; to achieve this level, an automated theorem prover is very helpful.

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

$$(x: L(0..\#L) = p' \Rightarrow Lh' = x) \Leftarrow h:=0. j:=\#L. h < j \Rightarrow R$$

we start with the right side.

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

The second refinement

$$h < j \Rightarrow R \Leftarrow \text{if } j-h = 1 \text{ then } p:=Lh=x \text{ else } j-h \geq 2 \Rightarrow R$$

can be proven by cases. And its first case is

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

The second case of the second refinement is

$$\begin{array}{l} (h < j \Rightarrow R \Leftarrow j-h \neq 1 \wedge (j-h \geq 2 \Rightarrow R)) \quad \text{portation} \\ = j-h \geq 2 \wedge (j-h \geq 2 \Rightarrow R) \Rightarrow R \quad \text{discharge} \\ = j-h \geq 2 \wedge R \Rightarrow R \quad \text{specialization} \\ = \top \end{array}$$

The next refinement

$$\begin{array}{l} j-h \geq 2 \Rightarrow R \Leftarrow j-h \geq 2 \Rightarrow h'=h < i' < j=j'. \\ \quad \text{if } Li \leq x \text{ then } h:=i \text{ else } j:=i. \\ \quad h < j \Rightarrow R \end{array}$$

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

$$\begin{aligned}
& (j-h \geq 2 \Rightarrow R \Leftarrow j-h \geq 2 \Rightarrow h'=h < i' < j=j'. Li \leq x \wedge (h:=i. h < j \Rightarrow R)) \quad \text{Condition} \\
\Leftarrow & (j-h \geq 2 \Rightarrow R \Leftarrow j-h \geq 2 \Rightarrow (h'=h < i' < j=j'. Li \leq x \wedge (h:=i. h < j \Rightarrow R))) \quad \text{Portation} \\
= & j-h \geq 2 \wedge (j-h \geq 2 \Rightarrow (h'=h < i' < j=j'. Li \leq x \wedge (h:=i. h < j \Rightarrow R))) \Rightarrow R \\
& \hspace{15em} \text{discharge } j-h \geq 2 \text{ and specialize} \\
\Leftarrow & (h'=h < i' < j=j'. Li \leq x \wedge (h:=i. h < j \Rightarrow R)) \Rightarrow R \quad \text{expand first } R \text{ and use Substitution} \\
= & (h'=h < i' < j=j'. Li \leq x \wedge (i < j \Rightarrow (x: L(i,..j) = p' \Rightarrow Lh' = x))) \Rightarrow R \\
& \hspace{15em} \text{dependent composition} \\
= & (\exists h'', i'', j'', p''. h''=h < i'' < j=j'' \wedge Li'' \leq x \\
& \hspace{10em} \wedge (i'' < j'' \Rightarrow (x: L(i'',..j'') = p' \Rightarrow Lh' = x))) \\
\Rightarrow & R \quad \text{eliminate } p'', h'', \text{ and } j'' \text{ by one-point, and rename } i'' \text{ to } i \\
= & (\exists i. h < i < j \wedge Li \leq x \wedge (i < j \Rightarrow (x: L(i,..j) = p' \Rightarrow Lh' = x))) \Rightarrow R \\
& \hspace{15em} \text{use context } i < j \text{ to discharge} \\
= & (\exists i. h < i < j \wedge Li \leq x \wedge (x: L(i,..j) = p' \Rightarrow Lh' = x)) \Rightarrow R \\
& \hspace{10em} \text{If } h < i \text{ and } Li \leq x \text{ and } L \text{ is sorted, then } x: L(i,..j) = x: L(h,..j) \\
= & (\exists i. h < i < j \wedge Li \leq x \wedge (x: L(h,..j) = p' \Rightarrow Lh' = x)) \Rightarrow R \\
& \hspace{10em} \text{note that } x: L(h,..j) = p' \Rightarrow Lh' = x \text{ is } R \\
& \hspace{15em} \text{since it doesn't use } i, \text{ bring it outside the scope of the quantifier} \\
= & (\exists i. h < i < j \wedge Li \leq x) \wedge R \Rightarrow R \quad \text{specialize} \\
= & \top
\end{aligned}$$

Its second case

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

is proven just like its first case.

The next refinement is

$$\begin{aligned}
& (j-h \geq 2 \Rightarrow h'=h < i' < j=j' \Leftarrow i:=div(h+j) 2) \quad \text{expand assignment} \\
= & (j-h \geq 2 \Rightarrow h'=h < i' < j=j' \Leftarrow i' = div(h+j) 2 \wedge p'=p \wedge h'=h \wedge j'=j) \\
& \hspace{10em} \text{use the equations in the antecedent as context to simplify the consequent} \\
= & (j-h \geq 2 \Rightarrow h = h < div(h+j) 2 < j = j \Leftarrow i' = div(h+j) 2 \wedge p'=p \wedge h'=h \wedge j'=j) \\
& \hspace{10em} \text{simplify } h=h \text{ and } j=j \text{ and use the properties of } div \\
= & (j-h \geq 2 \Rightarrow \top \Leftarrow i' = div(h+j) 2 \wedge p'=p \wedge h'=h \wedge j'=j) \quad \text{base law twice} \\
= & \top
\end{aligned}$$

The next refinement is

$$\begin{aligned}
& (T \Leftarrow h:=0. j:=\#L. U) \quad \text{replace } T \text{ and } U \\
= & (t' \leq t + \text{ceil}(\log(\#L)) \Leftarrow h:=0. j:=\#L. h < j \Rightarrow t' \leq t + \text{ceil}(\log(j-h))) \\
& \hspace{15em} \text{Substitution Law twice} \\
= & (t' \leq t + \text{ceil}(\log(\#L)) \Leftarrow 0 < \#L \Rightarrow t' \leq t + \text{ceil}(\log(\#L-0))) \\
= & \top
\end{aligned}$$

The next refinement

$$U \Leftarrow \text{if } j-h = 1 \text{ then } p:=Lh=x \text{ else } V$$

can be proven by cases. And its first case is

$$\begin{aligned}
& (U \Leftarrow j-h = 1 \wedge (p:=Lh=x)) \quad \text{expand } U \text{ and the assignment} \\
= & (h < j \Rightarrow t' \leq t + \text{ceil}(\log(j-h)) \Leftarrow j-h=1 \wedge p'=(Lh=x) \wedge h'=h \wedge i'=i \wedge j'=j \wedge t'=t) \\
& \hspace{10em} \text{use main antecedent as context in main consequent} \\
= & (h < j \Rightarrow t \leq t + \text{ceil}(\log 1) \Leftarrow j-h=1 \wedge p'=(Lh=x) \wedge h'=h \wedge i'=i \wedge j'=j \wedge t'=t) \\
& \hspace{15em} \text{Use } \log 1 = 0 \\
= & (h < j \Rightarrow \top \Leftarrow j-h=1 \wedge p'=(Lh=x) \wedge h'=h \wedge i'=i \wedge j'=j \wedge t'=t) \quad \text{base law twice} \\
= & \top
\end{aligned}$$

Its second case is

$$\begin{aligned}
& (U \Leftarrow j-h \neq 1 \wedge V) && \text{expand } U \text{ and } V \\
= & (h < j \Rightarrow t' \leq t + \text{ceil}(\log(j-h)) \Leftarrow j-h \neq 1 \wedge (j-h \geq 2 \Rightarrow t' \leq t + \text{ceil}(\log(j-h)))) && \text{portation} \\
= & h < j \wedge j-h \neq 1 \wedge (j-h \geq 2 \Rightarrow t' \leq t + \text{ceil}(\log(j-h))) \Rightarrow t' \leq t + \text{ceil}(\log(j-h)) && \text{simplify} \\
= & j-h \geq 2 \wedge (j-h \geq 2 \Rightarrow t' \leq t + \text{ceil}(\log(j-h))) \Rightarrow t' \leq t + \text{ceil}(\log(j-h)) && \text{discharge} \\
= & j-h \geq 2 \wedge t' \leq t + \text{ceil}(\log(j-h)) \Rightarrow t' \leq t + \text{ceil}(\log(j-h)) && \text{specialization} \\
= & \top
\end{aligned}$$

Before we prove the next refinement, we prove two little theorems first.

$$\begin{aligned}
& \text{if even } (h+j) \\
& \text{then } (\quad \text{div } (h+j) \ 2 < j \\
& \quad = (h+j)/2 < j \\
& \quad = j-h > 0 \\
& \quad \Leftarrow j-h \geq 2) \\
& \text{else } (\quad \text{div } (h+j) \ 2 < j \\
& \quad = (h+j-1)/2 < j \\
& \quad = j-h > -1 \\
& \quad \Leftarrow j-h \geq 2)
\end{aligned}$$

$$\begin{aligned}
& \text{if even } (h+j) \\
& \text{then } \quad 1 + \text{ceil}(\log(j - \text{div}(h+j) \ 2)) \\
& \quad = \text{ceil}(1 + \log(j - (h+j)/2)) \\
& \quad = \text{ceil}(\log(j-h)) \\
& \text{else } \quad 1 + \text{ceil}(\log(j - \text{div}(h+j) \ 2)) \\
& \quad = \text{ceil}(1 + \log(j - (h+j-1)/2)) \\
& \quad = \text{ceil}(\log(j-h+1)) \quad \text{If } h+j \text{ is odd then } j-h \text{ is odd and can't be a power of } 2 \\
& \quad = \text{ceil}(\log(j-h))
\end{aligned}$$

Finally, the last refinement

$$V \Leftarrow i := \text{div}(h+j) \ 2. \text{ if } Li \leq x \text{ then } h := i \text{ else } j := i. t := t+1. U$$

can be proven in two cases. First case:

$$\begin{aligned}
& (V \Leftarrow i := \text{div}(h+j) \ 2. Li \leq x \wedge (h := i. t := t+1. U)) \quad \text{drop } Li \leq x \text{ and replace } U \\
\Leftarrow & (V \Leftarrow i := \text{div}(h+j) \ 2. h := i. t := t+1. h < j \Rightarrow t' \leq t + \text{ceil}(\log(j-h))) \\
& \quad \text{then use Substitution Law three times} \\
= & (V \Leftarrow \text{div}(h+j) \ 2 < j \Rightarrow t' \leq t + 1 + \text{ceil}(\log(j - \text{div}(h+j) \ 2))) \\
& \quad \text{use the two little theorems} \\
\Leftarrow & (V \Leftarrow j-h \geq 2 \Rightarrow t' \leq t + \text{ceil}(\log(j-h))) \quad \text{definition of } V, \text{ reflexive Law} \\
= & \top
\end{aligned}$$

And the second case

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

is proven just like the first.

4.2.6 Fast Exponentiation

Exercise 151: Given rational variables x and z and natural variable y , write a program for $z' = x^y$ that runs fast without using exponentiation.

This specification does not say how fast the execution should be; let's make it as fast as we can. The idea is to accumulate a product, using variable z as accumulator. Define

$$P = z' = z \times x^y$$

We can solve the problem as follows, though this solution does not give the fastest possible computation.

$$\begin{aligned} z' = x^y &\Leftarrow z := 1. P \\ P &\Leftarrow \text{if } y=0 \text{ then ok else } y>0 \Rightarrow P \\ y>0 \Rightarrow P &\Leftarrow z := z \times x. y := y-1. P \end{aligned}$$

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.

$$\begin{aligned} y>0 \Rightarrow P &\Leftarrow \text{if even } y \text{ then even } y \wedge y>0 \Rightarrow P \text{ else odd } y \Rightarrow P \\ \text{even } y \wedge y>0 \Rightarrow P &\Leftarrow x := x \times x. y := y/2. P \\ \text{odd } y \Rightarrow P &\Leftarrow z := z \times x. y := y-1. P \end{aligned}$$

Each of these refinements is easily proven.

We have made the major improvement, but there are still several minor speedups. We make them partly as an exercise in achieving the greatest speed possible, and mainly as an example of program modification. To begin, if y is even and greater than 0, it is at least 2; after cutting it in half, it is at least 1; let us not waste that information. We re-refine

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

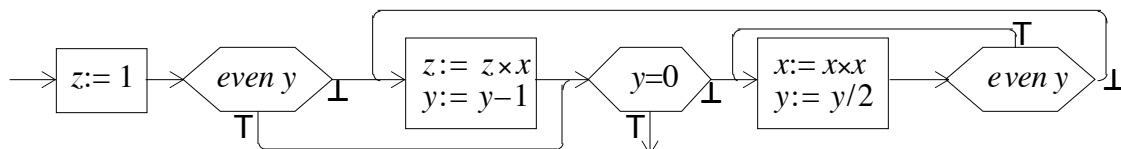
If y is initially odd and 1 is subtracted, then it must become even; let us not waste that information. We re-refine

$$\begin{aligned} \text{odd } y \Rightarrow P &\Leftarrow z := z \times x. y := y-1. \text{even } y \Rightarrow P \\ \text{even } y \Rightarrow P &\Leftarrow \text{if } y = 0 \text{ then ok else even } y \wedge y>0 \Rightarrow P \end{aligned}$$

And one more very minor improvement: if the program is used to calculate x^0 less often than x to an odd power (a reasonable assumption), it would be better to start with the test for evenness rather than the test for zeroness. We re-refine

$$P \Leftarrow \text{if even } y \text{ then even } y \Rightarrow P \text{ else odd } y \Rightarrow P$$

Program modification, whether to gain speed or for any other purpose, can be dangerously error-prone when practiced without the proper theory. Try writing this program in your favorite standard programming language, starting with the first simple solution, and making the same modifications. The first modification introduces a new case within a loop; the second modification changes one of the cases into an inner loop; the next modification changes the outer loop into a case within the inner loop, with an intermediate exit; the final modification changes the loop entry-point to a choice of two entry-points. The flow chart looks like this.



Without the theory, this sort of program surgery is bound to introduce a few bugs. With the theory we have a better chance of making the modifications correctly because each new refinement is an easy theorem.

Before we consider time, here is the fast exponentiation program again.

$$\begin{aligned}
 z' = x^y &\Leftarrow z := 1. P \\
 P &\Leftarrow \text{if even } y \text{ then } even\ y \Rightarrow P \text{ else } odd\ y \Rightarrow P \\
 even\ y \Rightarrow P &\Leftarrow \text{if } y=0 \text{ then } ok \text{ else } even\ y \wedge y>0 \Rightarrow P \\
 odd\ y \Rightarrow P &\Leftarrow z := z \times x. y := y-1. even\ y \Rightarrow P \\
 even\ y \wedge y>0 \Rightarrow P &\Leftarrow x := x \times x. y := y/2. y>0 \Rightarrow P \\
 y>0 \Rightarrow P &\Leftarrow \text{if even } y \text{ then } even\ y \wedge y>0 \Rightarrow P \text{ else } odd\ y \Rightarrow P
 \end{aligned}$$

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 $y: 2^n, \dots, 2^{n+1} \Rightarrow t' = t+n$, plus the isolated case $y=0 \Rightarrow t'=t$. We therefore propose the timing specification

$$\text{if } y=0 \text{ then } t'=t \text{ else } t' = t + \text{floor}(\log y)$$

where *floor* is the function that rounds down. We can prove this is the exact execution time, but it is easier to prove the less precise specification T defined as

$$T = \text{if } y=0 \text{ then } t'=t \text{ else } t' \leq t + \log y$$

To do so, we need to refine T with exactly the same refinement structure that we used to refine the result $z' = x^y$ so that we can conjoin the result and timing specifications according to Refinement by Parts. We can prove

$$\begin{aligned}
 T &\Leftarrow z := 1. T \\
 T &\Leftarrow \text{if even } y \text{ then } T \text{ else } y>0 \Rightarrow T \\
 T &\Leftarrow \text{if } y=0 \text{ then } ok \text{ else } y>0 \Rightarrow T \\
 y>0 \Rightarrow T &\Leftarrow z := z \times x. y := y-1. T \\
 y>0 \Rightarrow T &\Leftarrow x := x \times x. y := y/2. t := t+1. y>0 \Rightarrow T \\
 y>0 \Rightarrow T &\Leftarrow \text{if even } y \text{ then } y>0 \Rightarrow T \text{ else } y>0 \Rightarrow T
 \end{aligned}$$

It does not matter that specifications T and $y>0 \Rightarrow T$ are refined more than once. When we conjoin these specifications with the previous result specifications, we find that each specification is refined only once.

The timing can be written as a conjunction

$$(y=0 \Rightarrow t'=t) \wedge (y>0 \Rightarrow t' \leq t + \log y)$$

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.

4.2.7 Fibonacci Numbers

In this subsection, we tackle Exercise 217. The definition of the Fibonacci numbers

$$\begin{aligned} fib\ 0 &= 0 \\ fib\ 1 &= 1 \\ fib\ (n+2) &= fib\ n + fib\ (n+1) \end{aligned}$$

immediately suggests a recursive function definition

$$\begin{aligned} fib &= 0 \rightarrow 0 \mid 1 \rightarrow 1 \mid \langle n: nat+2 \rightarrow fib\ (n-2) + fib\ (n-1) \rangle \\ &= \langle n: nat \rightarrow \mathbf{if}\ n < 2 \mathbf{then}\ n \mathbf{else}\ fib\ (n-2) + fib\ (n-1) \rangle \end{aligned}$$

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

For $n \geq 2$, 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.

$$P \leftarrow \mathbf{if}\ n=0 \mathbf{then}\ (x:=0. y:=1) \mathbf{else}\ (n:=n-1. P. x'=y \wedge y'=x+y)$$

To move x and y along we need another variable. We could use a new variable, but we already have n ; is it safe to use n for this purpose? The specification $x'=y \wedge y'=x+y$ allows n to change, so we can use it if we want.

$$x'=y \wedge y'=x+y \leftarrow n:=x. x:=y. y:=n+y$$

The time for this solution is linear. To prove it, we keep the same refinement structure, but we replace the specifications with new ones concerning time. We replace P by $t' = t+n$ and add $t:=t+1$ in front of its use; we also change $x'=y \wedge y'=x+y$ into $t'=t$.

$$\begin{aligned} t' = t+n &\leftarrow \mathbf{if}\ n=0 \mathbf{then}\ (x:=0. y:=1) \mathbf{else}\ (n:=n-1. t:=t+1. t' = t+n. t'=t) \\ t'=t &\leftarrow n:=x. x:=y. y:=n+y \end{aligned}$$

Linear time is a lot better than exponential time, but we can do even better. Exercise 217 asks for a solution with logarithmic time. To get it, we need to take the hint offered in the exercise and use the equations

$$\begin{aligned} fib(2 \times k + 1) &= fib\ k^2 + fib(k+1)^2 \\ fib(2 \times k + 2) &= 2 \times fib\ k \times fib(k+1) + fib(k+1)^2 \end{aligned}$$

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

$$\begin{aligned} P &\leftarrow \mathbf{if}\ n=0 \mathbf{then}\ (x:=0. y:=1) \\ &\quad \mathbf{else}\ \mathbf{if}\ \mathit{even}\ n \mathbf{then}\ \mathit{even}\ n \wedge n > 0 \Rightarrow P \\ &\quad \mathbf{else}\ \mathit{odd}\ n \Rightarrow P \end{aligned}$$

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

$$\mathit{odd}\ n \Rightarrow P \leftarrow n := (n-1)/2. P. x' = x^2 + y^2 \wedge y' = 2 \times x \times y + y^2$$

The case $\mathit{even}\ n \wedge n > 0$ 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 $fib(2 \times k + 2)$.

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

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 .

$$\begin{aligned} x' = x^2 + y^2 \wedge y' = 2 \times x \times y + y^2 &\Leftarrow n := x. x := x^2 + y^2. y := 2 \times n \times y + y^2 \\ x' = 2 \times x \times y + y^2 \wedge y' = x^2 + y^2 + x' &\Leftarrow n := x. x := 2 \times x \times y + y^2. y := n^2 + y^2 + x \end{aligned}$$

To prove that this program is now logarithmic time, we define time specification

$$T = t' \leq t + \log(n+1)$$

and we put $t := t+1$ before calls to T . We must now prove

$$T \Leftarrow \text{if } n=0 \text{ then } (x:=0. y:=1) \text{ else if } \text{even } n \text{ then } \text{even } n \wedge n > 0 \Rightarrow T \text{ else } \text{odd } n \Rightarrow T$$

$$\text{odd } n \Rightarrow T \Leftarrow n := (n-1)/2. t := t+1. T. t'=t$$

$$\text{even } n \wedge n > 0 \Rightarrow T \Leftarrow n := n/2 - 1. t := t+1. T. t'=t$$

$$t'=t \Leftarrow n := x. x := x^2 + y^2. y := 2 \times n \times y + y^2$$

$$t'=t \Leftarrow n := x. x := 2 \times x \times y + y^2. y := n^2 + y^2 + x$$

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

$$(\text{odd } n \Rightarrow t' \leq t + \log(n+1)) \Leftarrow (n := (n-1)/2. t := t+1. t' \leq t + \log(n+1). t'=t)$$

$$= (\text{odd } n \Rightarrow t' \leq t + \log(n+1)) \Leftarrow t' \leq t+1 + \log((n-1)/2+1)$$

$$\text{note that } (a \Rightarrow b) \Leftarrow c = a \Rightarrow (b \Leftarrow c)$$

$$= \text{odd } n \Rightarrow (t' \leq t + \log(n+1) \Leftarrow t' \leq t+1 + \log((n-1)/2+1)) \quad \text{connection law}$$

$$\Leftarrow \text{odd } n \Rightarrow 1 + \log((n-1)/2+1) \leq \log(n+1) \quad \text{logarithm law}$$

$$= \text{odd } n \Rightarrow \log(n-1+2) \leq \log(n+1) \quad \text{arithmetic}$$

$$= \text{odd } n \Rightarrow \log(n+1) \leq \log(n+1) \quad \text{reflexivity and base}$$

$$= \top$$

$$(\text{even } n \wedge n > 0 \Rightarrow t' \leq t + \log(n+1)) \Leftarrow (n := n/2 - 1. t := t+1. t' \leq t + \log(n+1). t'=t)$$

by the same steps

$$= \text{even } n \wedge n > 0 \Rightarrow 1 + \log(n/2 - 1 + 1) \leq \log(n+1)$$

$$= \text{even } n \wedge n > 0 \Rightarrow \log n \leq \log(n+1)$$

$$= \top$$

—End of Fibonacci Numbers

Finding the execution time of any program can always be done by transforming the program into a function that expresses the execution time. To illustrate how, we do Exercise 216 (roller coaster), which is a famous program whose execution time is considered to be unknown. Let n be a natural variable. Then, including recursive time,

$$\begin{aligned} n'=1 &\Leftarrow \text{if } n=1 \text{ then } \text{ok} \\ &\quad \text{else if } \text{even } n \text{ then } (n := n/2. t := t+1. n'=1) \\ &\quad \text{else } (n := 3 \times n + 1. t := t+1. n'=1) \end{aligned}$$

It is not even known whether the execution time is finite for all $n > 0$.

We can express the execution time as fn , where function f must satisfy

$$\begin{aligned} t'=t+fn &\Leftarrow \text{if } n=1 \text{ then } \text{ok} \\ &\quad \text{else if } \text{even } n \text{ then } (n := n/2. t := t+1. t'=t+fn) \\ &\quad \text{else } (n := 3 \times n + 1. t := t+1. t'=t+fn) \end{aligned}$$

which can be simplified to

$$\begin{aligned} fn &= \text{if } n=1 \text{ then } 0 \\ &\quad \text{else if } \text{even } n \text{ then } 1 + f(n/2) \\ &\quad \text{else } 1 + f(3 \times n + 1) \end{aligned}$$

Thus we have an exact definition of the execution time. So why is the execution time considered to be unknown?

If the execution time of some program is n^2 , we consider that the execution time of that program is known. Why is n^2 accepted as a time bound, and $f n$ as defined above not accepted? Before answering, we suggest several non-reasons. The reason is not that f is defined recursively; the square function is defined in terms of multiplication, and multiplication is defined recursively. The reason cannot be that n^2 is well behaved (finite, monotonic, and smooth), while f jumps around wildly; every jump and change of value in f is there to fit the original program's execution time perfectly, and we shouldn't disqualify f just because it is a perfect bound. One might propose the length of time it takes to compute the time bound as a reason to reject f . Since it takes exactly as long to compute the time bound $f n$ as to run the program, we might as well just run the original program and look at our watch and say that's the time bound. But $\log \log n$ is accepted as a time bound even though it takes longer than $\log \log n$ to compute $\log \log n$.

The reason seems to be that function f is unfamiliar; it has not been well studied and we don't know much about it. If it were as well studied and familiar as square, we would accept it as a time bound.

We earlier looked at linear search in which we have to find the first occurrence of a given item in a given list. Suppose now that the list L is infinitely long, and we are told that there is at least one occurrence of the item x in the list. The desired result can be simplified to

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

and the program can be simplified to

$$\neg x: L(0, ..h') \wedge Lh'=x \iff h:=0. \neg x: L(h, ..h') \wedge Lh'=x$$

$$\neg x: L(h, ..h') \wedge Lh'=x \iff \text{if } Lh=x \text{ then ok else } (h:=h+1. \neg x: L(h, ..h') \wedge Lh'=x)$$

Adding recursive time, we can prove

$$t'=t+h' \iff h:=0. t'=t+h'-h$$

$$t'=t+h'-h \iff \text{if } Lh=x \text{ then ok else } (h:=h+1. t:=t+1. t'=t+h'-h)$$

The execution time is h' . Is this acceptable as a time bound? It gives us no indication of how long to wait for a result. On the other hand, there is nothing more to say about the execution time. The defect is in the given information: that x occurs somewhere, with no indication where.

End of Time

4.3 Space

Our example to illustrate space calculation is Exercise 212: the problem of the Towers of Hanoi. There are 3 towers and n disks. The disks are graduated in size; disk 0 is the smallest and disk $n-1$ is the largest. Initially tower A holds all n disks, with the largest disk on the bottom, 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.

$$\begin{aligned} \textit{MovePile } from \textit{ to using} &\Leftarrow \textbf{if } n=0 \textbf{ then } ok \\ &\textbf{else (} n:=n-1. \\ &\quad \textit{MovePile } from \textit{ using } to. \\ &\quad \textit{MoveDisk } from \textit{ to}. \\ &\quad \textit{MovePile } using \textit{ to } from. \\ &\quad n:=n+1 \textbf{)} \end{aligned}$$

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

$$\begin{aligned} t:=t+2^n-1 &\Leftarrow \textbf{if } n=0 \textbf{ then } ok \\ &\textbf{else (} n:=n-1. \\ &\quad t:=t+2^n-1. \\ &\quad t:=t+1. \\ &\quad t:=t+2^n-1. \\ &\quad n:=n+1 \textbf{)} \end{aligned}$$

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

$$\begin{aligned} &n=0 \wedge ok && \text{expand } ok \\ = &n=0 \wedge n'=n \wedge t'=t && \text{arithmetic} \\ \Rightarrow &t:=t+2^n-1 \end{aligned}$$

Second case, starting with its right side:

$$\begin{aligned} &n>0 \wedge (n:=n-1. t:=t+2^n-1. t:=t+1. t:=t+2^n-1. n:=n+1) \\ &\quad \text{drop conjunct } n>0; \text{ expand final assignment} \\ \Rightarrow &n:=n-1. t:=t+2^n-1. t:=t+1. t:=t+2^n-1. n'=n+1 \wedge t'=t \\ &\quad \text{use substitution law repeatedly from right to left} \\ = &n'=n-1+1 \wedge t'=t+2^{n-1}-1+1+2^{n-1}-1 && \text{simplify} \\ = &n'=n \wedge t'=t+2^n-1 \\ = &t:=t+2^n-1 \end{aligned}$$

To talk about the memory space used by a computation, we just add a space variable s . Like the time variable t , s is not part of the implementation, but only used in specifying and calculating space requirements. We use s for the space occupied initially at the start of execution, and s' for the space occupied finally at the end of execution. Any program may be used as part of a larger program, and it may not be the first part, so we cannot assume that the initial space occupied is 0, just as we cannot assume that a computation begins at time 0. In our example, the program calls itself recursively, and the recursive invocations begin at different times with different occupied space from the main (nonrecursive) invocation.

To allow for the possibility that execution endlessly consumes space, we take the domain of space to be the natural numbers extended with ∞ . Wherever space is being increased, we insert $s := s + (\text{the increase})$ to adjust s appropriately, and wherever space is being decreased, we insert $s := s - (\text{the decrease})$. 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

$$s' = s \Leftarrow \begin{array}{l} \text{if } n=0 \text{ then } ok \\ \text{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 \)} \end{array}$$

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

It is comforting to know there are no “space leaks”, but this does not tell us much about the space usage. There are two measures of interest: the maximum space occupied, and the average space occupied.

4.3.0 Maximum Space

Let m be the maximum space occupied before the start of execution (remember that any program may be part of a larger program that started execution earlier), and m' be the maximum space occupied by the end of execution. Wherever space is being increased, we insert $m := \max m \ s$ to keep m current. There is no need to adjust m at a decrease in space. In our example, we want to prove that the maximum space occupied is n . However, in a larger context, it may happen that the starting space is not 0, so we specify $m' = s+n$. We can assume that at the start $m \geq s$, since m is supposed to be the maximum value of s , but it may happen that the starting value of m is already greater than $s+n$, so the specification becomes $m \geq s \Rightarrow (m := \max m \ (s+n))$.

$$m \geq s \Rightarrow (m := \max m \ (s+n)) \Leftarrow \begin{array}{l} \text{if } n=0 \text{ then } ok \\ \text{else (} n := n-1. \\ \quad s := s+1. \ m := \max m \ s. \ m \geq s \Rightarrow (m := \max m \ (s+n)). \ s := s-1. \\ \quad ok. \\ \quad s := s+1. \ m := \max m \ s. \ m \geq s \Rightarrow (m := \max m \ (s+n)). \ s := s-1. \\ \quad n := n+1 \)} \end{array}$$

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

$$\begin{aligned}
& s := s+1. m := \max m s. m \geq s \Rightarrow (m := \max m (s+n)). s := s-1 \\
& \quad \text{Use a Condition Law, and expand final assignment} \\
\Rightarrow & s := s+1. m := \max m s. m \geq s \Rightarrow (m := \max m (s+n). s' = s-1 \wedge m' = m \wedge n' = n) \\
& \quad \text{Use Substitution Law} \\
= & s := s+1. m := \max m s. m \geq s \Rightarrow s' = s-1 \wedge m' = \max m (s+n) \wedge n' = n \\
& \quad \text{Use Substitution Law} \\
= & s := s+1. (\max m s) \geq s \Rightarrow s' = s-1 \wedge m' = \max (\max m s) (s+n) \wedge n' = n \\
& \quad \text{Simplify antecedent to } \top . \text{ Also } \max \text{ is associative} \\
= & s := s+1. s' = s-1 \wedge m' = \max m (s+n) \wedge n' = n \quad \text{use Substitution Law} \\
= & s' = s \wedge m' = \max m (s+1+n) \wedge n' = n \\
= & m := \max m (s+1+n)
\end{aligned}$$

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

$$\begin{aligned}
& n=0 \wedge ok \\
= & n' = n=0 \wedge s' = s \wedge m' = m \\
\Rightarrow & m \geq s \Rightarrow (m := \max m (s+n))
\end{aligned}$$

And second,

$$\begin{aligned}
& n > 0 \wedge (n := n-1. \\
& \quad s := s+1. m := \max m s. m \geq s \Rightarrow (m := \max m (s+n)). s := s-1. \\
& \quad ok. \\
& \quad s := s+1. m := \max m s. m \geq s \Rightarrow (m := \max m (s+n)). s := s-1. \\
& \quad n := n+1) \text{ Drop } n > 0 \text{ and } ok . \text{ Simplify long lines. Expand final assignment.} \\
\Rightarrow & n := n-1. m := \max m (s+1+n). m := \max m (s+1+n). n' = n+1 \wedge s' = s \wedge m' = m \\
& \quad \text{use Substitution Law three times} \\
= & n' = n \wedge s' = s \wedge m' = \max (\max m (s+n)) (s+n) \quad \text{associative and idempotent laws} \\
= & n' = n \wedge s' = s \wedge m' = \max m (s+n) \\
\Rightarrow & m \geq s \Rightarrow (m := \max m (s+n))
\end{aligned}$$

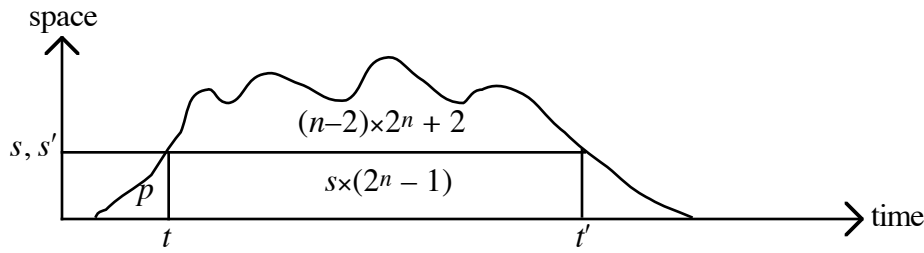
End of Maximum Space

4.3.1 Average Space

To find the average space occupied during a computation, we find the cumulative space-time product, and then divide by the execution time. Let p be the cumulative space-time product at the start of execution, and p' be the cumulative space-time product at the end of execution. We still need variable s , which we adjust exactly as before. We do not need variable t ; however, an increase in p occurs where there would be an increase in t , and the increase is s times the increase in t . In the example, where t was increased by 1, we now increase p by s . We prove

$$\begin{aligned}
p := p + s \times (2^n - 1) + (n-2) \times 2^n + 2 & \Leftarrow \\
\text{if } n=0 \text{ then } ok & \\
\text{else (} n := n-1. & \\
\quad s := s+1. p := p + s \times (2^n - 1) + (n-2) \times 2^n + 2. s := s-1. & \\
\quad p := p+s. & \\
\quad s := s+1. p := p + s \times (2^n - 1) + (n-2) \times 2^n + 2. s := s-1. & \\
\quad n := n+1) &
\end{aligned}$$

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:

$$\begin{aligned}
& n=0 \wedge ok && \text{expand } ok \\
= & n=0 \wedge n'=n \wedge s'=s \wedge p'=p && \text{arithmetic} \\
\Rightarrow & n'=n \wedge s'=s \wedge p' = p + s \times (2^n - 1) + (n-2) \times 2^n + 2 \\
= & p := p + s \times (2^n - 1) + (n-2) \times 2^n + 2 \\
& n > 0 \wedge (n := n-1. s := s+1. p := p + s \times (2^n - 1) + (n-2) \times 2^n + 2. s := s-1. n := n+1. \\
& \quad p := p+s. \\
& \quad n := n-1. s := s+1. p := p + s \times (2^n - 1) + (n-2) \times 2^n + 2. s := s-1. n := n+1) \\
& \quad \text{drop conjunct } n > 0 ; \text{ expand final assignment} \\
\Rightarrow & n := n-1. s := s+1. p := p + s \times (2^n - 1) + (n-2) \times 2^n + 2. s := s-1. n := n+1. p := p+s. \\
& n := n-1. s := s+1. p := p + s \times (2^{n-1}) + (n-2) \times 2^n + 2. s := s-1. n' = n+1 \wedge s' = s \wedge p' = p \\
& \quad \text{use substitution law 10 times from right to left} \\
= & n' = n \wedge s' = s \\
& \wedge p' = p + (s+1) \times (2^{n-1} - 1) + (n-3) \times 2^{n-1} + 2 + s + (s+1) \times (2^{n-1} - 1) + (n-3) \times 2^{n-1} + 2 \\
& \quad \text{simplify} \\
= & n' = n \wedge s' = s \wedge p' = p + s \times (2^n - 1) + (n-2) \times 2^n + 2 \\
= & p := p + s \times (2^n - 1) + (n-2) \times 2^n + 2
\end{aligned}$$

Instead of proving that the average space is exactly $n + n/(2^n - 1) - 2$, it is easier to prove that the average space is bounded above by n . To do so, instead of proving that the space-time product is $s \times (2^n - 1) + (n-2) \times 2^n + 2$, we would prove it is at most $(s+n) \times (2^n - 1)$. But we leave that as Exercise 212(f).

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

$$\begin{aligned}
\text{MovePile} & \Leftarrow \text{if } n=0 \text{ then } ok \\
& \text{else } (n := n-1. \\
& \quad s := s+1. m := \max m \text{ s. MovePile. } s := s-1. \\
& \quad t := t+1. p := p+s. ok. \\
& \quad s := s+1. m := \max m \text{ s. MovePile. } s := s-1. \\
& \quad n := n+1)
\end{aligned}$$

where *MovePile* is the specification

$$\begin{aligned}
& n' = n \\
& \wedge t' = t + 2^n - 1 \\
& \wedge s' = s \\
& \wedge (m \geq s \Rightarrow m' = \max m (s+n)) \\
& \wedge p' = p + s \times (2^n - 1) + (n-2) \times 2^n + 2
\end{aligned}$$

—End of Average Space

—End of Space

—End of Program Theory