SYSTEMATIC SOFTWARE DEVELOPMENT USING VDM

SECOND EDITION

SYSTEMATIC SOFTWARE DEVELOPMENT USING VDM

SECOND EDITION

CLIFF B JONES

The University, Manchester, England

Contents

Foreword to the First Edition

It is well known that ninety-nine percent of the world's problems are not susceptible to solution by scientific research. It is widely believed that ninety-nine percent of scientific research is not relevant to the problems of the real world. Yet the whole achievement and promise of modern technological society rests on the minute fraction of those scientific discoveries which are both useful and true.

The problems involved in the development and use of computer programs are well described and well appreciated, particularly by those who have suffered from them. Efforts of many researchers have been devoted to elucidate the theoretical basis of computer programming. This book shows how the results of the research are immediately and directly relevant in practical solution of the real problems of programming. By the clarity of its exposition, it makes the solution accessible to every student of computing science and to every professional programmer. A rigorous approach to software development will enable our society to reap the full benefit promised by the invention of the electronic digital computer.

Read it, study it, learn from it, enjoy it; but above all, put its lessons into practice.

C.A.R. Hoare
Oxford

Preface

> . . . the main achievement of the Alvey Software Engineering Programme is the success with which 'Formal Methods' from the academic world have been pulled through to industrial use. The implications of this achievement are difficult to overestimate, for these Formal Methods are the route to much better software writing, and the economic consequences will be considerable – on a par with those of the revolution in civil engineering in the last century.
> Brian Oakley

The aim of this book is to contribute to the wider use of formal methods in the specification and design of computer systems. VDM was developed in an industrial environment and is one of the most widely used formal methods. VDM is used in this book because it has achieved a level of maturity and acceptance: it has been taught for many years and has been used in a wide variety of applications. Furthermore, the British Standards Institution (BSI) work on developing a standard for VDM has been one of the stimuli for this revised edition.

This book teaches a particular systematic approach to software development concentrating on the stages from specification through design to implementation. The term *formal methods* embraces formal specification and verified design. Many aspects of a computer system must be specified including performance and cost. In this book attention is focused on *functional specification* (i.e. what the system does); the term *specification* is, however, used below without qualification for brevity. Formal specifications employ mathematical notation in order to achieve both precision and conciseness. A specification should be much shorter than a corresponding implementation. The key to brevity is *abstraction*. The specification of a system should abstract away issues which relate to implementation rather than to the intended behaviour of a computer system. The meaning of the operations are specified abstractly by recording properties which they should possess. Listing a collection of properties is one way in which a specification can be much shorter than an implementation. Another key technique for making specifications more concise than their implementations is to use abstract data objects which match the system being specified. This can be contrasted to the use of data objects which belong to the machine or language on which the system is to be implemented. The latter are the subject of implementations and should be avoided in specifications.

The other major aspect of formal methods is verified design. The idea that programs are mathematical texts opens up the possibility of reasoning about their formal relationship to specifications. Arguments can be constructed which, unlike the use of test cases, establish properties of programs in all cases. Verified design uses the concept of proof as a way of checking design steps. Steps in a systematic development can be based on, and verified against, a formal specification. This makes it possible to use proofs during the development process and to detect errors before work is based on a mistaken design step. The elimination of errors as early as possible is a key to improving the productivity of the development process and is a major argument for employing formal methods on industrial-sized applications. There is, of course, a difficulty in presenting the case for more formal methods in a textbook: the examples here are small and can be handled by *ad hoc* methods whereas the case for formality becomes strong on examples which are too large to be handled by traditional methods. The experience of other engineering disciplines supports the need to use soundly based methods on major tasks and the same experience is beginning to be gathered in the computer industry. One of the major rewards of employing formal methods in the development process is that the documentation created ensures that the systems are much easier to maintain.

The development of any large system has to be preceded by a specification of what is required. Without such a specification, the system's developers will have no firm statement of the needs of the would-be users of the system; these users will be the ones who, in more ways than one, are likely to pay for the inevitable misunderstandings. The need for precise specifications is accepted in most engineering disciplines. Computer systems are in no less need of precision than other engineering tasks. Unfortunately, the current practice in the software industry is to rely on specifications which use a mixture of informal text and pictures. Clear use of natural language obviously has a place in describing systems – but English cannot be relied upon as the sole specification language. In order to achieve precision, a specification must be written in a language which has a formal basis. Before the publication of the ALGOL report, the syntax of programming languages was given in *ad hoc* ways. Since the BNF (*Backus-Naur Form*) notation has been fully understood, no sensible language designer has described syntax in English sentences – however carefully written. The use of formal syntax meta-languages such as BNF has also made it possible to construct tools like parser generators. This history is encouraging but system description requires the specification of semantics as well as syntax. This book is intended to contribute to a parallel development for full semantic descriptions. Some notation and conventions beyond standard mathematics are useful in writing formal specifications. The VDM notation has itself been carefully defined. This has made it possible, for example, to establish the soundness of the proof rules for program design steps. VDM is not a closed body of material. An understanding of the underlying concepts makes it possible to introduce new notation (e.g. relations) if required for some class of applications. The more fluid areas of formal methods research tackle subjects like parallelism and are not addressed in this book.

This book is intended to be the basis of courses on formal methods. The material originally evolved from courses in industry. The first edition of the book has been used at university undergraduate level and in industrial courses. The only prerequisites are a knowledge of programming and some familiarity with discrete mathematics. The notation of both logic and set theory are reviewed but a reader who is totally unfamiliar with this material should first study a textbook such as *Set Theory and Related Topics* by S. Lipschutz published by McGraw-Hill.

The objective of this book is to bring students to the point where they can write and reason about small specifications written in – for example – VDM and read large specifications. Exposure to larger specifications and developments can be obtained from *Case Studies in Systematic Software Development* – also published by Prentice Hall International – studying such case studies will better equip people to tackle larger specifications themselves.

This edition contains eleven technical chapters. The first seven chapters are concerned with the specification notation but also include material on proofs about specifications themselves. Chapters 8–11 describe the material on verified design. The chapters are divided into numbered sections and the main ideas of a section can normally be presented in a one-hour lecture. The approach taken is formal with an emphasis on proof. It is possible to understand the material on specifications without following all of the proofs and one way of providing a short course is to omit Sections 1.3, 2.3, 3.2, 3.3, 4.2, 5.2, 6.2, 7.2, and

Chapters 8 to 11. However the study of proofs is rewarding, and experience shows that the specification notation is better understood if practice is gained with its manipulation via proofs. The study of a proof also deepens the appreciation of, and reinforces one's memory of, a theorem. Therefore, one should not omit the material in, for example, Sections 1.3 and 2.3 lightly because of the need to reinforce the properties of the logical operators. Chapter 12 contains a personal postscript. It has been said of the first edition that there are people for whom this is the only part of the book they have read. On balance, I should prefer that this happened more often than that a student should read the technical material but ignore the postscript.

The exercises are an integral part of the material and should be attempted by any reader who hopes to be able to use the methods described. Those exercises marked with an asterisk are more difficult and open-ended: they are best tackled in a group effort with some guidance available. Some mathematical comments which can be ignored – at least at first reading – are printed as footnotes. A Glossary of Symbols is provided in Appendix A. Technical terms are italicized at the point of introduction and are described in the Glossary of Terms (Appendix B). Appendices C–E contain summaries of technical material which is developed in the course of the book. Appendix F contains the relevant parts of the evolving BSI standard for VDM – it defines the concrete syntax of the notation used in this book (and [JS90]). As well as a bibliography, separate indices are given for functions/operations, types and general terms.

A set of *Teacher's Notes* is being printed which contains supporting material including answers to the unstarred exercises: these notes can be obtained from the publisher.

This is a major revision of the first edition of the book which both expands and simplifies the earlier treatment. A number of technical changes (which are discussed in the *Teacher's Notes*) have made it possible to simplify the proofs used. The material on logic notation and proofs has been streamlined. Chapter 10 – which addresses operation decomposition – has been completely rewritten and expanded; and the new Chapter 11 contains a small case study which indicates the way in which the steps of the VDM method relate to each other.

Acknowledgements

My sincere thanks go to the many people who have helped with the creation of this revision. The Balzac-like revisions have been patiently converted into presentable pages by Julie Hibbs. None of her labour have seen the light of day had not Mario Wolczko created and maintained his wonderful (LaTeX) `bsivdm.sty`. I am also grateful to Brian Monahan for suggesting the layout of inference rules used here. Comments on the first edition of the book have been made by many people including D. Andrews, John Fitzgerald, Ian Hayes, Andrew Malton, B. Monahan, M. Naftalin, R. Shaw, D. Simpson and many colleagues at Manchester University. Particular thanks go to Bo Stig Hansen for passing on the 'Reading Notes' from his course at the Technical University of Denmark and to the reviewers appointed by the publisher. Tony Hoare has provided inspiration and encouragement to produce this revision. Further inspiration has come from the meetings of IFIP Working Group 2.3. Financial support from the UK Science and Engineering Research Council (both through research grants and a Senior Fellowship award) and from the Wolfson Foundation is gratefully acknowledged. My thanks also go to Helen Martin of Prentice Hall International and to Ron Decent for his painstaking 'copy edit' of the text.

1

Logic of Propositions

> It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern both for applications and for elegance.
> John McCarthy

When writing formal specifications for computer systems, it is useful to adopt notation from established branches of mathematics. One of the most valuable bodies of notation is that of mathematical logic. A reason for the importance of logic notation is the central role it plays in the notion of proof. The use of specifications in the justification of design is described in Chapters 8 and 10. But, even when considering specifications on their own, there are a number of proofs to be performed. In this chapter, the basic ideas of propositional calculus are reviewed. After introducing the language, the concept of formal proof is introduced in Section 1.2 and expanded in Sections 1.3 (and, later, in Section 2.3).

1.1 Propositional operators

Propositions

A *proposition* is an expression which can have the value true or false. Thus, under the usual interpretation of arithmetic symbols, $2 + 3 = 5$ is true, but

$2+2=5$ is false. One way of forming propositions is, as in these examples, by writing relational operators (e.g. $=,<,\leq$) between arithmetic *terms* built up from, amongst other things, constants and operators. Another way of forming propositions is by using the truth-valued functions which are discussed in more detail in Section 2.1; for now, an intuitive reading of such function applications should suffice – thus $\textit{is-prime}(7)$ is true providing that the truth-valued function *is-prime* has the value true for exactly those natural numbers which are normally considered to be primes; whereas $\textit{is-prime}(8)$ is false.

The language which can be built by such propositions is rather limited if truths can only be stated about constants. One way to extend the language is to permit variables to occur in terms. The truth of expressions like $2+i=5$ depends on the value of i. Such expressions are called *predicates* in this book. The identifier i, in the example above, is said to be a *free variable*, and the interpretation for such free variables must come from their context. The truth of a predicate depends on the interpretation of its free variables; the example above is true only in contexts where i is bound to the value 3. A number of different ways of providing contexts, or bindings, for free variables are given in subsequent sections.

Operators

Propositions and predicates can be thought of as truth-valued expressions. Predicates simplify to propositions when their free variables are replaced by values. This section discusses the operators which are used to form composite truth-valued expressions; the operators are known as *propositional* operators. For example:

$$(2+3=5) \vee (2+2=5)$$

is a true proposition built by combining two simpler (constituent) propositions with a symbol which can be read as 'or'. Such propositional operators can be compared with familiar arithmetic operators ($+$,$*$ etc). Just like their arithmetic counterparts, propositional operators can be used in combinations to form long (or deeply nested) expressions; furthermore, they share the property that there are general laws about equivalence of expressions. Thus:

$$x+y=y+x$$

is the commutative law for addition and:

$$E_1 \vee E_2 \;\Leftrightarrow\; E_2 \vee E_1$$

could express the fact that 'or' is commutative.

Such laws apply even when predicates, or more complex logical expressions, are written in place of E_1 and E_2. In general, the E_i can be thought of as meta-variables which can be replaced by arbitrary logical expressions (in this section, truth-valued expressions built up from propositions; later, the constituents can be predicates).

Whereas arithmetic expressions are concerned with an infinite set of numeric values, propositions – when successfully evaluated – yield one of the two truth values. In recognition of the key role that George Boole played in the development of logic, these are often called *Boolean values*. This set is:

$$\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$$

(The term *logical value* is also used below.) The typography of the Boolean value **true** distinguishes it from the word 'true' used in a normal sentence. Strictly one should write 'E_i evaluates to **true**'; but, unless a special point has to be made, the briefer 'E_i is true' is used.

The fact that the set of Boolean values is finite provides a very simple way of expressing the meaning of logical operators: *truth tables* can be used to display the value of a compound proposition for all combinations of the possible values of constituent propositions. The reader is assumed to be familiar with the normal (two-valued) truth tables; the topic of truth tables is resumed in Section 3.3.

Some terminology is useful. An expression whose principal operator is 'or' is called a *disjunction*. Logical expressions whose principal operator is 'and' are called *conjunctions*. A *negation* is a logical expression whose principal operator is the symbol ($\neg$) for 'not'. An *implication* is a logical expression whose principal operator is 'implies'.[1] Its left-hand side is known as the *antecedent* and its right-hand side as the *consequent*.

It is possible to investigate properties of propositional operators via truth tables or 'models'. One useful law is that $E_1 \Rightarrow E_2$ has, for any propositions E_i, the same value as $\neg E_2 \Rightarrow \neg E_1$. This equivalence can be verified on a two-valued model as follows:

E_1	E_2	$E_1 \Rightarrow E_2$	$\neg E_1$	$\neg E_2$	$\neg E_2 \Rightarrow \neg E_1$
true	**true**	**true**	**false**	**false**	**true**
true	**false**	**false**	**false**	**true**	**false**
false	**true**	**true**	**true**	**false**	**true**
false	**false**	**true**	**true**	**true**	**true**

[1] Implications cause more confusion than the other constructs. Perhaps the easiest way to overcome the difficulty is to remember that $E_1 \Rightarrow E_2$ is equivalent to $\neg E_1 \vee E_2$.

operator	read as	priority
$\neg$	not	highest
$\wedge$	and	
$\vee$	or	
$\Rightarrow$	implies	
$\Leftrightarrow$	is equivalent to	lowest

Figure 1.1 Propositional operators and their precedence

Two logical expressions which have the same logical values are said to be *equivalent*. The basic truth tables are used in a step-by-step evaluation of larger expressions in the same way that complex arithmetic expressions are evaluated one step at a time. For example, in forming the column for $\neg E_2 \;\Rightarrow\; \neg E_1$, the operands of the implication are both negations.

The logical expression corresponding to the assertion of equality between arithmetic expressions is the *equivalence*. Remember that, although the operator does yield the value **true** exactly when the operands have the same value, a special symbol ($\Leftrightarrow$) is used to show that the equality is between Boolean values.

A list of the propositional operators is given in Figure 1.1. Just as in the construction of arithmetic expressions, the need for parentheses can be reduced by ranking the precedence of the operators. This order is also shown. In fact, the high precedence of the 'not' operator is used in examples above. More useful examples, such as writing:

$$\neg E_1 \vee E_2 \;\Rightarrow\; E_3 \wedge E_4 \vee E_5$$

for:

$$((\neg E_1) \vee E_2) \;\Rightarrow\; ((E_3 \wedge E_4) \vee E_5)$$

are encountered below.

Tautologies

Having established the language of propositional logic, it is possible to discuss general properties of expressions. Some logical expressions evaluate to **true** for all possible values of their constituent propositions. Such expressions are called *tautologies*. A simple example is:

$$E_1 \;\Rightarrow\; (\mathbf{false} \;\Rightarrow\; E_2)$$

The statement that an expression is a tautology is a *judgement* about the expression; such claims are not *per se* constructs of propositional logic. An obvious way of checking whether an expression is a tautology is by the construction of a truth table. One can, however, reason informally about such expressions. For example:

> the inner implication, which is the consequent of the principal implication, would be true for any value of E_2, thus the overall expression must be true for any value of E_1.

With practice, such arguments can be conducted safely on quite complex expressions. However, the final security for such arguments is that a completely formal check is possible: Section 1.3 provides a method which relies on the construction of formal proofs.

Some expressions which are not tautologies are false for all values of their constituent expressions: these are called *contradictions*. For example:

$$E \vee \mathbf{true} \;\Rightarrow\; E \wedge \mathbf{false}$$

is a trivial contradiction. Expressions which may be false or true, depending on their constituent propositions, are said to be *contingent*. For contingent expressions, the sets of values – of their constituent propositions – for which they are true can be found. For example:

$$E_1 \;\Rightarrow\; E_2 \wedge E_1$$

is true in any 'world' where E_1 is **false** or E_2 is **true** (or both). Each row of a truth table corresponds to a world. A tautology is an expression in which the result is true in every row; a contradiction has the result false in every row; and a contingent expression has some results true and others false.

Exercise 1.1.1 Replace the question marks with a Boolean value to make the following pairs of expressions equivalent:

$$\begin{array}{ll} E \wedge ? & E \\ E \wedge ? & \mathbf{false} \\ ? \;\Rightarrow\; E & \mathbf{true} \\ E \;\Rightarrow ? & \mathbf{true} \\ ? \;\Rightarrow\; E & E \\ E \;\Rightarrow ? & \neg\, E \end{array}$$

Write analogous rules to the first two for 'or'.

Exercise 1.1.2 Replace the question marks below by propositional operators so as to make the pairs of expressions equivalent (if in doubt, check using truth tables):

$E_1 \wedge E_2$	$E_2 \ ? \ E_1$
$E_1 \wedge (E_2 \wedge E_3)$	$(E_1 \ ? \ E_2) \ ? \ E_3$
$E_1 \wedge (E_2 \vee E_3)$	$E_1 \ ? \ E_2 \ ? \ E_1 \ ? \ E_3$
$\neg (E_1 \vee E_2)$	$? \ E_1 \ ? \ ? \ E_2$
$\neg \neg E$	$? \ E$
$E_1 \ \Rightarrow \ E_2$	$? \ E_2 \ \Rightarrow \ ? \ E_1$
$E_1 \ \Leftrightarrow \ E_2$	$(E_1 \ ? \ E_2) \wedge (E_2 \ ? \ E_1)$

Commutative and associative laws for conjunctions are given in the first and second cases. Write the equivalent laws for disjunctions.

Why are no parentheses required in the right-hand expression in the third line? This case shows the law for distributing 'and' over 'or'. Write the distributive law for 'or' over 'and'.

Exercise 1.1.3 (*) Inferences about conditional logical expressions follow from:

E	**if** E **then** E_1 **else** E_2
true	E_1
false	E_2

Conditional expressions can be used to define each of the propositional operators. Write the five definitions.

Sequents

A tautology is, in fact, a special case of a *sequent*:

$$\Gamma \vdash E$$

(where Γ is a list of logical expressions). Such a sequent records a judgement about the constituent logical expressions: it asserts that E can be deduced from Γ. Such formal proofs are introduced in the next section. The $\vdash$ symbol is often called a *turnstile*. The validity of sequents can be checked using truth tables. Once the *hypotheses* (elements of the list on the left of the turnstile) of the sequent have been evaluated, the *conclusion* (right-hand side) need only be evaluated in those rows where the hypotheses are all true. The truth table for $E_1 \wedge (E_2 \vee E_3) \ \vdash \ E_1 \wedge E_2 \ \vee \ E_1 \wedge E_3$, is given in Figure 1.2. Notice that this example needs a truth table with eight rows (because there are three sub-expressions), but that the conclusion of the sequent need not be evaluated for the last five rows. It is, however, also possible to reason informally. Considering the above sequent one can argue:

> For a sequent to be false there must be some world where its hypothesis is true and its conclusion false; if $E_1 \wedge (E_2 \vee E_3)$ is true, both

E_1	E_2	E_3	$E_1 \wedge (E_2 \vee E_3)$	$E_1 \wedge E_2 \vee E_1 \wedge E_3$
true	**true**	**true**	**true**	**true**
true	**true**	**false**	**true**	**true**
true	**false**	**true**	**true**	**true**
true	**false**	**false**	**false**	–
false	**true**	**true**	**false**	–
false	**true**	**false**	**false**	–
false	**false**	**true**	**false**	–
false	**false**	**false**	**false**	–

Figure 1.2 Example of sequent evaluation

> E_1 and at least one of E_2 or E_3 must be true; thus, either $E_1 \wedge E_2$ or $E_1 \wedge E_3$ (or both) must be true; therefore no world can be found in which the conclusion is false while the hypothesis is true.

Further examples of sequents are:

$$E_1 \Leftrightarrow E_2 \vdash E_1 \Rightarrow E_2$$
$$E_1 \wedge E_2 \vdash E_1 \Leftrightarrow E_2$$
$$\neg (E_1 \vee E_2) \vdash E_1 \Leftrightarrow E_2$$

A tautology is simply a sequent with no hypotheses – for example:

$$\vdash E_1 \Rightarrow (\textbf{false} \Rightarrow E_2)$$

Sequents can be formed with several hypotheses – in such cases they are separated by commas – thus:

$$E_1,\ E_2 \vee E_3 \vdash E_1 \wedge E_2 \vee E_1 \wedge E_3$$
$$E_1,\ E_2 \vdash E_1 \Rightarrow E_2$$
$$\neg E_1,\ \neg E_2 \vdash E_1 \Leftrightarrow E_2$$

There is a connection between the judgement that something is deducible (written with $\vdash$) and the implication (an expression of propositional calculus written with $\Rightarrow$). In fact, in classical logic the symbols turn out to be almost interchangeable. However, this is a result (the deduction theorem) which logic textbooks come to only after a thorough review of the different levels of language involved. In the full logic used in VDM, implications and sequents are anyway *not* interchangeable[2] so it is even more important to understand the distinction.

[2]The technical details of this point are explored in Section 3.3.

The implies symbol is a logical operator whose meaning is given by its truth table; implications can be nested giving rise to several occurrences of $\Rightarrow$ in one expression. A sequent is a statement or judgement about logical expressions: it records that the conclusion can be deduced from the hypotheses. With respect to the language of propositional (later, predicate) logic, the turnstile is a meta-symbol and sub-expressions containing turnstiles can not be used in forming larger logical expressions.

An example can be given to illustrate a rigorous argument. Two ways of stating that r is the absolute value of i are:

$$i < 0 \wedge r = -i \vee i \geq 0 \wedge r = i$$
$$(i < 0 \;\Rightarrow\; r = -i) \wedge (i \geq 0 \;\Rightarrow\; r = i)$$

To express that the second is a consequence of the first, write:

$$E_1 \wedge E_2 \vee \neg\, E_1 \wedge E_3 \;\vdash\; (E_1 \;\Rightarrow\; E_2) \wedge (\neg\, E_1 \;\Rightarrow\; E_3)$$

or, treating implication as an abbreviation:

$$E_1 \wedge E_2 \vee \neg\, E_1 \wedge E_3 \;\vdash\; (\neg\, E_1 \vee E_2) \wedge (E_1 \vee E_3)$$

In the case that $E_1 \wedge E_2$ is true, the first conjunct of the conclusion is true (because E_2 is the second disjunct) and so is the second conjunct (because E_1 is the first disjunct); therefore the conjunction is true; the case for $\neg\, E_1 \wedge E_3$ is similar.

The language which is built up from proposition variables (E_i) and the propositional operators is known as the *propositional logic*. This notation, together with rules for determining the truth of its expressions, forms the *propositional calculus*. A logical calculus in which the truth or falsity of any expression can be calculated is said to be *decidable*. The propositional calculus is decidable since there is a procedure to determine whether a sequent is true or not: it is only necessary to construct the truth table and evaluate the expressions using the truth tables given above. The truth tables provide a way in which propositional expressions can be evaluated. An alternative way of generating true statements is by selecting a small set of such statements from which all others can be generated; this is the proof-theoretic approach which occupies the rest of this chapter.

Exercise 1.1.4 Check which of the following represent true judgements (use truth tables or a rigorous argument recording any other results on which the argument relies):

$E_1 \vee E_2 \vdash E_1$
$E_1,\ E_2 \vdash E_1$
$E_1 \wedge E_2 \vdash E_1 \vee E_2$
$E_1 \vee E_2 \vdash E_1 \wedge E_2$
$E_2 \vdash E_1 \Rightarrow E_2$
$\neg E_1 \vdash E_1 \Rightarrow E_2$
$E_1 \Rightarrow E_2,\ E_1 \vdash E_2$
$\neg E_1 \vdash \neg (E_1 \wedge E_2)$
$\neg E_1 \vdash \neg (E_1 \vee E_2)$
$E_1 \wedge (E_2 \Leftrightarrow E_3) \vdash E_1 \wedge E_2 \Leftrightarrow E_1 \wedge E_3$
$E_1 \wedge E_2 \Leftrightarrow E_1 \wedge E_3 \vdash E_1 \wedge (E_2 \Leftrightarrow E_3)$

Exercise 1.1.5 (*) Write a truth table for an 'exclusive or' operator (i.e. similar to 'or' except that the result is **false** if both of the operands are **true**). Record some properties (as sequents) of this operator including its relation to equivalence.

1.2 Concept of proof

Section 1.1 mentions one way of formally verifying propositional expressions: truth tables provide a model theory for propositional calculus. Section 1.3 provides a *proof theory* for propositional calculus; in Section 2.3, this is extended to cover the predicate calculus. The combined proof theory is used throughout this book as proofs form a central part of the program development method described. One property of a formal specification is that proofs can be written which clarify its consequences; Chapters 8 to 11 use formal specifications as the basis of design: design steps give rise to 'proof obligations'. Once the formal methods are understood, most proof obligations can be discharged by rigorous arguments. Such arguments are, however, only safe if they are undertaken with a knowledge of how a formal proof could be constructed. It is, therefore, necessary to gain practice in the construction of such formal proofs. Furthermore, a study of the inference rules deepens one's understanding of the logical operators.

What is a proof?

It should be clear that the claim that something has been proved must eliminate doubt. Unfortunately, informal arguments cannot create such certainty. To provide a point of reference, consider the truth tables for propositional calculus discussed in Section 1.1. It is easy to construct a program to mechanize these in a way which determines the validity of sequents. Providing this program is correct, doubt about the validity of a sequent can always be eliminated by running the program. Foremost amongst the requirements for proofs, then, is that they should ensure certainty. In order to achieve this level of certainty with a proof, it is necessary to reduce proof construction to a 'game with symbols': each proof step must depend only on known (or proven) facts and be justified by one of a fixed set of inference rules. The inference rules themselves must require only the mechanical rearrangement of symbols. Such proofs are called *formal*.

But in order for them to be useful it must also be natural to both read and write proofs. It is difficult to be precise about what constitutes a natural proof. When an argument is presented informally, large steps are made without detailed justification. This is not, in itself, wrong. The aim of informal proof is to indicate how a proof could be constructed: the major steps are given in the knowledge that further details could be provided if these major steps are in doubt. Clearly, it is desirable that some overview of a formal proof can be achieved.

Another aspect of what constitutes a natural proof concerns the crucial distinction between the discovery and presentation of a proof. A proof is often found by working back from the goal; sub-goals are created and discharged until the sub-goals correspond to known facts. In order to show how the steps relate, it is normal to present an argument working forwards from the known facts towards the goal. This forward presentation is easier to read. But when readers become writers, it is unfortunate if they must learn to discover proofs one way and document their steps in a different order.

The style of proof presented in this chapter is known in logic textbooks as 'natural deduction'. The proofs are formal in the sense above. The inference rules essentially show how to generate true sequents from others. One claim to the adjective 'natural' is that there are introduction and elimination rules for each operator; more importantly the presentations enable a reader to understand the main steps in a proof; inner **from/infer** constructs present the detailed arguments for the major justification. The question of discovery (backward, goal-directed) versus presentation (forward) of proofs is not as easy to illustrate in a book as it is on the blackboard. The experience of teaching natural deduction proofs is, however, very encouraging and a style of proof discovery is investigated in some of the examples below.

Inference rules

Consider, for example, an inference rule for the introduction of a disjunction:

$$\boxed{\vee\text{-}I_l}\ \frac{E_2}{E_1 \vee E_2}$$

This states that, under the assumption that some logical expression (E_2) has been proved, then – as a conclusion – a disjunction of that logical expression with any other is also true (proved). As above, the E_i stand for arbitrary logical expressions: the rule is, in fact, a schema for many inferences. Thus, if at some point in a proof $post(x, f(x))$ has been established, then $\neg\, pre(x) \vee post(x, f(x))$ and thus (treating implication as an abbreviation) $pre(x) \;\Rightarrow\; post(x, f(x))$ is also true. There is a similar inference rule:

$$\boxed{\vee\text{-}I_r}\;\frac{E_1}{E_1 \vee E_2}$$

These two inference rules can be expressed together as:

$$\boxed{\vee\text{-}I}\;\frac{E_i}{E_1 \vee E_2}\; 1 \leq i \leq 2$$

The name ($\vee$-I) is a reminder that this rule justifies the introduction of disjunctions. Notice that the side condition shows that the known expression can be either the first or the second disjunct because the assumption is shown as E_i. The validity of such a rule follows from the truth tables: the resulting disjunction must be true in all worlds where one of its disjuncts is true. But the inference rule both corresponds to and strengthens one's intuition about the operator. This inference rule, and the ones which follow, are mechanical in the sense that they can be readily checked: if it is claimed that a step of a proof follows by 'or introduction' from an earlier step, then one or other of the disjuncts must exactly match the earlier expression.

In general, an inference rule has a list of hypotheses and a conclusion separated by a horizontal line. The rules contain expressions whose operands are the meta-variables (E_i) discussed above and this brings with it the notion of facts *matching* the expressions in the sense that there is a substitution (from meta-variables to expressions) which makes the expressions in the rules match the facts. If existing facts can be found which match (under a consistent substitution) each of the hypotheses, then a new fact can be generated which matches the conclusion. The use of the matching concept should be obvious but note that, when steps involve complex expressions, matching must observe the structure of an expression as determined by the priority of its operators. Thus $p \wedge q \vee r$ matches $E_1 \vee E_2$ but not $E_1 \wedge E_2$ because 'and' binds $p \wedge q$ to an operand of the weaker 'or'. In complex cases the reader might find it useful to insert extra parentheses.

How can conjunctions be generated in a proof? The 'and introduction' rule is:

$$\boxed{\wedge\text{-}I}\;\frac{E_1;\; E_2}{E_1 \wedge E_2}$$

from ...		
	$\vdots$	
5	p	
6	$r \wedge s$	
	$\vdots$	
8	$p \vee q$	$\vee$-I(5)
9	r	$\wedge$-E(6)
10	$r \wedge (p \vee q)$	$\wedge$-I(9,8)
infer ...		

Skeleton proof

Here, there are two hypotheses separated by a semicolon. If there are several hypotheses for an inference rule, they can be matched in any order with existing lines in a proof. The skeleton proof shown on page 12 shows how the matching for both of these rules works. (The line numbering is for illustration only.) Assuming that lines 5 and 6 have been established, the $\vee$-I rule can be used to establish line 8 – the justification on the right of the line shows both the name of the inference rule and the lines to which it is being applied; the $\wedge$-I rule is applied to lines 9 and 8 to justify line 10.) There are connections between inference rules and sequents which are discussed below. The reason that semicolons are used to separate hypotheses of inference rules (rather than commas as in sequents) is that some inference rules require sequents as hypotheses. The 'and introduction' inference rule states that, in order to conclude $E_1 \wedge E_2$, the conjuncts must both be proved separately. As one would expect, there is more work to be done to justify a conjunction than a disjunction. On the other hand, precisely because a conjunction is stronger, the rule which permits elimination of a conjunction ($\wedge$-E) shows that either conjunct is true:

$$\boxed{\wedge\text{-}E}\ \frac{E_1 \wedge E_2}{E_i}\ 1 \leq i \leq 2$$

Line 9 in the skeleton proof on page 12 is formed by $\wedge$-E.

from	$(p \wedge q) \wedge r$	
1	$p \wedge q$	$\wedge$-E(h)
2	p	$\wedge$-E(1)
3	q	$\wedge$-E(1)
4	r	$\wedge$-E(h)
5	$q \wedge r$	$\wedge$-I(3,4)
infer	$p \wedge (q \wedge r)$	$\wedge$-I(2,5)

Associativity of conjunction

Boxed proofs

Sequents can be used to show how the proof rules, which are given above, give rise to deductions. Thus, for example, if:

$$\Gamma \vdash p$$

has been proven, then $\vee$-I can be used to generate:[3]

$$\Gamma \vdash p \vee q$$

A list of sequents can be used to record a whole proof but when more than one hypothesis is involved, the presentation becomes clumsy. The natural deduction style, which is explained in detail in the next section, shows the dependencies on hypotheses at the head of a box beginning with the keyword **from**; the overall goal closes the box with the keyword **infer**; all of the numbered lines within the box are true under the assumptions of the embracing **from/infer** constructs. As an example of such a proof, 'and' is shown to be associative. An *associative* operator is one for which:

$$(x \mathbf{\ op\ } y) \mathbf{\ op\ } z = x \mathbf{\ op\ } (y \mathbf{\ op\ } z)$$

For 'and' it is necessary to show that:

$$(p \wedge q) \wedge r \vdash p \wedge (q \wedge r)$$

[3] The objective is to find enough inference rules so that all true statements of the model theory can be proven. That this is even achievable is not obvious. In fact, the notions of validity and provability are equivalent in a sense which is discussed in Section 3.3.

Thus the proof of this part of the associativity result for 'and' is presented in the natural deduction style on page 13. Such 'boxed proofs' recur throughout the remainder of the book. The sequent to be proved is shown as the outermost **from/infer** construct and the lines between show how the conclusion follows from the hypotheses. Each line is justified by writing, on the right, the inference rule which is used; in parentheses are listed either the line numbers or 'h' (for the hypothesis) of the expressions to which the rule is applied.

The earlier discussion about forward versus backward proof presentation can be seen in the associativity of conjunction example – although it is clearer on the less obvious results of the next section. The preceding discussion has been in terms of working forward from the known facts. But it would be possible to look at the form of the overall goal and generate (what became) lines 2 and 5 as sub-goals; lines 3 and 4 are then sub-goals to achieve line 5, and the 'and elimination' steps then become the obvious way of achieving the set of collected sub-goals. Such proofs could, of course, be presented in the reverse order (e.g. as **infer/from**). This is not done because one is writing proofs in order that they can be read. As pointed out above, it is easier to understand an argument which proceeds from known facts towards a goal.

The use of the rules as 'tactics' for decomposing a goal should not be viewed as an algorithm to generate proofs. As collections of derived rules are built up, many rules are applicable; some of them would lead to blind alleys.

The rules of the natural deduction game are that new lines can only be generated from earlier lines in the same, or some enclosing **from/infer**. As when generating truth tables, any logical expression can be substituted for the E_i. When proof rules are used in reasoning about programs, specific propositions are substituted for the E_i. In the proof of the associativity of 'and', no specific propositions are used and thus the proof that:

$$p \wedge (q \wedge r) \vdash (p \wedge q) \wedge r$$

justifies a new (derived) inference rule which can be used in subsequent proofs:

$$\boxed{\wedge\text{-}ass1}\ \frac{E_1 \wedge (E_2 \wedge E_3)}{(E_1 \wedge E_2) \wedge E_3}$$

For such elementary results, the level of detail needed here appears to be excessive. It is, however, worth remembering that a simple arithmetic result like $(i+j)+k = (k+j)+i$ would take several steps of commutativity and associativity to justify formally.

Another example where the reader should be familiar with the necessary formality is the use of concrete syntax descriptions of languages. Appendix F gives syntax rules for VDM and it can be seen that $p \wedge (q \wedge r)$ is a 'binary expression'.

To emphasize the link between this idea and the inference rules of logic, notice that the syntax rules could be written:

$$\boxed{\text{syntax1}}\frac{}{\wedge\colon\ binaryoperator}$$

$$\boxed{\text{syntax2}}\frac{E_1\colon expression;\ \ op\colon binaryoperator;\ \ E_2\colon expression}{E_1\, op E_2\colon\ \ infixexpression}$$

and so on.

1.3 Proofs in propositional calculus

The entire proof theory of propositional calculus can be based on very few rules. In order to minimize this number, this section treats as basic only 'or' and 'not' and defines the other operators in terms of these basic ones. An increasingly useful proof theory is constructed by justifying *derived rules*. Among other things, these rules facilitate manipulation of the defined operators 'and', 'implies' and 'equivalence' (all of the rules – basic and derived – needed in subsequent chapters are given in Appendix C).

Axiom 1.1 ($\vee$-I) The basic rules include one for introducing disjunctions:

$$\boxed{\vee\text{-}I}\frac{E_i}{E_1 \vee E_2}\ 1 \le i \le 2$$

Axiom 1.2 ($\neg\vee$-I) Negated disjunctions are introduced[4] by a rule which has two hypotheses. Intuitively, to know that neither E_1 nor E_2 are true, they must both be shown to be impossible:

$$\boxed{\neg\vee\text{-}I}\frac{\neg\, E_1;\ \neg\, E_2}{\neg\,(E_1 \vee E_2)}$$

Axiom 1.3 ($\vee$-E) The task of eliminating a disjunction involves reasoning by cases. If some result E can be deduced from E_1 and, independently, from E_2, then – if it can be shown that $E_1 \vee E_2$ holds – E must be true (without assumptions). Reading this axiom in a goal-directed way: one way to conclude E is to split the task into two cases and to show that their disjunction always holds and that E follows from either disjunct:

[4] The need for rules which combine 'not' and 'or' is discussed in Section 3.3.

$$\boxed{\vee\text{-}E}\ \frac{E_1 \vee E_2;\ E_1 \vdash E;\ E_2 \vdash E}{E}$$

Notice that the necessary subsidiary proofs are shown in the hypothesis as sequents. This rule gives rise to nested **from/infer** constructs in proofs (see the proof of the commutativity of 'or' on page 17 for an example).

Axiom 1.4 ($\neg \vee$-E) Eliminating a negated disjunction is easy; neither of the disjuncts can be true if their disjunction has been proved to be impossible:

$$\boxed{\neg \vee\text{-}E}\ \frac{\neg (E_1 \vee E_2)}{\neg E_i}\ 1 \le i \le 2$$

Double negations cancel each other in this logic so that two simple inference rules are:

$$\boxed{\neg\neg\text{-}I}\ \frac{E}{\neg\neg E}$$

$$\boxed{\neg\neg\text{-}E}\ \frac{\neg\neg E}{E}$$

Axiom 1.5 ($\neg\neg$-I/E) These two rules can be combined using a notation (a double horizontal line) for bi-directional rules which shows that it is valid to make the inference in either direction:

$$\boxed{\neg\neg\text{-}I/E}\ \genfrac{}{}{3pt}{}{E}{\neg\neg E}$$

Other rules (notably *contr*) are discussed when needed below.

Proving commutativity

The first formal proof to be undertaken in this section shows that 'or' is commutative ($E_1 \vee E_2 \vdash E_2 \vee E_1$). In contrast to the proof in the preceding section, this result and its proof use E_i for the propositions. The reader should check carefully the matches of these identifiers with the (similar) identifiers in the inference rules. The proof is given on page 17. Notice that this proof nests **from/infer** constructs; as with the overall proof, the inner constructs correspond to sequents and any steps within them to their justification. Here, the inner **from/infer** constructs represent subsidiary proofs of:

$E_1 \vdash E_2 \vee E_1$
$E_2 \vdash E_2 \vee E_1$

from $E_1 \vee E_2$		
1	**from** E_1	
	infer $E_2 \vee E_1$	$\vee$-I(h1)
2	**from** E_2	
	infer $E_2 \vee E_1$	$\vee$-I(h2)
infer $E_2 \vee E_1$		$\vee$-E(h,1,2)

Commutativity of disjunction: $\vee$-*comm*

from $(E_1 \vee E_2) \vee E_3$			
1	**from** $E_1 \vee E_2$		
1.1		**from** E_1	
		infer $E_1 \vee (E_2 \vee E_3)$	$\vee$-I(h1.1)
1.2		**from** E_2	
1.2.1		$E_2 \vee E_3$	$\vee$-I(h1.2)
		infer $E_1 \vee (E_2 \vee E_3)$	$\vee$-I(1.2.1)
	infer $E_1 \vee (E_2 \vee E_3)$		$\vee$-E(h1,1.1,1.2)
2	**from** E_3		
2.1	$E_2 \vee E_3$		$\vee$-I(h2)
	infer $E_1 \vee (E_2 \vee E_3)$		$\vee$-I(2.1)
infer $E_1 \vee (E_2 \vee E_3)$			$\vee$-E(h,1,2)

Associativity of disjunction (first part): $\vee$-*ass*

These are then used in the final 'or elimination'. Notice that a reference to the number of an inner **from/infer** construct (e.g. 1) refers to the implied sequent; the hypotheses of a **from/infer** construct can be referred to (cf. justification of the infer in 1) as h1.

Since the commutativity proof is general in E_i, a derived inference rule is made available for future proofs:

$$\boxed{\vee\text{-}comm}\ \frac{E_1 \vee E_2}{E_2 \vee E_1}$$

Derived rules can make proofs much clearer. Such derived rules are not, however, theoretically necessary since it would always be possible to generate appropriate additional steps in place of the use of the derived rule. In practice, they are needed to create a more natural level of reasoning.

Finding proofs

Another proof which relies heavily on $\vee$-E is that for the associativity of 'or':

$$(E_1 \vee E_2) \vee E_3 \vdash E_1 \vee (E_2 \vee E_3)$$

This proof is shown on page 17 and presents the opportunity to say more about the structure of natural deduction proofs. Clearly, the outermost box corresponds to the required sequent. Within the **from/infer** is a list of (numbered) lines which comprise a proof. The line numbering reflects the nesting of proofs. An inner **from/infer** construct is given one line number. In addition to its hypotheses written in the **from** line, those of any embracing **from/infer** construct can be used. Thus the subsidiary proof labelled 'line 1' represents:

$$(E_1 \vee E_2) \vee E_3,\ E_1 \vee E_2 \vdash E_1 \vee (E_2 \vee E_3)$$

and the five lines contained in its **from/infer** construct, represent its proof: each line in the proof corresponds to a true sequent.

The associativity proof on page 17 also provides an example in terms of which it is possible to give some indication of how natural deduction proofs are discovered. The overall goal is:

from $(E_1 \vee E_2) \vee E_3$	
$\vdots$	
infer $E_1 \vee (E_2 \vee E_3)$	?

Outer sequent

The question mark in the justification position of a line shows that it is yet to be proved. When a result has to be proved based on an assumption which is a disjunction, it is worth trying to prove the desired result from each disjunct (thus setting up a final $\vee$-E). Here, this case distinction heuristic gives rise to the nesting:

from $(E_1 \vee E_2) \vee E_3$			
3	**from** $E_1 \vee E_2$		
	$\vdots$		
	infer $E_1 \vee (E_2 \vee E_3)$		?
7	**from** E_3		
	$\vdots$		
	infer $E_1 \vee (E_2 \vee E_3)$		?
infer $E_1 \vee (E_2 \vee E_3)$			$\vee$-E(h,3,7)

Split by cases

There is a problem with numbering the lines when constructing a proof since other steps might have to be introduced. Here 3 and 7 are used to leave space since there might be other lines to be inserted prior to 3 or between the nested constructs 3 and 7. There is no necessity when searching for a proof to tackle the sub-goals in a fixed order; here, it is quite permissible to tackle sub-goal 7 first. One advantage of writing – in the justification – the applications of proof rules as though they were functions is that the applications can be nested; thus, the justification for the conclusion of 7 can be filled in as follows:

from $(E_1 \vee E_2) \vee E_3$			
3	**from** $E_1 \vee E_2$		
	$\vdots$		
	infer $E_1 \vee (E_2 \vee E_3)$		?
7	**from** E_3		
	infer $E_1 \vee (E_2 \vee E_3)$		$\vee$-I($\vee$-I(h7))
infer $E_1 \vee (E_2 \vee E_3)$			$\vee$-E(h,3,7)

Completion of one case

This should be compared with the construct labelled 2 in the complete proof on page 17. The only open step is now that labelled 3; since h3 is a disjunction, it is again subjected to (case) decomposition by 'or elimination':

from $(E_1 \vee E_2) \vee E_3$		
3	**from** $E_1 \vee E_2$	
3.2	**from** E_1	
	$\vdots$	
	infer $E_1 \vee (E_2 \vee E_3)$	?
3.4	**from** E_2	
	$\vdots$	
	infer $E_1 \vee (E_2 \vee E_3)$	?
	infer $E_1 \vee (E_2 \vee E_3)$	$\vee$-E(h3,3.2,3.4)
7	**from** E_3	
	$\vdots$	
	infer $E_1 \vee (E_2 \vee E_3)$	$\vee$-I($\vee$-I(h7))
infer $E_1 \vee (E_2 \vee E_3)$		$\vee$-E(h,3,7)

Nested case split

The relationship of this stage of the discovery process to the associativity proof given on page 17 should be clear.

One of the advantages of the natural deduction style is that the proofs can be read, from the outer level, inwards. With practice, this also becomes a way of constructing proofs. But the hints given are no more than heuristics: insight is needed in order to discover good proofs.

Exercise 1.3.1 The proof on page 17 only justifies one direction of the bi-directional associativity rule: prove the other half (i.e. $E_1 \vee (E_2 \vee E_3) \vdash (E_1 \vee E_2) \vee E_3$). This can be done either by aping the earlier proof or by using commutativity ($\vee$-*comm*) and the existing result.

Derived rules

Combining the result of Exercise 1.3.1 with the proof on page 17, the two parts of the associativity proof justify the following (bi-directional) derived rule.

Lemma 1.6 ($\vee$-*ass*) Disjunction is associative:

$$\boxed{\vee\text{-}ass}\ \frac{(E_1 \vee E_2) \vee E_3}{E_1 \vee (E_2 \vee E_3)}$$

Having established associativity, it is now possible to omit parentheses in expressions involving 'or' (at the same level). Thus, rather than write either:

$$(E_1 \vee E_2) \vee E_3$$

or:

$$E_1 \vee (E_2 \vee E_3)$$

it is permissible to write:

$$E_1 \vee E_2 \vee E_3$$

Furthermore, the 'or' introduction and elimination rules can be generalized as follows:

$$\boxed{\vee\text{-}I}\ \frac{E_i}{E_1 \vee \cdots \vee E_n}\ 1 \leq i \leq n$$

$$\boxed{\neg \vee\text{-}I}\ \frac{\neg E_1;\ \cdots;\ \neg E_n}{\neg (E_1 \vee \cdots \vee E_n)}$$

$$\boxed{\vee\text{-}E}\ \frac{E_1 \vee \cdots \vee E_n;\ E_1 \vdash E;\ \cdots;\ E_n \vdash E}{E}$$

$$\boxed{\neg \vee\text{-}E}\ \frac{\neg (E_1 \vee \cdots \vee E_n)}{\neg E_i}\ 1 \leq i \leq n$$

Many of the results established in this section are familiar from Section 1.1. It must, however, be realized that the proofs here are in no way argued from the truth tables: the formal proofs are conducted purely by playing the game with symbols. The certainty of correctness comes, here, from the fact that the game can be mechanized – a program can be written to check such proofs.

Defining conjunctions

Rule 1.7 ($\wedge$-*defn*) The 'and' operator can be introduced to the logic by the definition:

$$\boxed{\wedge\text{-}defn}\ \frac{\neg (\neg E_1 \vee \neg E_2)}{E_1 \wedge E_2}$$

Notice that this is a bi-directional rule. Having defined 'and', rules for its manipulation can be derived.

Lemma 1.8 ($\wedge$-*I*) Thus a proof is given on page 22 of:

$$\boxed{\wedge\text{-}I}\ \frac{E_1;\ E_2}{E_1 \wedge E_2}$$

from $E_1;\ E_2$		
1	$\neg\neg E_1$	$\neg\neg$-I(h)
2	$\neg\neg E_2$	$\neg\neg$-I(h)
3	$\neg(\neg E_1 \vee \neg E_2)$	$\neg\vee$-I(1,2)
infer $E_1 \wedge E_2$		$\wedge$-*defn*(3)

Introduction of conjunction: $\wedge$-I

from $\neg(E_1 \wedge E_2);\ \neg E_1 \vdash E;\ \neg E_2 \vdash E$		
1	$\neg\neg(\neg E_1 \vee \neg E_2)$	$\wedge$-*defn*(h)
2	$\neg E_1 \vee \neg E_2$	$\neg\neg$-E(1)
infer E		$\vee$-E(2,h,h)

Eliminating negated conjunctions: $\neg\wedge$-E

Here, the proof discovery process uses the only rule available to tackle the conclusion; this gives rise to the sub-goal at line 3. Line 3, in turn, matches the $\neg\vee$-I rule which gives rise to sub-goals 1 and 2; these are obvious candidates for the $\neg\neg$-I rule.

Exercise 1.3.2 Prove $E_1 \wedge E_2 \vdash E_i$ for $1 \leq i \leq 2$. (Hint: expand the conjunction and then use $\neg\vee$-E and $\neg\neg$-E).

Exercise 1.3.3 Prove $\neg E_i \vdash \neg(E_1 \wedge E_2)$ for $1 \leq i \leq 2$. (Hint: begin by using $\vee$-I and $\neg\neg$-I).

More proofs about conjunctions

Exercises 1.3.2 and 1.3.3 justify the derived rules known as $\wedge$-E and $\neg\wedge$-I.

Lemma 1.9 ($\neg\wedge$-E) The next rule to be justified (see page 22) is:

$$\boxed{\neg\wedge\text{-}E}\ \frac{\neg(E_1 \wedge E_2);\ \neg E_1 \vdash E;\ \neg E_2 \vdash E}{E}$$

Notice how the $\vee$-E uses the two sequents given in the overall hypothesis.

It is important to observe that the inference rules must not, in general, be applied to inner expressions; the hypotheses of deduction rules are intended to match whole lines in the proof (not arbitrary sub-expressions thereof). Ignoring this rule can result in invalid arguments. An exception to this restriction is that the definition rules (e.g. $\wedge$-*defn*) can be applied to arbitrary sub-expressions. (There are also special substitution rules – e.g. $\vee$-*subs* – derived below.)

The next two lemmas can both can be justified by repeated use of $\wedge$-E and $\wedge$-I.

Lemma 1.10 ($\wedge$-*comm*) The commutativity of 'and':

$$\boxed{\wedge\text{-}comm}\ \frac{E_1 \wedge E_2}{E_2 \wedge E_1}$$

Lemma 1.11 ($\wedge$-*ass*) The associativity of 'and':

$$\boxed{\wedge\text{-}ass}\ \frac{E_1 \wedge (E_2 \wedge E_3)}{\overline{(E_1 \wedge E_2) \wedge E_3}}$$

As with disjunctions, this justifies the use of the more general rules:

$$\boxed{\wedge\text{-}I}\ \frac{E_1;\ \cdots;\ E_n}{E_1 \wedge \cdots \wedge E_n}$$

$$\boxed{\neg\wedge\text{-}I}\ \frac{\neg\, E_i}{\neg\,(E_1 \wedge \cdots \wedge E_n)}\ 1 \le i \le n$$

$$\boxed{\wedge\text{-}E}\ \frac{E_1 \wedge \cdots \wedge E_n}{E_i}\ 1 \le i \le n$$

$$\boxed{\neg\wedge\text{-}E}\ \frac{\neg\,(E_1 \wedge \cdots \wedge E_n);\ \neg\, E_1 \vdash E;\ \cdots;\ \neg\, E_n \vdash E}{E}$$

There are many different ways of proving more advanced results. Although brevity is not itself the main touchstone of style, short proofs are often clearer than long ones.

Lemma 1.12 ($\wedge$-*subs*) A very helpful inference rule which provides a valid way of applying rules on inner sub-expressions is:

$$\boxed{\wedge\text{-}subs}\ \frac{E_1 \wedge \cdots \wedge E_i \wedge \cdots \wedge E_n;\ E_i \vdash E}{E_1 \wedge \cdots \wedge E \wedge \cdots \wedge E_n}$$

from $E_1 \vee E_2 \wedge E_3$		
1	**from** E_1	
1.1	$E_1 \vee E_2$	$\vee$-I(h1)
1.2	$E_1 \vee E_3$	$\vee$-I(h1)
	infer $(E_1 \vee E_2) \wedge (E_1 \vee E_3)$	$\wedge$-I(1.1,1.2)
2	**from** $E_2 \wedge E_3$	
2.1	$(E_1 \vee E_2) \wedge E_3$	$\wedge$-$subs$($\vee$-I)(h2)
	infer $(E_1 \vee E_2) \wedge (E_1 \vee E_3)$	$\wedge$-$subs$($\vee$-I)(2.1)
infer $(E_1 \vee E_2) \wedge (E_1 \vee E_3)$		$\vee$-E(h,1,2)

Distributivity of 'or' over 'and': $\vee\wedge$-$dist$

Its justification applies $\wedge$-E n times, the sequent from the hypotheses of the rule, and then n applications of $\wedge$-I. This rule can be used as, for example, $\wedge$-$subs(\vee$-$I)$ to deduce:

$$E_1 \wedge E_2 \wedge E_3 \vdash E_1 \wedge (E_2 \vee E) \wedge E_3$$

Lemma 1.13 ($\vee$-$subs$) Similarly, there is a derived rule:

$$\boxed{\vee\text{-}subs}\ \frac{E_1 \vee \cdots \vee E_i \vee \cdots \vee E_n;\ E_i \vdash E}{E_1 \vee \cdots \vee E \vee \cdots \vee E_n}$$

Its justification uses $\vee$-I in $n-1$ subsidiary proofs; with the sequent in one; followed by a final $\vee$-E step.

Lemma 1.14 The left distributive laws of propositional calculus are:

$$\boxed{\vee\wedge\text{-}dist}\ \frac{E_1 \vee E_2 \wedge E_3}{(E_1 \vee E_2) \wedge (E_1 \vee E_3)}$$

$$\boxed{\wedge\vee\text{-}dist}\ \frac{E_1 \wedge (E_2 \vee E_3)}{E_1 \wedge E_2 \vee E_1 \wedge E_3}$$

The general pattern of these proofs is similar; an example is shown on page 24. Distribution from the right is easy to prove – it relies on left distribution and commutativity.

Exercise 1.3.4 Prove $(E_1 \vee E_2) \wedge (E_1 \vee E_3) \vdash E_1 \vee (E_2 \wedge E_3)$. (Hint: remember to set up the final $\vee$-E).

Exercise 1.3.5 Prove $E_1 \wedge (E_2 \vee E_3) \vdash E_1 \wedge E_2 \vee E_1 \wedge E_3$. (Hint: use $\wedge$-E to find a disjunction on which to base an $\vee$-E).

Exercise 1.3.6 Prove $E_1 \wedge E_2 \vee E_1 \wedge E_3 \vdash E_1 \wedge (E_2 \vee E_3)$

de Morgan's laws

Some of these elementary proofs are surprisingly lengthy but, having built up useful derived rules, proofs of more interesting results do not get significantly longer. In particular, the proofs of de Morgan's laws are very short.

Exercise 1.3.7 Prove the results necessary to justify de Morgan's laws.

$$\boxed{\vee\text{-}deM}\ \frac{\neg(E_1 \vee E_2)}{\neg E_1 \wedge \neg E_2}$$

$$\boxed{\wedge\text{-}deM}\ \frac{\neg(E_1 \wedge E_2)}{\neg E_1 \vee \neg E_2}$$

Remember that both directions must be proved for bi-directional rules.

Defining implication

Rule 1.15 ($\Rightarrow$-*defn*) Implication can be defined:

$$\boxed{\Rightarrow\text{-}defn}\ \frac{\neg E_1 \vee E_2}{E_1 \;\Rightarrow\; E_2}$$

A key result about implication is known as *modus ponens*; its proof (see below) relies on a contradiction rule.

Axiom 1.16 (*contr*) The basic inference rule used is:

$$\boxed{contr}\ \frac{E_1;\ \neg E_1}{E_2}$$

The contradiction (*contr*) rule only makes sense in an environment with other assumptions: if, under some assumptions, both E_1 and its negation can be deduced, then there must be some contradiction in the assumptions and anything can be deduced.

from $E_1 \Rightarrow E_2;\ E_1$		
1	$\neg E_1 \vee E_2$	$\Rightarrow$-*defn*(h)
2	E_1	h
3	**from** $\neg E_1$	
	infer E_2	*contr*(2,h3)
4	**from** E_2	
	infer E_2	h4
infer E_2		$\vee$-E(1,3,4)

Modus ponens: $\Rightarrow$-E

Lemma 1.17 ($\Rightarrow$-E) The law known as *modus ponens* can be viewed as a way of eliminating implications:

$$\boxed{\Rightarrow\text{-}E}\ \frac{E_1 \Rightarrow E_2;\ E_1}{E_2}$$

The proof of *modus ponens* is on page 26. Notice how the final step of the construct 3 uses the contradiction rule.

In classical propositional calculus, it can be shown that if E_2 can be proved under the assumption E_1 (i.e. $E_1 \vdash E_2$), then $\vdash E_1 \Rightarrow E_2$ holds. This is called the 'deduction theorem'. Section 3.3 explains why – in order to handle partial functions – the logic used in this book does not admit all truths of classical logic. In the case of the deduction theorem, only a weaker form is valid which relies on the assumption of the 'excluded middle' for E_1:

$$E_1 \vee \neg E_1$$

This claim is written – with delta standing for 'defined' – as: $\delta(E_1)$.

Lemma 1.18 ($\Rightarrow$-I) The deduction theorem (here) is:

$$\boxed{\Rightarrow\text{-}I}\ \frac{E_1 \vdash E_2;\ \delta(E_1)}{E_1 \Rightarrow E_2}$$

As the name of the inference rule suggests, it can be used to introduce implications; its justification is shown on page 27. Line 2.1 is justified by showing the use of the inference rule which is given in the hypothesis.

There is (literally) no end of results which can be established.

from $E_1 \vdash E_2;\ \delta(E_1)$		
1	$E_1 \vee \neg E_1$	h, δ
2	**from** E_1	
2.1	$\quad E_2$	h, h2
	infer $\neg E_1 \vee E_2$	$\vee$-I(2.1)
3	**from** $\neg E_1$	
	infer $\neg E_1 \vee E_2$	$\vee$-I(h3)
4	$\neg E_1 \vee E_2$	$\vee$-E(1,2,3)
infer $E_1 \Rightarrow E_2$		$\Rightarrow$-*defn*(4)

Deduction theorem: $\Rightarrow$-I

Lemma 1.19 An interesting result is:

$$\boxed{\text{L1.19}}\ \frac{E_1 \vee E_2 \Rightarrow E_3}{(E_1 \Rightarrow E_3) \wedge (E_2 \Rightarrow E_3)}$$

Notice that, since no conveniently short name is available for this rule, it has *only* been given a Lemma number (L1.19) to which subsequent proofs can refer.

Lemma 1.20 Another result used below is:

$$\boxed{\text{L1.20}}\ \frac{E_1 \Rightarrow (E_2 \Rightarrow E_3)}{E_1 \wedge E_2 \Rightarrow E_3}$$

Defining equivalence

Rule 1.21 ($\Leftrightarrow$-*defn*) The final operator in the logic is also introduced by a definition:

$$\boxed{\Leftrightarrow\text{-}defn}\ \frac{(E_1 \Rightarrow E_2) \wedge (E_2 \Rightarrow E_1)}{E_1 \Leftrightarrow E_2}$$

An extensive set of derived rules[5] in given in Appendix C; they are arranged for easy use rather than in the order in which their proofs have been given. It is legitimate to use any of these rules in proofs of results in subsequent sections.

Exercise 1.3.8 The proofs that certain vacuous implications hold are straightforward; prove the results necessary to establish:

[5]The full axiomatization is given in the *Teacher's Notes*.

$$\boxed{\Rightarrow vac\text{-}I}\ \frac{\neg E_1}{E_1 \ \Rightarrow\ E_2}$$

$$\boxed{\Rightarrow vac\text{-}I}\ \frac{E_2}{E_1 \ \Rightarrow\ E_2}$$

Exercise 1.3.9 Prove the result necessary to establish that the contrapositive of an implication holds:

$$\boxed{\Rightarrow\text{-}contrp}\ \frac{E_1 \ \Rightarrow\ E_2}{\neg E_2 \ \Rightarrow\ \neg E_1}$$

Exercise 1.3.10 Prove Lemmas 1.19 and 1.20.

Exercise 1.3.11 Prove the results which justify:

$$\boxed{\Leftrightarrow\text{-}I}\ \frac{E_1 \wedge E_2}{E_1 \ \Leftrightarrow\ E_2}$$

$$\boxed{\Leftrightarrow\text{-}I}\ \frac{\neg E_1 \wedge \neg E_2}{E_1 \ \Leftrightarrow\ E_2}$$

$$\boxed{\Leftrightarrow\text{-}E}\ \frac{E_1 \ \Leftrightarrow\ E_2}{E_1 \wedge E_2 \vee \neg E_1 \wedge \neg E_2}$$

$$\boxed{\wedge\Leftrightarrow\text{-}dist}\ \frac{E_1 \wedge (E_2 \ \Leftrightarrow\ E_3)}{(E_1 \wedge E_2) \ \Leftrightarrow\ (E_1 \wedge E_3)}$$

Generate a counter-example (truth values) which shows that the following sequent does not hold:

$$(E_1 \wedge E_2) \ \Leftrightarrow\ (E_1 \wedge E_3) \vdash E_1 \wedge (E_2 \ \Leftrightarrow\ E_3)$$

Prove the result to justify:

$$\boxed{\vee\Leftrightarrow\text{-}dist}\ \frac{E_1 \vee E_2 \ \Leftrightarrow\ E_1 \vee E_3}{E_1 \vee (E_2 \ \Leftrightarrow\ E_3)}$$

Exercise 1.3.12 (*) Write inference rules for the ‘exclusive or’ operator of Exercise 1.1.5 on page 9 and develop a theory which includes some distribution properties.

2

Reasoning about Predicates

In science nothing capable of proof ought to be
accepted without proof.
Richard Dedekind

This chapter extends the logical notation of the preceding chapter to cover predicate calculus. It begins by introducing ways of building interesting logical expressions from truth-valued functions. Section 2.2 describes the essential extension (quantifiers) to the logical notation and the final section gives an overview of the relevant proof methods.

2.1 Truth-valued functions

Signatures

A *function* is a mathematical abstraction of a familiar concept: a mapping between two sets of values. The domain of a function is a set of values to which it can be applied; *application* of a function to a value in its domain yields a result value. For example $square(3) = 9$ and $gcd(18, 42) = 6$. The value 3 is in the domain of the function *square* and applying *square* to 3 yields the result 9; in such an application, 3 is also referred to as the *argument* of the function *square*. The function *gcd* (greatest common divisor or highest common factor) is applied to pairs of numbers.

For any function, it is useful to record its *domain* (i.e. the specified set of values to which the function can be applied) and *range* (i.e. the specified set of values which contains the results of function application). The *signature* of a function is written with the domain and range sets separated by an arrow:

$$square\colon\ \mathbb{Z} \to \mathbb{N}$$

The domain of a function of more than one argument[1] is given as a list all of the argument sets separated by crosses.

$$gcd\colon\ \mathbb{N}_1 \times \mathbb{N}_1 \to \mathbb{N}_1$$

Where the special symbols name the following (infinite) sets:

$$\mathbb{N}_1 = \{1, 2, \ldots\}$$
$$\mathbb{N} = \{0, 1, 2, \ldots\}$$
$$\mathbb{Z} = \{\ldots, -1, 0, 1, \ldots\}$$

Notice that the signature uses the names of the *sets* of values (e.g. the integers, $\mathbb{Z}$, for the domain of *square*; the natural numbers, $\mathbb{N}$, for its range); the values to which a function is applied are *elements* of the set shown as the domain and the results are *elements* of the set shown as the range.

Some functions are used so frequently that it is convenient to avoid parentheses when they are applied to their arguments. This is particularly appropriate if algebraic properties become more apparent by writing functions as *operators*. Thus $2 + 3$ is preferred to add(2,3) or even +(2,3) and the use of infix operators makes the distributive law:

$$i * (j + k) = i * j + i * k$$

clearer. The signature of such functions might be written:

$$add\colon\ \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$$

But they will be used in infix operators.

As well as the obvious arithmetic operators, the examples in this chapter use the modulus operator which yields the remainder after integer division:

$$7 \mathbf{~mod~} 2 = 1$$
$$27 \mathbf{~mod~} 3 = 0$$

Its signature is:

$$mod\colon\ \mathbb{N} \times \mathbb{N}_1 \to \mathbb{N}$$

The decision as to whether a particular operator should be presented in infix style ($i \mathbf{~mod~} j$) as opposed to writing it as a function with parentheses around

[1] Such functions can be viewed as taking one argument from a Cartesian product.

its arguments ($mod(i,j)$) is purely one of convenience; similarly, there is no deep significance in the adoption of some special symbol as opposed to a keyword (in the sans serif fount).

A *truth-valued function* is one whose range is the Boolean, or truth value, set. The function which characterizes the prime numbers has the signature:

$$is\text{-}prime\colon \mathbb{N}_1 \to \mathbb{B}$$

This truth-valued function is defined formally in Section 2.2.

An expression which contains the application of a truth-valued function to an element of its domain forms a proposition. Thus:

$$is\text{-}prime(7)$$
$$is\text{-}prime(23)$$
$$\neg\, is\text{-}prime(8)$$
$$is\text{-}prime(7) \vee is\text{-}prime(8) \vee is\text{-}prime(9)$$

are true propositions.

Defining functions

Functions can be defined in terms of already understood functions (or operators) and constants; in addition, the expressions in such direct definitions use parameter names in an obvious way. For example the signature and direct definition of *square* can be written:

$$square : \mathbb{Z} \to \mathbb{N}$$
$$square(i) \quad \triangleq \quad i * i$$

In order to distinguish the direct definition of a function from propositions which might involve equality (e.g. $square(2) = 4$), a Greek delta (Δ) is combined with the equality sign to give the definition symbol ($\triangleq$).

In addition to known functions, certain other constructs are available to form direct function definitions. For example, conditional expressions can be used in an obvious way to write:

$$abs : \mathbb{Z} \to \mathbb{N}$$
$$abs(i) \quad \triangleq \quad \textbf{if } i < 0 \textbf{ then } -i \textbf{ else } i$$

Another simple device is to use **let** to define a value. Thus the absolute value of the product of two integers could be found by:

$$\begin{array}{l} absprod : \mathbb{Z} \times \mathbb{Z} \to \mathbb{N} \\ absprod(i, j) \quad \triangleq \\ \qquad \textbf{let } k = i * j \textbf{ in} \\ \qquad \textbf{if } k < 0 \textbf{ then } -k \textbf{ else } k \end{array}$$

Extensions to the language for direct function definition (e.g. cases, recursion) are introduced below as they are required.

Such direct definitions can be written for truth-valued functions. Thus, if **mod** is understood, a truth-valued function (operator) which indicates whether its first argument divides its second without remainder, can be defined:

$$\begin{array}{l} divides : \mathbb{N}_1 \times \mathbb{N} \to \mathbb{B} \\ divides(i, j) \quad \triangleq \quad j \textbf{ mod } i = 0 \end{array}$$

But, since this is useful as an infix operator, $divides(i, j)$ is written i **divides** j.

Other examples include:

$$\begin{array}{l} is\text{-}even : \mathbb{N} \to \mathbb{B} \\ is\text{-}even(i) \quad \triangleq \quad 2 \textbf{ divides } i \end{array}$$

$$\begin{array}{l} is\text{-}odd : \mathbb{N} \to \mathbb{B} \\ is\text{-}odd(i) \quad \triangleq \quad \neg\, is\text{-}even(i) \end{array}$$

$$\begin{array}{l} is\text{-}common\text{-}divisor : \mathbb{N} \times \mathbb{N} \times \mathbb{N}_1 \to \mathbb{B} \\ is\text{-}common\text{-}divisor(i, j, d) \quad \triangleq \quad d \textbf{ divides } i \wedge d \textbf{ divides } j \end{array}$$

Notice how these definitions are built up on previously defined functions. The separation and naming of separate concepts plays an important part in the construction of large (understandable) specifications.

Values (in its domain) for which a truth-valued function yields **true**, are said to *satisfy* the function. Thus 7 satisfies *is-prime*, 6 satisfies *is-even*, and the triple of values (42,18,6) satisfies *is-common-divisor*.

One way in which a free identifier in a proposition becomes bound to a value is by the application of a function to some value. Thus:

$$\begin{array}{l} less\text{-}than\text{-}three : \mathbb{N} \to \mathbb{B} \\ less\text{-}than\text{-}three(i) \quad \triangleq \quad i < 3 \end{array}$$

is a definition of a truth-valued function whose application to 2 completes the proposition; it evaluates to **true** and thus 2 is said to satisfy *less-than-three*.

Exercise 2.1.1 Define a truth-valued function:

$$\begin{aligned}
&\textit{is-hexable} : \mathbb{Z} \to \mathbb{B}\\
&\textit{is-hexable}(i) \quad \triangleq \quad \cdots
\end{aligned}$$

which determines whether a number can be represented as a single hexadecimal digit.

Exercise 2.1.2 Define a truth-valued function which checks if its (integer) argument[2] corresponds to a leap year:

$$\begin{aligned}
&\textit{is-leapyr} : \mathbb{N} \to \mathbb{B}\\
&\textit{is-leapyr}(i) \quad \triangleq \quad \cdots
\end{aligned}$$

Exercise 2.1.3 Define a truth-valued function which determines whether its third argument is a common multiple of its other two arguments. (Hint: remember to use other functions in order to make it easier to understand.)

Exercise 2.1.4 It is often useful to employ an inverse operation to specify a function. This topic is covered in Chapter 3, but the reader should be able to see how a 'post-condition' can be used to relate the inputs to the outputs of a function. Thus:

$$\textit{post-sub}(i, j, k) \quad \triangleq \quad i = j + k$$

is a truth-valued function which can be used to check that $k = i - j$. Define (without using a square root operator) a truth-valued function:

$$\begin{aligned}
&\textit{post-sqrt} : \mathbb{N} \times \mathbb{Z} \to \mathbb{B}\\
&\textit{post-sqrt}(i, r) \quad \triangleq \quad \cdots
\end{aligned}$$

such that both $\textit{post-sqrt}(9, 3)$ and $\neg\, \textit{post-sqrt}(9, 4)$ are true (decide what to do about expressions like $\textit{post-sqrt}(9, -3)$ and $\textit{post-sqrt}(8, ?)$).

Exercise 2.1.5 Define a truth-valued function which determines whether a quotient q and remainder r represent a valid result for division of i by j. Complete (without using division):

$$\begin{aligned}
&\textit{post-idiv} : \mathbb{N} \times \mathbb{N}_1 \times \mathbb{N} \times \mathbb{N} \to \mathbb{B}\\
&\textit{post-idiv}(i, j, q, r) \quad \triangleq \quad \cdots
\end{aligned}$$

such that:

$$\begin{aligned}
&\textit{post-idiv}(7, 2, 3, 1)\\
&\neg\, \textit{post-idiv}(7, 2, 2, 3)
\end{aligned}$$

[2] Strictly, the simple algorithm should be limited so as to avoid, for example, difficulties of the revision of the calendar in September 1752: such issues are ignored here.

Exercise 2.1.6 (*) Some indication of the variety of ways in which inference rules can be used is given at the end of Section 1.2 by the alternative presentation of a concrete syntax. It is also possible to present the type information as inference rules rather than in function signatures. Create some inference rules for this purpose and show how they can be used to infer the types of expressions.

2.2 Quantifiers

The existential quantifier

The language for building logical expressions can be extended by including quantifiers. Their presentation in this section differs from the way in which the propositional operators are introduced in Section 1.1: there, a rich set of equivalences and a simple evaluation mechanism (i.e. truth tables) made it interesting to study the propositional operators with arbitrary logical expressions; here, the quantifiers are discussed with specific truth-valued functions and only a limited set of derived rules is developed for use in subsequent chapters.

Quantifiers extend the expressive power of the logical notation but can be motivated as abbreviations. The disjunction $\textit{is-prime}(7) \vee \textit{is-prime}(8) \vee \textit{is-prime}(9)$ can be written:

$$\exists i \in \{7, 8, 9\} \cdot \textit{is-prime}(i)$$

This quantified expression can be read as:

> there exists a value in the set $\{7, 8, 9\}$ which satisfies the truth-valued function *is-prime*

The expression consists of an *existential quantifier* ($\exists$); a *bound identifier* (i); a *constraint* ($\in \{\ldots\}$); and, after the raised dot, a *body*. Any free occurrences of the bound identifier within the body become bound in the quantified expression. All such occurrences refer to the bound identifier. Quantifiers thus provide another way of defining a context for free identifiers.

For finite sets, an existentially quantified expression can be expanded into a disjunction with one disjunct for each member of the set. This is a useful reminder to read $\exists$ as 'there exists one or more'. Thus:

$$\exists i \in \{11, 12, 13\} \cdot \textit{is-odd}(i)$$

is true because it is equivalent to:

$$\textit{is-odd}(11) \vee \textit{is-odd}(12) \vee \textit{is-odd}(13)$$

The reason that quantifiers extend the expressive power of the logic is that the sets in the constraint of a quantified expression can be infinite. Such an expression abbreviates a disjunction which could never be completely written out. For example:

$$\exists i \in \mathbb{N}_1 \cdot \textit{is-prime}(i)$$

or:

$$\exists i \in \mathbb{N}_1 \cdot \neg\, \textit{is-prime}(2^i - 1)$$

express facts about prime numbers.

One way of establishing the existence of a quantity with a certain property is by exhibiting one. Thus the truth of the preceding existentially quantified expressions follows from:

$$\textit{is-prime}(7)$$
$$\neg\, \textit{is-prime}(2^8 - 1)$$

To be consistent with the position about the verification of existentially quantified expressions, any expression which is existentially quantified over the empty set must be false. Thus, for any truth-valued function p:

$$\neg\, \exists x \in \{\,\} \cdot p(x)$$

Existentially quantified expressions can be used in definitions of truth-valued functions. Thus the familiar 'less than' relation on integers (normally written $i < j$) could be defined:

$$\textit{lessthan} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$$
$$\textit{lessthan}(i, j) \quad \triangleq \quad \exists k \in \mathbb{N}_1 \cdot i + k = j$$

The preceding section uses **mod** as a given function. Although further notation is needed to provide a definition, a useful property can be stated:

$$i \ \mathbf{mod}\ j = r \ \Rightarrow\ \exists m \in \mathbb{N} \cdot m * j + r = i$$

Many textbooks on logic do not use the constraint part of quantified expressions. This is acceptable where the whole text is concerned with one type of value. Program specifications are, however, frequently concerned with many different types of values and it is then wise to make the constraint explicit in order to avoid confusion (e.g. claiming that no value can be doubled to yield an odd number and then being confronted with 1.5).

Universal quantifiers

Just as some disjunctions can be viewed as existentially quantified expressions, a conjunction such as:

$$is\text{-}even(2) \wedge is\text{-}even(4) \wedge is\text{-}even(6)$$

can be written as a *universally quantified* expression:

$$\forall i \in \{2, 4, 6\} \cdot is\text{-}even(i)$$

Here again, the increase in expressive power comes from universal quantification over infinite sets. For example:

$$\forall i \in \mathbb{N} \cdot is\text{-}even(2 * i)$$
$$\forall i \in \mathbb{N} \cdot is\text{-}even(i) \;\Rightarrow\; is\text{-}odd(i + 1)$$
$$\forall i \in \mathbb{N} \cdot \forall j \in \mathbb{N}_1 \cdot 0 \leq (i \;\mathbf{mod}\; j) < j$$

The truth-valued function *is-prime* which is used above can be directly defined by using quantifiers. The general idea of a prime number is one whose only divisors are 1 and the number itself. This is easy to express but care is necessary with the end cases: both 1 and 2 have the stated property. Disallowing the former, but not the latter, leads to:

$$is\text{-}prime : \mathbb{N} \to \mathbb{B}$$
$$is\text{-}prime(i) \quad \triangleq \quad i \neq 1 \wedge \forall d \in \mathbb{N}_1 \cdot d \;\mathbf{divides}\; i \;\Rightarrow\; d = 1 \vee d = i$$

The question of universal quantification over the empty set must be considered. It is necessary to adopt the position that, for any p:

$$\forall x \in \{\,\} \cdot p(x)$$

is true. The intuition behind this is less obvious than with the existential quantifier – although one could argue that there could not be any counter-examples that could make $p(x)$ false in the empty set. One could also argue as follows – suppose:

$$\forall x \in X \cdot p(x)$$

were true for some X and p, then removing one element from X should not change the value of the quantified expression even when the last element is removed. More convincing than either of these general arguments is seeing how conveniently this end-case works in practice. For example, *is-prime* could be defined:

$$is\text{-}prime(i) \quad \triangleq \quad i \neq 1 \wedge \forall d \in \{2, \ldots, i - 1\} \cdot \neg\, (d \;\mathbf{divides}\; i)$$

Where:

$$\{2, \ldots, i-1\}$$

is the set of integers which are greater than one and less than i; in the case that i is one or two, this set is empty and the truth of the quantified expression over the empty set gives the required result.

Multiple quantifiers

Where they are all the same, multiple quantifiers and bound sets can be combined. Thus:

$$\forall i \in \mathbb{N} \cdot \forall j \in \mathbb{N} \cdot p(i, j)$$
$$\forall j \in \mathbb{N} \cdot \forall i \in \mathbb{N} \cdot p(i, j)$$
$$\forall i \in \mathbb{N}, j \in \mathbb{N} \cdot p(i, j)$$
$$\forall i, j \in \mathbb{N} \cdot p(i, j)$$

all have the same meaning. In fact, where a logical expression contains variables which are not bound, they are considered to be bound by a universal quantifier at the outermost level. Thus, where the types are obvious $i < i+1$ can be considered to be shorthand for $\forall i \in \mathbb{N} \cdot i < i+1$.

It is possible to build up expressions using both existential and universal quantifiers. For example:

$$\forall i, j \in \mathbb{N} \cdot i \leq j \;\Rightarrow\; \exists k \in \mathbb{N} \cdot i + k = j$$
$$\forall i \in \mathbb{N} \cdot \exists j \in \mathbb{N} \cdot i < j \wedge \textit{is-prime}(j)$$
$$\exists i, j \in \mathbb{N} \cdot \forall d \in \mathbb{N}_1 \cdot \textit{is-common-divisor}(i, j, d) \;\Rightarrow\; d = 1$$

all express true facts about natural numbers. It is important, however, to realize that inversion of differing quantifiers can change the truth of an expression. Consider:

$$\forall j \in \mathbb{N} \cdot \exists i \in \mathbb{N} \cdot i = j$$

This is clearly true, whereas:

$$\exists i \in \mathbb{N} \cdot \forall j \in \mathbb{N} \cdot i = j$$

is false. In general:

$$(\exists i \in \mathbb{N} \cdot \forall j \in \mathbb{N} \cdot p(i, j)) \;\Rightarrow\; (\forall j \in \mathbb{N} \cdot \exists i \in \mathbb{N} \cdot p(i, j))$$

is true but the right-to-left implication is not.

As with the priority of propositional operators, it is possible to reduce the need for parentheses by adopting some conventions. The body of a quantified expression is considered throughout this book to extend as far to the right as possible – thus:

$$\forall m, n \in \mathbb{N} \cdot (m = n \vee (\exists p \in \mathbb{Z} \cdot (p \neq 0 \wedge m + p = n)))$$

can be written:

$$\forall m, n \in \mathbb{N} \cdot m = n \vee \exists p \in \mathbb{Z} \cdot p \neq 0 \wedge m + p = n$$

The bound variables in a closed quantified expression are like the variables in a program in that they can be changed (systematically) without changing the meaning of the expression. Thus, the preceding expression is equivalent to:

$$\forall i, j \in \mathbb{N} \cdot i = j \vee \exists k \in \mathbb{Z} \cdot k \neq 0 \wedge i + k = j$$

When changing bound variables, it is necessary to ensure that the meaning is not changed by using an identifier which already occurs free.

Given that universal and existential quantification are (respectively) generalized conjunctions and disjunctions, the following forms of de Morgan's laws should come as no surprise:

$$\begin{array}{l} (\forall x \in X \cdot p(x)) \;\Leftrightarrow\; \neg\,(\exists x \in X \cdot \neg\, p(x)) \\ \neg\,(\forall x \in X \cdot p(x)) \;\Leftrightarrow\; (\exists x \in X \cdot \neg\, p(x)) \end{array}$$

These laws permit some simple equivalence proofs to be conducted:

$$\begin{array}{l} (\forall i \in \mathbb{N}_1 \cdot \exists j \in \mathbb{N}_1 \cdot i < j \wedge \textit{is-prime}(j)) \\ \quad \Leftrightarrow\; \neg\,(\exists i \in \mathbb{N}_1 \cdot \neg\,(\exists j \in \mathbb{N}_1 \cdot i < j \wedge \textit{is-prime}(j))) \\ \quad \Leftrightarrow\; \neg\,(\exists i \in \mathbb{N}_1 \cdot \forall j \in \mathbb{N}_1 \cdot \neg\,(i < j \wedge \textit{is-prime}(j))) \\ \quad \Leftrightarrow\; \neg\,(\exists i \in \mathbb{N}_1 \cdot \forall j \in \mathbb{N}_1 \cdot j \leq i \vee \neg\, \textit{is-prime}(j)) \\ \quad \Leftrightarrow\; \neg\,(\exists i \in \mathbb{N}_1 \cdot \forall j \in \mathbb{N}_1 \cdot \textit{is-prime}(j) \;\Rightarrow\; j \leq i) \end{array}$$

Having accepted that $\exists$ corresponds to 'there exists one or more', there are occasions where it is useful to be able to express 'there exists exactly one'. This is written as $\exists!$. For example:

$$\begin{array}{l} \forall i, j \in \mathbb{N}_1 \cdot \\ \qquad \textit{is-prime}(i) \wedge \textit{is-prime}(j) \wedge i \neq j \\ \qquad \Rightarrow\; \exists!\, d \in \mathbb{N}_1 \cdot \textit{is-common-divisor}(i, j, d) \end{array}$$

This quantifier:

$$\exists!\, x \in X \cdot p(x)$$

can be defined as an abbreviation for:

$$\exists x \in X \cdot p(x) \wedge \forall y \in X \cdot p(y) \;\Rightarrow\; x = y$$

All of the laws of the propositional calculus (cf. Section 1.3) remain true when general logical expressions (i.e. including quantified expressions) are substituted for the E_i. The language which is now available (propositional operators, truth-valued functions and quantified expressions) is known as the *predicate calculus*.[3]

Exercise 2.2.1 Which of the following expressions are true?

$$\exists i \in \mathbb{N} \cdot i = i$$
$$\forall i \in \mathbb{N} \cdot i = i$$
$$\exists i \in \mathbb{N} \cdot i \neq i$$
$$\exists i, j \in \mathbb{N}_1 \cdot i \ \mathbf{mod}\ j \geq j$$

$$\forall i \in \mathbb{Z} \cdot \exists j \in \mathbb{Z} \cdot i + j = 0$$
$$\exists j \in \mathbb{Z} \cdot \forall i \in \mathbb{Z} \cdot i + j = 0$$

$$\forall i, j \in \mathbb{N} \cdot i \neq j$$
$$\forall i \in \mathbb{N} \cdot \exists j \in \mathbb{N} \cdot j = i - 1$$
$$\forall i \in \mathbb{N} \cdot \exists j \in \mathbb{N} \cdot i < j < 2 * i \wedge \textit{is-odd}(j)$$
$$\forall i \in \mathbb{N}_1 \cdot \neg\, \textit{is-prime}(4 * i)$$
$$\forall i \in \mathbb{N} \cdot \exists j \in \mathbb{N} \cdot j \leq 3 \wedge \textit{is-leapyr}(i + j)$$

$$\exists!\, i \in \mathbb{N} \cdot i = i$$
$$\exists!\, i \in \mathbb{Z} \cdot i * i = i$$

Exercise 2.2.2 Express, using quantifiers, the fact that there is not a largest integer.

Exercise 2.2.3 Define – with quantifiers (but without using ordering operators) – a truth-valued function corresponding to $(i \geq j)$:

$$\textit{greatereq} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$$
$$\textit{greatereq}(i, j) \quad \triangleq \quad \cdots$$

Exercise 2.2.4 The function, *sign*, yields a value in the set:

$$\{-1, 0, 1\}$$

[3]Strictly, in this book, only the first-order predicate calculus is used. This means that variables are only quantified over simple values like natural numbers – names of truth-valued functions are not quantified. It is observed above that the truth of sentences in the propositional calculus is decidable (cf. checking by truth tables). Although it is less obvious, there are semi-decision procedures for the pure predicate calculus; the truth of sentences in the predicate calculus with interpreted functions and equality is, however, not decidable.

depending on whether its argument is negative, zero, or strictly positive. Write a definition and record some properties of *sign*.

Exercise 2.2.5 (*) An extended modulus operator can be applied to negative (as well as positive) numbers. There are various forms of this operator. Mathematically, it is convenient to ensure that:

$$m \textbf{ mod } n + (m \div n) * n = m$$

where $\div$ is an integer division operator. Define this operator.

2.3 Proofs in the predicate calculus

The development of the proof rules for the predicate calculus can be based on one of the quantifiers and the notion of equality. In this respect the way in which the theory is presented is very similar to that of Section 1.3. There are, however, some technical problems with free variables and their substitution which make the development of the derived rules somewhat more difficult than for the propositional calculus. Relatively few rules about quantifiers are needed in the chapters which follow. This section develops those which are required; a wider-ranging set of rules is given in Appendix C.

Substitution

A preliminary to the presentation of any rules is the establishment of some conventions on the use of letters. Letters at the end of the alphabet (x, etc.) are used for variables. The convention to use E_i for logical expressions is maintained, but is extended to show specific free variables; thus $E(x)$ has the variable x occurring free. It is explained above that terms are expressions (such as $2 + 3$); the letter s – possibly subscripted – is used to denote terms.

An essential notion is that of syntactic substitution. The expression $E(s/x)$ is formed by substituting all free occurrences of the variable x by the term s. Thus 7 can be substituted for x as follows:

$$(x = 3 + 4)(7/x) \;=\; (7 = 3 + 4)$$

But the restriction that this syntactic operation only affects free variables ensures that:

$$(\forall x \in X \cdot x = x)(7/x) \;=\; (\forall x \in X \cdot x = x)$$

There is a more ticklish problem with substitution concerning the *capture* of a variable. In making the substitution:

$$(y = 10 \vee \forall x \in \mathbb{N} \cdot x \neq 10 \;\Rightarrow\; x \neq y)(x/y)$$

the change from y to x should not cause a confusion between the free and bound variables. In such a case, it is sufficient to remember that bound variables can be systematically changed so that:

$$\forall x \in \mathbb{N} \cdot x \neq 10 \;\Rightarrow\; x \neq y$$

and:

$$\forall i \in \mathbb{N} \cdot i \neq 10 \;\Rightarrow\; i \neq y$$

are equivalent. In a case where a free variable would be captured by a substitution, the danger is avoided by preceding the substitution with a suitable systematic change to the bound variable in question. Thus, the substitution above might yield:

$$x = 10 \vee \forall i \in \mathbb{N} \cdot i \neq 10 \;\Rightarrow\; i \neq x$$

Although these technicalities of substitution are important, the need to rely on them can be minimized by a careful choice of variables.

Reasoning about quantifiers

It would be possible to take either the existential or the universal quantifier as basic and define the other in terms of the basic one. Having used the disjunction as one of the basic forms for the presentation of the propositional calculus, it is natural to take the existential quantifier first. An obvious example of the rule for the introduction of this quantifier is:

$$7 \in \mathbb{N}_1, \textit{is-prime}(7) \;\vdash\; \exists i \in \mathbb{N}_1 \cdot \textit{is-prime}(i)$$

This states that knowing the type of the term (7) and knowing that it possesses a particular property (*is-prime*) establishes that there exist (one or more) values of the type which satisfy the property.

Axiom 2.1 ($\exists$-I) In general:

$$\boxed{\exists\text{-}I}\;\frac{s \in X;\; E(s/x)}{\exists x \in X \cdot E(x)}$$

If the reader compares this rule with that for $\vee$-I, it can be seen as a natural generalization. The conclusion of the proof rule is essentially a disjunction over all of the elements of X; the second hypothesis establishes that the property does hold for some element of X. (Notice that the first hypothesis establishes that the set X is not empty.)

The form of the $\vee$-E rule shows that a conclusion which follows from a number of expressions also follows from their disjunction. The general idea of the $\exists$-E rule is the same. The need to show that E follows from each possible value in X is avoided by a subtle use of free variables.

Axiom 2.2 ($\exists$-E) Thus:

$$\boxed{\exists\text{-}E}\ \frac{\exists x \in X \cdot E(x);\ y \in X, E(y/x) \ \vdash\ E_1}{E_1}\ y \text{ is arbitrary}$$

The restriction that y is *arbitrary* requires that it is a variable which has not occurred in earlier proof steps; it should also be the case that y does not occur as a free variable in E_1. These restrictions prevent invalid inferences.

A comparison with the development of the propositional calculus should again suggest the need for rules concerning negated forms of existentially quantified expressions.

Axiom 2.3 For the existential quantifier, these rules are:

$$\boxed{\neg\,\exists\text{-}I}\ \frac{x \in X \ \vdash\ \neg\, E(x)}{\neg\,(\exists x \in X \cdot E(x))}$$

$$\boxed{\neg\,\exists\text{-}E}\ \frac{\neg\,(\exists x \in X \cdot E(x));\ s \in X}{\neg\, E(s/x)}$$

The relationship between these rules and those for disjunction should be obvious when existential quantification (over finite sets) is viewed as an abbreviation for disjunction.

Defining universal quantification

Rule 2.4 ($\forall$-*defn*) The definition of the universal quantifier is given by the rule:

$$\boxed{\forall\text{-}defn}\ \frac{\neg\,(\exists x \in X \cdot \neg\, E(x))}{\forall x \in X \cdot E(x)}$$

This gives rise to the two generalized forms of de Morgan's law.

Lemma 2.5 ($\exists$-*deM*) For existential quantifiers:

$$\boxed{\exists\text{-}deM}\ \frac{\neg\,(\exists x \in X \cdot E(x))}{\forall x \in X \cdot \neg\, E(x)}$$

Lemma 2.6 ($\forall$-*deM*) For universal quantifiers:

from $x \in X \vdash p(x)$		
1	**from** $x \in X$	
1.1	$p(x)$	h,h1
	infer $\neg\neg p(x)$	$\neg\neg$-I(1.1)
2	$\neg \exists x \in X \cdot \neg p(x)$	$\neg\exists$-I(1)
infer $\forall x \in X \cdot p(x)$		$\forall$-*defn*(2)

Universal quantifier introduction: $\forall$-I

from $\forall x \in X \cdot p(x); s \in X$		
1	$\neg \exists x \in X \cdot \neg p(x)$	$\forall$-*defn*(h)
2	$\neg\neg p(s/x)$	$\neg\exists$-E(h,1)
infer $p(s/x)$		$\neg\neg$-E(2)

Universal quantifier elimination: $\forall$-E

$$\boxed{\forall\text{-}deM}\ \frac{\neg\,(\forall x \in X \cdot E(x))}{\exists x \in X \cdot \neg\, E(x)}$$

These can be proved as derived rules.

Given the basic definition, it is possible to derive the introduction and elimination rules for the universal quantifier.

Lemma 2.7 ($\forall$-I)

$$\boxed{\forall\text{-}I}\ \frac{x \in X \vdash E(x)}{\forall x \in X \cdot E(x)}$$

is proved on page 43.

Lemma 2.8 ($\forall$-E)

$$\boxed{\forall\text{-}E}\ \frac{\forall x \in X \cdot E(x);\ s \in X}{E(s/x)}$$

is justified on page 43.

All of the rules for universal quantifiers are natural generalizations of the corresponding rules for conjunctions.

Lemma 2.9 presents some significant derived rules concerning the distribution of the two quantifiers over conjunctions and disjunctions.

Lemma 2.9

$$\boxed{\exists \vee\text{-}dist}\;\begin{array}{c} \exists x \in X \cdot E_1(x) \vee E_2(x) \\ \hline\hline (\exists x \in X \cdot E_1(x)) \vee (\exists x \in X \cdot E_2(x)) \end{array}$$

$$\boxed{\exists \wedge\text{-}dist}\;\begin{array}{c} \exists x \in X \cdot E_1(x) \wedge E_2(x) \\ \hline (\exists x \in X \cdot E_1(x)) \wedge (\exists x \in X \cdot E_2(x)) \end{array}$$

$$\boxed{\forall \vee\text{-}dist}\;\begin{array}{c} (\forall x \in X \cdot E_1(x)) \vee (\forall x \in X \cdot E_2(x)) \\ \hline \forall x \in X \cdot E_1(x) \vee E_2(x) \end{array}$$

$$\boxed{\forall \wedge\text{-}dist}\;\begin{array}{c} (\forall x \in X \cdot E_1(x)) \wedge (\forall x \in X \cdot E_2(x)) \\ \hline\hline \forall x \in X \cdot E_1(x) \wedge E_2(x) \end{array}$$

The reader should understand why the converses of $\exists \wedge$-*dist* and $\forall\vee$-*dist* do not hold. Other useful derived rules are presented in Appendix C; The full axiomatization is given in the *Teacher's Notes*.

There are fewer exercises here than in Chapter 1 since subsequent sections provide ample examples of the use of the rules, and the development of the predicate calculus itself is not the immediate object.

Exercise 2.3.1

Prove the derived rules for $\neg\forall$-I and $\neg\forall$-E:

$$\boxed{\neg\forall\text{-}I}\;\begin{array}{c} s \in X;\; \neg\, E(s/x) \\ \hline \neg\,(\forall x \in X \cdot E(x)) \end{array}$$

$$\boxed{\neg\forall\text{-}E}\;\begin{array}{c} \neg\,(\forall x \in X \cdot E_1(x));\;\; y \in X, \neg\, E_1(y/x) \;\vdash\; E_2 \\ \hline E_2 \end{array}\; y \text{ is arbitrary}$$

Exercise 2.3.2 (*) Prove the derived rules for $\exists \vee$-*dist*, $\exists \wedge$-*dist*, $\forall\vee$-*dist* and $\forall\wedge$-*dist* (remember to prove both forms of bi-directional rules).

3

Functions and Operations

> The advantages of implicit definition over construction are roughly those of theft over honest toil.
> B. Russell

Several functions are defined above in a direct way. In this chapter, an implicit specification style for functions and programs is introduced. Such implicit specifications abstract away from the detail of how a result is to be computed: they document only the required properties of the result. Being a standard part of mathematics, it is functions which are treated in the first section. This mathematically familiar area provides the opportunity to illustrate the fact that implicit specifications can often be more concise than implementations (i.e. direct definitions). The second section of this chapter is concerned with proofs that direct definitions satisfy implicit specifications. In order to keep the reliance on new concepts to a minimum, most of the examples are concerned with natural numbers and their operators. Later chapters of the book extend the range of data types. Section 3.3 reviews the reasons why the logical system which is used in this book is weaker than classical logic as presented in most textbooks: the system for dealing with partial functions is explained in detail. (Some readers may choose to read only the first part of this somewhat technical section.) The final section of this chapter extends the implicit specification notation to handle programs.

3.1 Implicit specification of functions

Reasons for being implicit

An *implicit specification* states *what* is to be computed whereas the direct definitions in previous chapters show *how* a result can be computed. (In programming terms, the direct definition is being treated as an implementation which has to be shown to satisfy the implicit specification.) There are several reasons for wanting to record an implicit specification. Perhaps the most obvious reason is that the specification is often significantly shorter than a direct definition. For example, a direct definition of, say, the Newton-Raphson approximation algorithm is much longer and harder to understand than stating that the result of a square root function should be such that when squared it differs from the argument by at most some tolerance.

It must, however, be conceded that such convenient algebraic properties do not always exist. Because of the way in which UK income tax is calculated, for example, even the specification of a function which determines tax deductions is very algorithmic. However, implicit specifications are often significantly more concise than implementations, and thinking in terms of specifications leads one to capitalize on this conciseness whenever possible. For significant problems there is a spectrum of specifications ranging from the very abstract to something which essentially describes the implementation. The full range of this spectrum becomes clear when data objects are discussed in subsequent chapters. One advantage of finding a specification far from the algorithmic end of the spectrum is that it may expose a range of alternative implementations.

An implicit specification is a way of recording a functional requirement without commitment to a particular method of calculation. Another attribute of such a specification is that it can state the properties of the required result in a way which is understandable to the user. For example, the user of the square root function can be expected to be interested in the property of the final answer and to wish to leave the details of the chosen implementation to the developer. There is, however, an attendant danger in implicit specification. Taking the same example again, the property as stated above would allow for either the negative or the positive root to be generated. This may be what is required; if not, it is easy to see how additional properties can be stated. An implicit specification must be such that *all* of the properties on which users wish to rely are consequences of the specification: the user should rely only on the specification (and its consequences).

There are two remaining arguments for recording implicit specifications. The points are more subtle but, in practical applications, very important. Whereas any particular algorithm will yield a specific result, a specification can state a range of acceptable results. Square root (over real numbers with a tolerance) provides a good example. Furthermore, implicit specifications provide an explicit place for recording assumptions about arguments. Many computer programs are designed under assumptions on their inputs and operating environment. The square root example could again be pressed into service by saying that its argument must be a positive real number (if the result is also to be a real number); but more interesting examples below illustrate how assumptions often concern the relationships between values. A specification can provide a way of making explicit those assumptions (pre-conditions) which are otherwise hidden consequences of an algorithm.

Format of implicit function specification

A function which is to yield the maximum number from the set of numbers to which it is applied gives:

$$maxs(\{3, 7, 1\}) = 7$$

Its specification might be written:

$$\begin{array}{l} maxs\ (s{:}\,\mathbb{N}\textbf{-set})\ r{:}\,\mathbb{N} \\ \textbf{pre}\ s \neq \{\,\} \\ \textbf{post}\ r \in s \wedge \forall i \in s \cdot i \leq r \end{array}$$

The first line of this specification defines the signature of the function. The syntax here is slightly different from that of Chapter 2: in implicit specifications the style is intentionally closer to that of programming languages like Pascal. Names are given to both arguments and results. The names are followed by their types. Thus *maxs* takes a finite set of natural numbers as its argument (Chapter 4 presents set notation in detail) and yields a single natural number as result. The names given are the link to the pre- and post-conditions: the identifiers used within these two truth-valued functions refer to the values of the objects which are named in the first line.

The *pre-condition* of a function records assumptions about the arguments to which it is to be applied. For this example, its type is:

$$pre\text{-}maxs{:}\,\mathbb{N}\textbf{-set} \to \mathbb{B}$$

Notice how the keyword **pre** is used in the specification but that a name is formed (in an obvious way) for the pre-condition if it is to be used out of context.

The pre-condition shows that *maxs* is a *partial function* which is required to be defined only when it is applied to non-empty sets. The post-condition requires that the result must be a member of the argument set and that no number in that set exceeds the result. The type of the post-condition for this example is:

$$post\text{-}maxs: \mathbb{N}\textbf{-set} \times \mathbb{N} \to \mathbb{B}$$

Notice that:

$$pre\text{-}maxs(\{3,7,1\}) \;\Leftrightarrow\; \textbf{true}$$
$$post\text{-}maxs(\{3,7,1\},7) \;\Leftrightarrow\; \textbf{true}$$

Writing a direct definition of *maxs* would pose a number of problems which are worth considering so as better to appreciate the implicit specification. The first problem would be one of naming. To write:

$$maxs(s) \;\triangleq\; \cdots$$

means that no name is available for the result[1] so that multiple properties cannot be written. One indirect way to provide such a name is to write:

$$maxs(s) = r \;\;\Rightarrow\;\; r \in s \wedge \forall i \in s \cdot i \leq r$$

This is, in fact, an indication of the origin of post-conditions. A direct definition can be given by using recursion:

$$\begin{array}{l} maxs(s) \;\triangleq \\ \quad \textbf{let } i \in s \textbf{ in} \\ \quad \textbf{if card } s = 1 \\ \quad \textbf{then } i \\ \quad \textbf{else } max(i, maxs(s - \{i\})) \end{array}$$

But this introduces a number of problems: leaving aside the *max* function for the moment, the arbitrary choice[2] implied in the **let** is unusual; the algorithm shown does not expose the essential properties of the result clearly; the fact that the function is partial is now a hidden property. Clearly, this is an example where the implicit specification is a useful description of the intended function.

The function which yields the larger of two integers might be specified:

$$\begin{array}{l} max\;(i: \mathbb{Z}, j: \mathbb{Z})\; r: \mathbb{Z} \\ \textbf{pre true} \\ \textbf{post } (r = i \vee r = j) \wedge i \leq r \wedge j \leq r \end{array}$$

Here there are no assumptions on the arguments (other than their type) and the pre-condition is **true**. As with *maxs*, the post-condition lists several conjoined

[1] One could use the iota (ι) description binding.

[2] The subject of under-determined functions is discussed at the end of Section 3.2.

$$
\begin{array}{l}
f\ (p{:}\ Tp)\ r{:}\ Tr \\
\textbf{pre}\ \cdots p \cdots \\
\textbf{post}\ \cdots p \cdots r \cdots
\end{array}
$$

Figure 3.1 Format of function specification

properties. In this simple example, the direct definition:

$$max(i,j) \quad \triangleq \quad \textbf{if } i \leq j \textbf{ then } j \textbf{ else } i$$

is no longer (nor more opaque) than the specification. In general, however, decomposing a post-condition into separate (conjoined) expressions results in a very clear specification which presents the required properties of the results in a way a user will appreciate. For example, it is convenient, in specifying a sort routine, to separate the properties of the order of the result and the necessity for the result to be a permutation of the starting sequence.

Figure 3.1 gives the general format of an implicit function specification as used in this book. The truth-valued pre-condition can refer only to the values of the parameters; the post-condition normally refers to the values of both parameters and result. The names are bound by the variable names in the signature of the specification. (Because of the need to refer to arguments, as well as results, the name 'post-condition' is not ideal, but convention has established the term and it is used throughout this book.)

Meaning of implicit specification

Informally, such a specification requires that, to be correct with respect to the specification, a function must – when applied to arguments (of the right type) which satisfy the pre-condition – yield a result (of the right type) which satisfies the post-condition. This statement, and the requisite proof style, are presented formally in the next section. Notice that, for values which do *not* satisfy the pre-condition, nothing can be assumed about the result.

It is possible to write contradictory specifications which cannot be satisfied. Informally, it is clear that a specification must avoid this error; the notion of *satisfiability* is used below to formalize this.

In order to specify a function which can yield any element of a set, it is necessary only to remove one of the conjuncts of the post-condition of *maxs* – thus:

$arbs\ (s{:}\,\mathbb{N}\text{-}\mathbf{set})\ r{:}\,\mathbb{N}$
pre $s \neq \{\,\}$
post $r \in s$

Just as with the earlier discussion of square root, any algorithm would determine a particular result; the implicit specification indicates the permitted range of results.

Implicit specifications can – as has been seen above – also use quantifiers: this often avoids the use of recursion required in a direct definition. For example:

$gcd\ (i{:}\,\mathbb{N}_1, j{:}\,\mathbb{N}_1)\ r{:}\,\mathbb{N}_1$
pre true
post $is\text{-}common\text{-}divisor(i, j, r) \wedge$
$\qquad \neg\, \exists s \in \mathbb{N}_1 \cdot is\text{-}common\text{-}divisor(i, j, s) \wedge s > r$

Thus the advantages of implicit specification over direct definition include:

- direct description of (multiple) properties which are of interest to the user;
- characterizing a set of possible results by a post-condition;
- explicit record (by Boolean expression) of the pre-condition;
- less commitment to a specific algorithm;
- provision of a name for the result.

Where none of these points apply, a direct definition can be written. Indeed, since pre- and post-conditions are themselves (truth-valued) functions, it is clear that one must resort somewhere to direct definition or face an infinite regress.

Exercise 3.1.1 Write an implicit specification of a function which yields the minimum value from a set of integers.

Exercise 3.1.2 Write an implicit specification of a function which performs integer subtraction. Just as one teaches children, base the post-condition on the idea of 'the number which would have to be added to j to get i'.

Exercise 3.1.3 What change would be made to the preceding specification to ensure that neither zero nor negative numbers occur as arguments or results.

Exercise 3.1.4 Write an implicit specification of a function which yields the absolute value of an integer. Do not use a conditional expression in the post-condition.

Exercise 3.1.5 Specify a function which yields the smallest common multiple of two natural numbers. Build up useful subsidiary functions to make the specification readable.

Exercise 3.1.6 Specify the **mod** function (over positive integers).

Exercise 3.1.7 Proofs that direct definitions satisfy implicit specifications are considered below. For now check the following implementations against your specification and give a counter-example (test case value) if they are wrong.
For integer subtraction:

$$sub(i, j) \quad \triangleq \quad 2 * i/j$$

For natural number subtraction (cf. Exercise 3.1.3):

$$subp(i, j) \quad \triangleq \quad \textbf{if } i = j \textbf{ then } 0 \textbf{ else } 1 + subp(i, j + 1)$$

Would this be correct for the earlier case (Exercise 3.1.2)?
For absolute value (cf. Exercise 3.1.4):

$$abs(i) \quad \triangleq \quad max(i, -i)$$

For smallest common multiple (cf. Exercise 3.1.5):

$$scm(i, j) \quad \triangleq \quad i * j$$

3.2 Correctness proofs

Satisfaction notion

A direct definition of a function is said to *satisfy* an implicit specification if, for all arguments of the required type which satisfy the pre-condition, the evaluation of the direct definition yields results which are of the required type and satisfy the post-condition. This can be stated formally as a *proof obligation* which must be discharged in order to justify the claim of satisfaction. For the implicit specification given in Figure 3.1 on page 49, the pre- and post-conditions are truth-valued functions with the following names and signatures:

$$pre\text{-}f\colon Tp \to \mathbb{B}$$
$$post\text{-}f\colon Tp \times Tr \to \mathbb{B}$$

The keyword form in implicit function specifications provides a shorthand way of presenting these truth-valued functions. When used in logical expressions, they are handled just as if they had been given direct definitions:

$$pre\text{-}f(p) \triangleq \cdots p \cdots$$
$$post\text{-}f(p, r) \triangleq \cdots p \cdots r \cdots$$

Proof obligation 3.1 A direct definition:

$$f : Tp \to Tr$$
$$f(\cdots) \quad \triangleq \quad \cdots$$

satisfies the specification if (and only if):

$$\forall p \in Tp \cdot pre\text{-}f(p) \;\Rightarrow\; f(p) \in Tr \wedge post\text{-}f(p, f(p))$$

Thus the recursive function for *maxs* in the preceding section satisfies its specification; the same function also satisfies the specification of *arbs*. The concern in this section is with the construction of formal proofs of such statements.

Notice that the oft-used phrase '... is correct' should really be interpreted as '... satisfies the ... specification'. Without reference to a specification, the notion of correctness has no meaning.

The proof obligation for satisfaction, which is given above, makes the role of the pre-condition explicit: for argument values which do not satisfy the pre-condition, no constraint is placed on the implementation by the specification. Thus the overall specification is satisfied by an implementation regardless of the results which it produces for those arguments which fail to satisfy the pre-condition; the requirement is that the results are acceptable for those arguments which do satisfy the pre-condition. Similarly, a direct definition, which produces – for each argument – any answer which lies in the range of answers defined by the post-condition, satisfies the specification. Perhaps the most surprising consequence of the proof obligation is that the direct definition is allowed to produce no result (i.e. to be undefined) on arguments which do not satisfy the pre-condition.

The first examples of formal proofs are very simple in order to exhibit the general form of proof. Using $\mathbb{R}$ for the set of real numbers, a constant function *pi* can be specified:

$pi\ (y{:}\,\mathbb{R})\ r{:}\,\mathbb{R}$
pre true
post $abs(\pi - r) \leq 10^{-2}$

The argument y happens to play no part in the post-condition and could have been omitted since the function yields the same result for any argument; it is included only to make the form of the proof obligation easier to relate to the general case. A (rather crude) direct definition might be:

$$pi(y) \quad \triangleq \quad 3.141$$

from $x \in \mathbb{R}$		
1	$pi(x) = 3.141$	R3.2(h)
2	$3.141 \in \mathbb{R}$	$\mathbb{R}$
3	$pi(x) \in \mathbb{R}$	$=$-*subs*(2,1)
4	$abs(\pi - 3.141) \leq 10^{-2}$	$\mathbb{R}$
5	$abs(\pi - pi(x)) \leq 10^{-2}$	$=$-*subs*(4,1)
6	$post\text{-}pi(x, pi(x))$	*post-pi*(h,3,5)
7	$pi(x) \in \mathbb{R} \wedge post\text{-}pi(x, pi(x))$	$\wedge$-*I*(3,6)
infer	$pre\text{-}pi(x) \;\Rightarrow\; pi(x) \in \mathbb{R} \wedge post\text{-}pi(x, pi(x))$	$\Rightarrow$ *vac-I*(7)

Theorem 3.3: *pi* (sequent form)

In order to reason about such direct definitions it is necessary to use them in proofs. The smoothest transition from function definitions to proofs is to provide inference rule presentations for any direct definitions.

Rule 3.2 The rule for *pi* is:

$$\boxed{\text{R3.2}}\ \frac{y \in \mathbb{R}}{pi(y) = 3.141}$$

Theorem 3.3 The appropriate instance of proof obligation 3.1 is:

$$\forall x \in \mathbb{R} \cdot pre\text{-}pi(x) \;\Rightarrow\; pi(x) \in \mathbb{R} \wedge post\text{-}pi(x, pi(x))$$

The consequent of Theorem 3.3 has a universal quantifier; an obvious strategy is to prove the validity of the following sequent:

$$x \in \mathbb{R} \vdash pre\text{-}pi(x) \;\Rightarrow\; pi(x) \in \mathbb{R} \wedge post\text{-}pi(x, pi(x))$$

then the actual result follows from the $\forall$-I rule:

from $x \in \mathbb{R} \vdash pre\text{-}pi(x) \;\Rightarrow\; pi(x) \in \mathbb{R} \wedge post\text{-}pi(x, pi(x))$	
infer $\forall x \in \mathbb{R} \cdot pre\text{-}pi(x) \;\Rightarrow\; pi(x) \in \mathbb{R} \wedge post\text{-}pi(x, pi(x))$	$\forall$-I(h)

Theorem 3.3: *pi*

Here, the sequent form of the result is proved separately. Clearly, the universal quantifier could be introduced in the same proof but this would result in a deeper nesting than is necessary. In the proofs below, only the sequent form is proved because the quantified form would always follow in the same way.

The use of different identifiers in the specification and the proof obligation is deliberate: in the first two examples it is done to clarify the substitutions being performed. The proof of the sequent form is shown on page 53. In this proof of the sequent form of Theorem 3.3, appeals to the definitions of the **post** clause of the implicit specification are shown as *post-pi*, etc.: they are essentially unfoldings of those definitions with specific arguments. Thus, line 5 is an exact expansion of line 6; but notice that the justification of line 6 also refers to the hypothesis and to line 3 in order to establish that the arguments are of appropriate type. The other new form of justification which is used here is the $=$-*subs* used to justify lines 3 and 5. This substitution of a term – by one known to be equal to it – is intuitively simple. Notice, however, that if there is more than one occurrence it is *not* necessary to make all possible substitutions.

Axiom 3.4 ($=$-*subs*) So:

$$\boxed{=\text{-}subs}\ \frac{s = s';\ E_1}{E_2}$$

where the expression E_2 is obtained from E_1 by substituting *one or more* occurrences of s by s'.

Analyzing a proof

The general form of all of the proofs in this section is to make heavy use of the definitions and facts about the data types being manipulated with relatively little use of complex logical properties. Before seeking to understand how this proof could be discovered, the validity of the individual steps should be understood. Line 1 introduces to the proof on page 53 the knowledge about the function *pi* in the planned way by using R3.2 (the appeal to the hypothesis reflects the fact that the rule only applies to arguments of the correct type); lines 2 and 4 simply introduce facts about the underlying data type (real numbers); line 3 follows by substituting the left-hand side of the equality in line 1 for its right-hand side where the latter occurs in line 2; lines 4 and 5 are constructed in a similar way to lines 2–3; line 6 uses the definition of *post-pi* implied by the post-condition of the specification of *pi* (remember that the appeals to the hypothesis and line 3 are to establish types); line 7 and the final conclusion use derived inference rules of propositional calculus.

But how was this proof found? It is possible to construct the natural deduction proof fairly mechanically. Begin by writing the chosen sequent as:

$$
\begin{array}{lll}
\textbf{from } x \in \mathbb{R} & & \\
\quad\vdots & & \\
\textbf{infer } \mathit{pre\text{-}pi}(x) \;\Rightarrow\; \mathit{pi}(x) \in \mathbb{R} \wedge \mathit{post\text{-}pi}(x, \mathit{pi}(x)) & & ?
\end{array}
$$

Theorem 3.3 (first step)

Considering the goal, there are a collection of inference rules which could create an implication. The obvious rule would be $\Rightarrow$-I, but, noticing the special case that *pre-pi*(x) is true, prompts the selection of $\Rightarrow$ *vac-I* to create:

$$
\begin{array}{lll}
\textbf{from } x \in \mathbb{R} & & \\
\quad\vdots & & \\
\mathrm{k} & \mathit{pi}(x) \in \mathbb{R} \wedge \mathit{post\text{-}pi}(x, \mathit{pi}(x)) & ? \\
\textbf{infer} & \mathit{pre\text{-}pi}(x) \;\Rightarrow\; \mathit{pi}(x) \in \mathbb{R} \wedge \mathit{post\text{-}pi}(x, \mathit{pi}(x)) & \Rightarrow \mathit{vac\text{-}I}(\mathrm{k})
\end{array}
$$

Theorem 3.3 (second step)

Line k (there is clearly a numbering problem when working backwards!) is a conjunction and the obvious rule for its creation is $\wedge$-I, thus:

$$
\begin{array}{lll}
\textbf{from } x \in \mathbb{R} & & \\
\quad\vdots & & \\
\mathrm{i} & \mathit{pi}(x) \in \mathbb{R} & ? \\
\mathrm{j} & \mathit{post\text{-}pi}(x, \mathit{pi}(x)) & ? \\
\mathrm{k} & \mathit{pi}(x) \in \mathbb{R} \wedge \mathit{post\text{-}pi}(x, \mathit{pi}(x)) & \wedge\text{-}I(\mathrm{i,j}) \\
\textbf{infer} & \mathit{pre\text{-}pi}(x) \;\Rightarrow\; \mathit{pi}(x) \in \mathbb{R} \wedge \mathit{post\text{-}pi}(x, \mathit{pi}(x)) & \Rightarrow \mathit{vac\text{-}I}(\mathrm{k})
\end{array}
$$

Theorem 3.3 (third step)

The reader should now be able to see how such a proof would be completed. The advantage of proceeding in this way is that the open justifications clearly mark the remaining work. The problem with this style when tackled with pen and ink is knowing how much space to leave: this results in excessive use of a waste-paper basket! A text editor can be used to some advantage, but special-purpose proof editors (see [BM79, Lin88, JL88, RT89]) can offer much more support.

More examples

Subsequent proofs are, for obvious reasons, presented only in their final form (furthermore, only the sequent form is given) and comments on their discovery are made only when some new feature is present. The reader should, however, use the technique of working back from a goal when undertaking the exercises.

In order to illustrate the role of non-trivial pre-conditions the following simple specification is used:

$$\begin{array}{l} \mathit{foo}\ (i\colon \mathbb{N}, j\colon \mathbb{N})\ \ r\colon \mathbb{N} \\ \textbf{pre}\ \ i = 2 \\ \textbf{post}\ \ r = 2 * j \end{array}$$

together with the direct definition:

$$\mathit{foo}(i, j) \quad \triangleq \quad i * j$$

Rule 3.5 The rule form of which is:

$$\boxed{\text{R3.5}}\ \frac{i, j \in \mathbb{N}}{\mathit{foo}(i, j) = i * j}$$

Theorem 3.6 The sequent form of proof obligation 3.1 becomes:

$$\begin{array}{l} m, n \in \mathbb{N}\ \vdash \\ \qquad \textit{pre-foo}(m, n)\ \Rightarrow\ \mathit{foo}(m, n) \in \mathbb{N} \wedge \textit{post-foo}(m, n, \mathit{foo}(m, n)) \end{array}$$

This proof obligation is discharged on page 57. Its proof is similar to that for pi, but its discovery does result in an $\Rightarrow$-I because the goal is an implication which has a non-trivial antecedent.

It is not necessary to produce all proofs at such a fine level of detail. In particular, the substitution steps can be handled less formally (once again, given the proviso that the formal steps can be created should doubt arise). The proof on page 57 might be written as:

from $m, n \in \mathbb{N}$		
1	**from** $m = 2$	
1.1	$\quad foo(m, n) = m * n$	R3.5(h)
1.2	$\quad m * n \in \mathbb{N}$	$\mathbb{N}$,h
1.3	$\quad foo(m, n) \in \mathbb{N}$	$=$-*subs*(1.2,1.1)
1.4	$\quad foo(m, n) = 2 * n$	$=$-*subs*(h1,1.1)
1.5	$\quad post\text{-}foo(m, n, foo(m, n))$	*post-foo*(h,1.3,1.4)
	infer $foo(m, n) \in \mathbb{N} \wedge post\text{-}foo(m, n, foo(m, n))$	$\wedge$-I(1.3,1.5)
2	$\delta(m = 2)$	h
3	$m = 2 \Rightarrow foo(m, n) \in \mathbb{N} \wedge post\text{-}foo(m, n, foo(m, n))$	$\Rightarrow$-I(1,2)
infer	$pre\text{-}foo(m, n) \Rightarrow foo(m, n) \in \mathbb{N} \wedge post\text{-}foo(m, n, foo(m, n))$	*pre-foo*(h,3)

Theorem 3.6: *foo*

from $m, n \in \mathbb{N}$		
1	$m = 2 \Rightarrow m * n \in \mathbb{N} \wedge m * n = 2 * n$	$\mathbb{N}$,h
2	$m = 2 \Rightarrow foo(m, n) \in \mathbb{N} \wedge foo(m, n) = 2 * n$	1, R3.5
infer	$pre\text{-}foo(m, n) \Rightarrow foo(m, n) \in \mathbb{N} \wedge post\text{-}foo(m, n, foo(m, n))$	

Theorem 3.6: outline proof

Exercise 3.2.1 Prove that the specification:

$double\ (x: \mathbb{Z})\ r: \mathbb{Z}$
post $r = 2 * x$

is satisfied by:

$$double(x) \quad \triangle \quad x + x$$

Remember to present the rule form of the direct definition.

Exercise 3.2.2 Prove that the specification:

$conv\ (f: \mathbb{R})\ c: \mathbb{R}$
post $c * 9/5 + 32 = f$

is satisfied by:

$$conv(f) \quad \triangleq \quad (f + 40) * 5/9 - 40$$

Exercise 3.2.3 Prove that the specification:

$choose\ (i{:}\,\mathbb{N})\ j{:}\,\mathbb{N}$
pre $i = 3 \lor i = 8$
post $(i = 3 \ \Rightarrow\ j = 8) \land (i = 8 \ \Rightarrow\ j = 3)$

is satisfied by:

$$choose(i) \quad \triangleq \quad 11 - i$$

Notice that this proof does *not* require case analysis.

Case analysis

Direct definitions of functions can also use conditional expressions. An example is the direct definition of the *max* function:

$$max(i,j) \quad \triangleq \quad \textbf{if } i \le j \textbf{ then } j \textbf{ else } i$$

For this, two inference rules are needed but they are given the same number.

Rule 3.7 The rules are:

$$\boxed{\text{R3.7}}\ \frac{i,j \in \mathbb{Z};\ i \le j}{max(i,j) = j}$$

$$\boxed{\text{R3.7}}\ \frac{i,j \in \mathbb{Z};\ j < i}{max(i,j) = i}$$

The implicit specification is:

$max\ (i{:}\,\mathbb{Z}, j{:}\,\mathbb{Z})\ r{:}\,\mathbb{Z}$
pre true
post $(r = i \lor r = j) \land i \le r \land j \le r$

Theorem 3.8 The proof obligation is:

$$i,j \in \mathbb{Z} \vdash$$
$$\quad pre\text{-}max(i,j) \ \Rightarrow\ max(i,j) \in \mathbb{Z} \land post\text{-}max(i,j,max(i,j))$$

The proof of this sequent is given on page 59. As before, the reader should first check the forward steps in this proof. The generation of this proof introduces one new tactic. Line 4 is generated (as in *pi*) by noticing that the pre-condition is true. In the proof of *pi* the analysis could proceed because the definition of

from $m, n \in \mathbb{Z}$		
1	$m \leq n \vee m > n$	$\mathbb{Z}$,h
2	**from** $m \leq n$	
2.1	$\quad max(m, n) = n$	R3.7(h,h2)
2.2	$\quad max(m, n) \in \mathbb{Z}$	$=$-*subs*(2.1,h)
2.3	$\quad (n = m \vee n = n) \wedge m \leq n \wedge n \leq n$	$\mathbb{Z}$,h,h2,$\wedge$,$\vee$
2.4	$\quad post\text{-}max(m, n, n)$	*post-max*(h,2.3)
2.5	$\quad post\text{-}max(m, n, max(m, n))$	$=$-*subs*(2.4,2.1)
	infer $max(m, n) \in \mathbb{Z} \wedge post\text{-}max(m, n, max(m, n))$	$\wedge$-*I*(2.2,2.5)
3	**from** $m > n$	
3.1	$\quad max(m, n) = m$	R3.7(h,h3)
3.2	$\quad max(m, n) \in \mathbb{Z}$	$=$-*subs*(3.1,h)
3.3	$\quad (m = m \vee m = n) \wedge m \leq m \wedge n \leq m$	$\mathbb{Z}$,h,h3,$\wedge$,$\vee$
3.4	$\quad post\text{-}max(m, n, m)$	*post-max*(h,3.3)
3.5	$\quad post\text{-}max(m, n, max(m, n))$	$=$-*subs*(3.4,3.1)
	infer $max(m, n) \in \mathbb{Z} \wedge post\text{-}max(m, n, max(m, n))$	$\wedge$-*I*(3.2,3.5)
4	$max(m, n) \in \mathbb{Z} \wedge post\text{-}max(m, n, max(m, n))$	$\vee$-*E*(1,2,3)
infer	$pre\text{-}max(m, n) \Rightarrow max(m, n) \in \mathbb{Z} \wedge post\text{-}max(m, n, max(m, n))$	$\Rightarrow$ *vac-I*(4)

Theorem 3.8: *max*

the function is straightforward; here, the expansion of *max* is a long expression which requires simplification. The best way to simplify a conditional is by case analysis. Here, the case distinction is obvious and the sub-goals generated are lines 2 and 3 (and the subsidiary line 1). Once these are identified, the proof is straightforward.

The proofs so far are presented with a great deal of detail. The level of detail can be chosen to suit the problem in hand and, in later proofs, several inference steps are performed in a single line. Furthermore, as the reader becomes confident in the construction of such proofs, only the outer levels of a proof need be recorded; the inner steps can be completed if doubt arises. The key point about such (rigorous) proof outlines is that it is clear what needs to be done to extend them to formal proofs – for this reason, errors are less likely.

In the following specification:

$abs\ (i{:}\,\mathbb{Z})\ r{:}\,\mathbb{Z}$
post $0 \le r \wedge (r = i \vee r = -i)$

the pre-condition which is **true** is omitted; thinking of the pre-condition as permission to ignore certain argument combinations, its omission indicates that the implementation must cater for any arguments (of the required types). Similarly, the proof obligation can be simplified to reflect the fact that E and **true** $\Rightarrow E$ are equivalent expressions.

Exercise 3.2.4 Prove that the specification:

$abs\ (i{:}\,\mathbb{Z})\ r{:}\,\mathbb{Z}$
post $0 \le r \wedge (r = i \vee r = -i)$

is satisfied by:

$$abs(i) \quad \triangleq \quad \textbf{if } i < 0 \textbf{ then } -i \textbf{ else } i$$

Exercise 3.2.5 The *sign* function can be specified:

$sign\ (i{:}\,\mathbb{Z})\ r{:}\,\mathbb{Z}$
post $i = 0 \wedge r = 0 \vee i < 0 \wedge r = -1 \vee i > 0 \wedge r = 1$

Write a direct definition and prove that it satisfies the specification.

Using the specification of a subsidiary function

The specification of *abs* is given in Exercise 3.2.4.

Theorem 3.9 Thus any proposed implementation must be such that:

$$i \in \mathbb{Z} \vdash abs(i) \in \mathbb{Z} \wedge post\text{-}abs(i, abs(i))$$

A direct definition which uses conditional expressions and arithmetic operators is considered in Exercise 3.2.4. Suppose, however, that the following implementation were to be considered:

$$abs(i) \quad \triangleq \quad max(i, -i)$$

It would, of course, be possible to expand out the right-hand side of this definition by using the direct definition of *max*. But in the development of programs, a high-level design step introduces components (via specifications) whose development follows the justification of the design step. The first hint of how this works can be given by making the proof rely only on the implicit specification, rather than the direct definition, of *max*.

from $i \in \mathbb{Z}$		
1	$-i \in \mathbb{Z}$	$\mathbb{Z}$,h
2	$max(i, -i) \in \mathbb{Z}$	max,h,1
3	$abs(i) = max(i, -i)$	R3.10(h)
4	$abs(i) \in \mathbb{Z}$	$=$-*subs*(3,2)
5	$(max(i, -i) = i \vee max(i, -i) = -i) \wedge$ $i \leq max(i, -i) \wedge -i \leq max(i, -i)$	*post-max*(h,1)
6	$0 \leq max(i, -i)$	$\mathbb{Z}$,5
7	$post\text{-}abs(i, max(i, -i))$	*post-abs*(h,2,6,5)
8	$post\text{-}abs(i, abs(i))$	$=$-*subs*(3,7)
infer	$abs(i) \in \mathbb{Z} \wedge post\text{-}abs(i, abs(i))$	$\wedge$-I(4,8)

Theorem 3.9: *abs*

Rule 3.10 As normal, the rule is:

$$\boxed{\text{R3.10}}\ \frac{i \in \mathbb{Z}}{abs(i) = max(i, -i)}$$

The proof of Theorem 3.9 is shown on page 61.

The relationship between the range type of a function and the post-condition can be understood by studying this example. If the signature were changed to:

$abs(i{:}\,\mathbb{Z})r{:}\,\mathbb{N}$

the first conjunct of *post-abs* could be omitted. The overall proof task would not, however, change because it is still necessary to show that the result is of the appropriate type: it would simply be necessary to rearrange the steps. Thus the choice of whether to show constraints by type information or by clauses in a post-condition can be made on pragmatic grounds.

Exercise 3.2.6 Given the specification:

$mult\ (i{:}\,\mathbb{Z}, j{:}\,\mathbb{Z})\ r{:}\,\mathbb{Z}$
post $r = i * j$

prove that:

$mult(i, j) \quad \triangleq \quad$ **if** $i \geq 0$ **then** $multp(i, j)$ **else** $multp(-i, -j)$

satisfies the specification. In making this proof the following properties of *multp* should be assumed:

$multp\ (i{:}\,\mathbb{Z}, j{:}\,\mathbb{Z})\ r{:}\,\mathbb{Z}$
pre $i \geq 0$
post $r = i * j$

Induction

The most powerful way of building up (direct definitions of) functions is by using recursion. A *recursive definition* of a function is one in which the right-hand side of the definition uses the function which is being defined. Speaking operationally, such definitions have to terminate and this is normally achieved by placing the recursive reference in a conditional expression. In addition to the conditional expression and the function being defined, the right-hand side can, of course, use any previously known functions.

Recursive definitions facilitate the construction of powerful functions from the most humble beginnings. Each of the data types in the succeeding chapters starts with a simple set of basic constructing functions or *generators* and then builds other operators and/or functions over these generators. For natural numbers, the generators are very simple: zero is a natural number and the *successor* function (*succ*) generates natural numbers from natural numbers. Thus:

$0{:}\ \mathbb{N}$
$succ{:}\ \mathbb{N} \to \mathbb{N}$

Clearly, one can think of these as the value zero and the function 'plus one'. The number for which the normal (Arabic) symbol is 2 is the result of $succ(succ(0))$. Such a unary notation would be rather clumsy for larger constants, but 0 and *succ* provide a minimal basis for arithmetic. They enable any of the infinite set of (finite) numbers to be generated.

The inverse of *succ* is a (partial) *predecessor* function. This can be characterized by an axiom.

Axiom 3.11 Because $pred(0)$ is not defined, the only property is:

$$\boxed{\text{A3.11}}\ \frac{i \in \mathbb{N}}{pred(succ(i)) = i}$$

General addition (over $\mathbb{N}$) can be defined by a recursive function:

$add : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$
$add(i, j) \quad \triangleq \quad \textbf{if } i = 0 \textbf{ then } j \textbf{ else } succ(add(pred(i), j))$

or, in less pedantic style:

$$add(i,j) \quad \triangleq \quad \textbf{if } i = 0 \textbf{ then } j \textbf{ else } add(i-1,j)+1$$

This can be rendered into inference rules as follows.

Rule 3.12 The non-recursive part of the conditional offers no surprise:

$$\boxed{\text{R3.12}}\ \frac{j \in \mathbb{N}}{add(0,j) = j}$$

Rule 3.13 From the earlier treatment of conditional expressions, the reader should expect a rule of the form:

$$\boxed{\text{R3.13a}}\ \frac{i,j \in \mathbb{N};\ i \neq 0}{add(i,j) = add(i-1,j)+1}$$

For a total function like *add* there is no problem but it is shown in the next section that appropriate rules for partial functions use the recursive call in the hypothesis. This form is used here for uniformity:

$$\boxed{\text{R3.13}}\ \frac{i,j \in \mathbb{N};\ i \neq 0;\ add(i-1,j) = k}{add(i,j) = k+1}$$

Given general addition, multiplication of natural numbers can be defined:

$$multp : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$
$$multp(i,j) \quad \triangleq \quad \textbf{if } i = 0 \textbf{ then } 0 \textbf{ else } multp(i-1,j)+j$$

Further functions, such as exponentiation, can be built up in the same way.

But what of proofs about such recursive definitions? An attempt to proceed with only the tools available so far would show that, for proofs about arbitrary values i, the rule R3.13 offers no way of eliminating the name of the function (add). The proof technique which is used to reason about elements from infinite sets is *induction*. The form of induction rule for natural numbers which suits most of the purposes here[3] is expressed in terms of proving some property p for any natural number.

Axiom 3.14 As above, the emphasis is put on proving a sequent ($n \in \mathbb{N} \vdash p(n)$) rather than a quantified expression ($\forall n \in \mathbb{N} \cdot p(n)$) since the latter can always be obtained by a step using $\forall$-I. To prove some property p: $\mathbb{N} \to \mathbb{B}$ holds for all natural numbers, the induction axiom is:

[3]The use of a sequent as consequent is unconventional but suits the boxed proofs best; similarly, employing a sequent – rather than an implication – in the inductive step fits with the logic described in the next section.

$$\boxed{\mathbb{N}\text{-}ind}\ \frac{p(0);\quad n \in \mathbb{N},\ p(n) \vdash p(n+1)}{n \in \mathbb{N} \vdash p(n)}$$

This rule gets used in proofs of the following shape (the induction step has been written using m – rather than n – as a local variable to emphasize its independence):

from $n \in \mathbb{N}$		
	$\vdots$	
i	$p(0)$	
j	**from** $m \in \mathbb{N},\ p(m)$	
	$\vdots$	
	infer $p(m+1)$	
infer $p(n)$		$\mathbb{N}$-*ind*(i,j)

Skeleton induction proof

Such a proof is called an *inductive proof*; the steps to establish $p(0)$ are the *base case* and the **from/infer** to prove $p(n) \vdash p(n+1)$ is the *induction step*. It is important to understand the roles of the various p assertions which arise. The $p(n)$ which is the hypothesis of the inductive step is an assumption; the consequent of that same inner **from/infer** $(p(n+1))$ has only been proved under the assumption. The final $p(n)$ of the outer **from/infer** has been proven (by induction) for an arbitrary natural number.

Many people have a feeling of unease when they first encounter inductive reasoning. One way to gain confidence in inductive proofs is to view them as recipes for *creating* proofs for arbitrary natural numbers. Suppose someone were to challenge a property which had been proved by induction. If they doubt that the property is true for a specific number, say 7, a proof could be generated by:

- copying out the proof of $p(0)$;
- generating 7 versions of the inductive step substituting successive natural numbers in each case.

The resulting proof would be long and tedious but would show that $p(7)$ held without appeal to $\mathbb{N}$-*ind*: the inductive proof can be used as a recipe for generating a proof for any natural number. This claim relies on the fact that any natural number is finite (even though there is an infinite set of such numbers).

from $n \in \mathbb{N}$		
1	$sumn(0) = 0$	R3.15
2	$sumn(0) \in \mathbb{N}$	1, $\mathbb{N}$
3	$0 = 0 * (0 + 1)/2$	$\mathbb{N}$
4	$sumn(0) = 0 * (0 + 1)/2$	$=$-*subs*(3,1)
5	$post\text{-}sumn(0, sumn(0))$	*post-sumn*,4
6	$sumn(0) \in \mathbb{N} \wedge post\text{-}sumn(0, sumn(0))$	$\wedge$-*I*(2,5)
7	**from** $n \in \mathbb{N},\ sumn(n) \in \mathbb{N},\ post\text{-}sumn(n, sumn(n))$	
7.1	$n + 1 \neq 0$	h7,$\mathbb{N}$
7.2	$n + 1 \in \mathbb{N}$	h7,$\mathbb{N}$
7.3	$sumn(n) = n * (n + 1)/2$	*post-sumn*,ih7
7.4	$sumn(n + 1) = n + 1 + n * (n + 1)/2$	R3.16(7.2,7.1,7.3)
7.5	$sumn(n + 1) \in \mathbb{N}$	7.4,$\mathbb{N}$
7.6	$sumn(n + 1) = (n + 1) * (n + 2)/2$	7.4,$\mathbb{N}$
7.7	$post\text{-}sumn(n + 1, sumn(n + 1))$	*post-sumn*,7.6
	infer $sumn(n + 1) \in \mathbb{N} \wedge post\text{-}sumn(n + 1, sumn(n + 1))$	$\wedge$-*I*(7.5,7.7)
infer $sumn(n) \in \mathbb{N} \wedge post\text{-}sumn(n, sumn(n))$		$\mathbb{N}$-*ind*(6,7)

Theorem 3.17: *sumn*

The earlier examples of *add* and *multp* are used in proofs which are the subject of exercises below. The first example used here is a function *sumn* which is intended to compute the sum of the first n natural numbers. Its definition is written (recursively) as:

$$sumn : \mathbb{N} \to \mathbb{N}$$
$$sumn(n) \quad \triangleq \quad \mathbf{if}\ n = 0\ \mathbf{then}\ 0\ \mathbf{else}\ n + sumn(n - 1)$$

This can be expressed as two inference rules.

Rule 3.15 The base case is:

$$\boxed{\text{R3.15}}\ \frac{}{sumn(0) = 0}$$

Rule 3.16 The recursive case is:

$$\boxed{\text{R3.16}}\ \frac{n \in \mathbb{N};\ n \neq 0;\ sumn(n - 1) = k}{sumn(n) = n + k}$$

To show that this possesses the known arithmetic property, it is 'specified' by:

$sumn\ (n{:}\,\mathbb{N})\ r{:}\,\mathbb{N}$
post $r = n * (n+1)/2$

This gives rise to its proof obligation.

Theorem 3.17 The sequent form is:

$$n \in \mathbb{N} \vdash sumn(n) \in \mathbb{N} \wedge post\text{-}sumn(n, sumn(n))$$

The required proof is given on page 65. Notice that the p of $\mathbb{N}$-*ind* (p: $\mathbb{N} \to \mathbb{B}$) is $sumn(n) \in \mathbb{N} \wedge post\text{-}sumn(n, sumn(n))$. Once again, the proof is presented in the easiest order for reading. Its discovery results from using $\mathbb{N}$-*ind* to generate both line 6 and the subsidiary proof numbered 7; the detailed steps are routine with the algebra from 7.4 to 7.6 being the core of the induction argument. (Notice that the appeal in line 7.3 to the hypothesis of base 7 emphasizes that it is an inductive hypothesis by writing 'ih7'.)

An inductive proof can be used to show that a recursively defined function, for squaring a number, satisfies the implicit specification:

$sq\ (i{:}\,\mathbb{N})\ r{:}\,\mathbb{N}$
post $r = i^2$

Given the definition:

$sq : \mathbb{N} \to \mathbb{N}$
$sq(i) \quad \triangleq \quad \textbf{if } i = 0 \textbf{ then } 0 \textbf{ else } 2 * i - 1 + sq(i-1)$

The inference rules are:

Rule 3.18 For the base case:

$$\boxed{\text{R3.18}}\ \frac{}{sq(0) = 0}$$

Rule 3.19 For the recursive case:

$$\boxed{\text{R3.19}}\ \frac{i \in \mathbb{N};\ i \neq 0;\ sq(i-1) = k}{sq(i) = 2 * i - 1 + k}$$

Theorem 3.20 The proof obligation is:

$$n \in \mathbb{N} \vdash sq(n) \in \mathbb{N} \wedge post\text{-}sq(n, sq(n))$$

The proof is shown on page 68. Lines 1 to 6 constitute the basis of the proof and the inductive step is labelled 7. The sub-goals are generated by applying the induction rule.

Exercise 3.2.7 Prove the general addition function *add* satisfies the specification:

$add\ (i:\mathbb{N}, j:\mathbb{N})\ r:\mathbb{N}$
post $r = i + j$

Base the inductive proof on the rules 3.12 and 3.13.

Exercise 3.2.8 The rules of the preceding exercise were written for the 'less pedantic' form of the recursive definition of *add*. Write a pair of rules (which should use neither $+$ nor $-$) for the definition of *add*. This shows more clearly how recursion builds up the language of functions over natural numbers from just its generators .

Exercise 3.2.9 Show that the recursive definition of *multp* given in this section satisfies the specification:

$multp\ (i:\mathbb{N}, j:\mathbb{N})\ r:\mathbb{N}$
post $r = i * j$

More about induction

There are a few extra points which are worth noting about induction over the natural numbers because induction is to be a crucial tool in handling the data types presented in later chapters. It is explained above that induction is needed because the set of natural numbers ($\mathbb{N}$) is infinite. But each member of that set is a finite number; it is the *unbounded size* of the numbers which requires inductive proofs.

There are also forms of induction rule other than $\mathbb{N}$-*ind*. For example, an almost mechanical rewriting yields:

Axiom 3.21 An induction rule which relies on the presence of *predecessor*:

$$\boxed{\mathbb{N}\text{-}indp}\ \frac{p(0);\quad n \in \mathbb{N}_1, p(n-1) \vdash p(n)}{n \in \mathbb{N} \vdash p(n)}$$

Both $\mathbb{N}$-*ind* and $\mathbb{N}$-*indp* rely on only one step of *successor*; a more powerful rule is one which permits the assumption that the required property, p, is true of all predecessors of the number for which $p(n)$ is the goal of the inductive step. Paradoxically, this rule appears simpler than its cousins because the base case becomes a special case of the inductive step.

Axiom 3.22 The so-called, *complete induction* rule is:

from $n \in \mathbb{N}$		
1	$sq(0) = 0$	R3.18
2	$sq(0) \in \mathbb{N}$	1,$\mathbb{N}$
3	$0 = 0^2$	$\mathbb{N}$
4	$sq(0) = 0^2$	$=$-*subs*(3,1)
5	$\textit{post-sq}(0, sq(0))$	*post-sq*(4)
6	$sq(0) \in \mathbb{N} \wedge \textit{post-sq}(0, sq(0))$	$\wedge$-*I*(2,5)
7	**from** $n \in \mathbb{N},\ sq(n) \in \mathbb{N},\ \textit{post-sq}(n, sq(n))$	
7.1	$sq(n) = n^2$	*post-sq*(ih7)
7.2	$n + 1 \in \mathbb{N}_1$	$\mathbb{N}$,h7
7.3	$(n+1)^2 \in \mathbb{N}$	$\mathbb{N}$,7.2
7.4	$(n+1)^2 = n^2 + 2 * n + 1$	$\mathbb{N}$,h7
7.5	$= sq(n) + 2 * n + 1$	$=$-*subs*(7.4,7.1)
7.6	$= sq(n+1)$	R3.19(7.2,7.5)
7.7	$sq(n+1) \in \mathbb{N}$	$=$-*subs*(7.6,7.3)
7.8	$\textit{post-sq}(n+1, sq(n+1))$	*post-sq*(7.2,7.7,7.6)
	infer $sq(n+1) \in \mathbb{N} \wedge \textit{post-sq}(n+1, sq(n+1))$	$\wedge$-*I*(7.7,7.8)
infer $sq(n) \in \mathbb{N} \wedge \textit{post-sq}(n, sq(n))$		$\mathbb{N}$-*ind*(6,7)

Theorem 3.20: *sq*

$$\boxed{\mathbb{N}\text{-}cind}\ \frac{n \in \mathbb{N},\ (\forall m \in \mathbb{N} \cdot m < n \ \Rightarrow\ p(m)) \ \vdash\ p(n)}{n \in \mathbb{N} \ \vdash\ p(n)}$$

There are also two subtleties of implicit specifications which are worth emphasizing. Where an implicit specification, or rather its post-condition, underdetermines the result of a function; it is required that the implementation is a function. Therefore it can always be assumed that $x = y \ \Rightarrow\ f(x) = f(y)$ for any f. (Section 3.4 shows that a truly non-deterministic approach is taken to the specifications of operations.)

The converse situation is where the post-condition admits no valid answer. Such self-contradictory post-conditions are a risk which hide behind the 'advantages' of implicit definitions (cf. Russell's quote at the beginning of this chapter). Strictly, each implicit function specification should be shown to be *satisfiable*:

$$\forall d \in D \cdot \textit{pre-f}(d) \ \Rightarrow\ \exists r \in R \cdot \textit{post-f}(d, r)$$

This has not been done here because of the straightforward form of most of the specifications. The topic of satisfiability is taken up in Section 5.2, in the context of operations, when there is a greater danger of error.

Exercise 3.2.10 (*) Use the rule $\mathbb{N}$-*cind* to prove that:

$$
\begin{array}{l}
multp : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\
multp(i, j) \quad \triangleq \\
\qquad \textbf{if } i = 0 \textbf{ then } 0 \textbf{ else if } is\text{-}even(i) \textbf{ then } 2 * multp(i/2, j) \textbf{ else } j + multp(i-1, j)
\end{array}
$$

satisfies the specification given earlier in this section. Develop other algorithms which require proofs by complete induction because they split a task into parts which give rise to recursive calls on other than predecessors of the parameter of the function.

3.3 Reasoning about partial functions

Partial functions

A *total* function yields a result for any argument in the domain – as given in the signature – of the function. Functions which do not meet this requirement (i.e. do not always yield a result) are called *partial*. Many of the standard fields of mathematics assume that all functions are total whereas partial functions arise naturally in software applications. The importance of partial functions has been recognized above by recording their pre-conditions. This section reviews the impact of partial functions on the logic used to reason about them.

Partial functions are distinguished here by recording a non-trivial pre-condition. If the domain is a single set, it is straightforward to define a restricted set which includes only those elements which satisfy the pre-condition; the function then becomes total over the restricted set. The more interesting pre-conditions are those which relate different parameters: in such cases, it is less natural[4] to make functions total.

Consider the following example:

$$
\begin{array}{l}
subp\ (i: \mathbb{Z}, j: \mathbb{Z})\ r: \mathbb{Z} \\
\textbf{pre } i \geq j \\
\textbf{post } r = i - j
\end{array}
$$

Informally, it is clear that this specification is satisfied by the recursive function:

$$subp(i, j) \quad \triangleq \quad \textbf{if } i = j \textbf{ then } 0 \textbf{ else } subp(i, j+1) + 1$$

[4]Such pre-conditions require defining Cartesian products and restrictions thereon.

As discussed earlier, terms can be formed by applying a function to arguments of the appropriate type. Thus $subp(5,3)$ is a term (whose value is 2). There is, however, a problem with terms built from functions where the arguments do not satisfy the pre-condition of the function: what, for example, is to be made of the term $subp(3,5)$? In programming terms, it could be said that *subp* fails to terminate; here, it fits the context better to say that the term does not denote a value. This leads to problems with $subp(i,j)$ since the question of whether or not this term denotes a value depends on the values (as provided by the context) of i and j.

The quantifier form of proof obligation 3.1 for *subp* is:

$$\begin{array}{l}\forall i,j \in \mathbb{Z} \cdot \\ \qquad pre\text{-}subp(i,j) \;\Rightarrow\; subp(i,j) \in \mathbb{N} \wedge post\text{-}subp(i,j,subp(i,j))\end{array}$$

which expands into:

$$\forall i,j \in \mathbb{Z} \cdot i \geq j \;\Rightarrow\; subp(i,j) \in \mathbb{N} \wedge subp(i,j) = i - j$$

When the antecedent of the implication is false, the term involving *subp* does not denote a natural number. It is tempting to say that this problem can be ignored because the implication could be considered to be true whenever its antecedent is false (regardless of the consequent). This is, in fact, one property of the logic studied here. However, the whole topic has to be put on a firm footing – for example, something must be done about the fact that the standard (two-valued) truth tables mentioned in Section 1.1 make no mention of propositions which fail to denote a Boolean value. Note that using the definition of implication does not resolve the problem since:

$$\forall i,j \in \mathbb{Z} \cdot i < j \vee subp(i,j) \in \mathbb{N} \wedge subp(i,j) = i - j$$

has an undefined term in its second disjunct when its first is true.

Many more examples arise in this book where terms fail to denote values and the challenge is to provide a logical system which handles this problem. Far from being a contrived difficulty, this is a common feature of programs (and the fragments from which they are built). Loop constructs may fail to terminate for some input values and the logic to be used in their proofs must have a way of discussing the set over which the loop can be safely used.

Truth tables

If terms fail to denote values (and hence propositions fail to denote truth values) what meaning is to be given to the logical operators? The approach adopted here is to extend the meaning of the operators in a specific way. In order to explain the extension, truth tables are used to indicate a model theory. In these tables, the absence of a value is marked by $*$; but there is no sense in which this is a new value – it is just a reminder that no value is available. Because nine cases must now be considered, the truth tables are presented in a compact square style in preference to the series of columns used in Section 1.1. The extended truth table for disjunction is:

$\vee$	**true**	$*$	**false**
true	**true**	**true**	**true**
$*$	**true**	$*$	$*$
false	**true**	$*$	**false**

In a sense which is made formal below, this is the 'most generous' extension of the two-valued truth table in that a result is given whenever possible. Notice that the truth table is symmetrical, as also is that for conjunction:

$\wedge$	**true**	$*$	**false**
true	**true**	$*$	**false**
$*$	$*$	$*$	**false**
false	**false**	**false**	**false**

Properties such as commutativity are natural consequences of the symmetry of these tables. The table for negation is:

$\neg$	
true	**false**
$*$	$*$
false	**true**

The truth tables for implication and equivalence are derived by viewing them as the normal abbreviations:

$\Rightarrow$	**true**	$*$	**false**
true	**true**	$*$	**false**
$*$	**true**	$*$	$*$
false	**true**	**true**	**true**

$\Leftrightarrow$	**true**	$*$	**false**
true	**true**	$*$	**false**
$*$	$*$	$*$	$*$
false	**false**	$*$	**true**

The reader should observe that the truth table for implication resolves the problem encountered above. When the antecedent of the proof obligation for

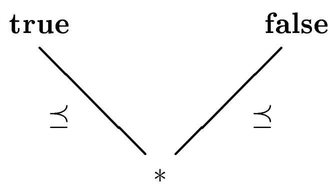

Figure 3.2 Ordering for truth values

subp is false, the whole implication is true even though a term in the consequent has no value.

It is useful to think of these operators being evaluated by a program which has access to the parallel evaluation of its operands. As soon as a result is available for one operand, it is considered; if the single result determines the overall result (e.g. one true for a disjunction), evaluation ceases and the (determined) result is returned.

A more mathematical characterization of the chosen tables can be given. The description in terms of a parallel program has the property that any result, delivered on the basis of incomplete information, will not be wrong however the information is completed (e.g. having one true operand for a disjunction, it does not matter whether the other operand evaluates to true or false). The concept of 'v_1 could become v_2 if evaluated further' defines an ordering relation. For the Boolean values this can be written:

$* \preceq \mathbf{true}$
$* \preceq \mathbf{false}$

This is pictured in Figure 3.2. A function is said to be *monotone* in an ordering if it respects the ordering in the sense that larger arguments give rise to larger results. That is, f is monotone if – and only if:

$$a \preceq b \;\Rightarrow\; f(a) \preceq f(b)$$

For example, given the obvious ordering on the integers($<$), addition is monotone in both of its operands while subtraction is monotone only in its first operand. The truth tables which are given above are monotonic extensions of the classical (two-valued) tables. In fact, they are the strongest such tables which do not contradict the (two-valued) tables of classical logic.

Proof theory

What, however, is to be the proof theory for this logic of partial functions (LPF)? The proof theory introduced in Chapters 1 and 2 is designed for this logic! That proof theory is consistent with the normal (two-valued) logic but cannot prove all results – it is incomplete; for LPF, whose model theory is sketched above, the axiomatization is *complete* (i.e. all true statements can be proved).[5]

All results proved using the logic introduced above are, then, true in classical logic. Given the full axiomatization in the *Teacher's Notes*, all results which are true of the LPF model can be proved. The essential difference between LPF and classical logic is that some results which are true in the latter can not be proved in LPF. The most obvious difference between LPF and classical logic is that the so-called 'law of the excluded middle' does not hold in the former. Looking at the truth table for 'not' makes it clear that:

$$E \vee \neg E$$

need not be true since it relies on E denoting a value. A simple example of why this weakness is considered a virtue is that with partial functions (e.g. division) there is no reason to expect:

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

to be true. On the other hand, a property like:

$$\forall x \in \mathbb{R} \cdot x = 0 \vee x/x = 1$$

is true in LPF and can be proved without difficulty – see page 74.

The lack of the law of the excluded middle is an intended weakness in LPF. It does, however, make certain proofs more difficult than in classical logic. For example, a slightly shorter proof of:

$$(E_1 \vee E_2) \wedge (E_1 \vee E_3) \vdash E_1 \vee E_2 \wedge E_3$$

than that needed in LPF is possible in classical logic. The same point explains the need for a longer axiomatization for LPF than is needed for classical logic: without the $\neg \vee$-$E/\neg \vee$-I rules the system would not be complete; in classical logic these properties follow from the law of the excluded middle.

Since the law of the excluded middle does not hold, nor does $E \Rightarrow E$: it does not have a value if E does not. This has the deeper consequence that the

[5]It is an important property of a notation that it can express sufficient things. For the standard logic, 'or' and 'not' are *expressively complete* in that, with just these two operators, any truth table can be generated. The wholly **true** (respectively, **false**) tables can be represented by logical expressions. Because (in LPF) these constants cannot be generated in this way, two constants must be explicitly brought into the axiomatization of LPF.

from $x \in \mathbb{R}$		
1	$x = 0 \vee x \neq 0$	h,$\mathbb{R}$
2	**from** $x = 0$	
	infer $x = 0 \vee x/x = 1$	$\vee$-I(h2)
3	**from** $x \neq 0$	
3.1	$x/x = 1$	$\mathbb{R}$,h,h3
	infer $x = 0 \vee x/x = 1$	$\vee$-I(3.1)
infer $x = 0 \vee x/x = 1$		$\vee$-E(1,2,3)

Illustrative LPF proof

so-called 'deduction theorem' of classical logic does not hold; knowing:

$$E_1 \vdash E_2$$

does not justify:

$$\vdash E_1 \Rightarrow E_2$$

unless it is also known that E_1 is defined (i.e. $\delta(E_1)$).

Many of the tautologies of classical logic are not true in LPF. This is a direct consequence of the need, in LPF, for expressions to be defined. It is a pleasing property of LPF that true judgements can be formed from the tautologies of classical logic by writing $\delta(E_i)$ to the left of the turnstile for each proposition. Thus one can in LPF make the reliance on definedness explicit by presenting judgements (sequents) with exactly the required $\delta(E_i)$ assumptions.

Notions of equality

Proofs in preceding sections have imported information about function definitions into proofs by using inference rules. It is now possible to clarify why this approach fits well with LPF. Consider again the definition:

$$subp(i, j) \quad \triangleq \quad \textbf{if } i = j \textbf{ then } 0 \textbf{ else } subp(i, j+1) + 1$$

The equality written *within* such direct definitions (e.g. as in $i = j$) is 'weak' in the sense that it is undefined if either of its operands is undefined. This is, in fact, the only reasonable interpretation for something which must clearly be computable.

But this weak equality is not, in general, adequate for the interpretation of the definitions themselves. Given the definition above, the term $subp(3,5)$ is identical in value to $subp(3,6)+1$ even though both terms are undefined. The definition has been written with a special symbol($\triangleq$) as a reminder that only one definition should occur for any function; in general, there is a relational operator for this stronger equality ($==$). The tables which follow contrast these two notions. Here, the undefined values are indexed by their type (e.g. $*_{\mathbb{N}}$).

$=$	0	1	2	$\ldots$	$*_{\mathbb{N}}$
0	**true**	**false**	**false**		$*_{\mathbb{B}}$
1	**false**	**true**	**false**		$*_{\mathbb{B}}$
2	**false**	**false**	**true**		$*_{\mathbb{B}}$
$\vdots$					
$*_{\mathbb{N}}$	$*_{\mathbb{B}}$	$*_{\mathbb{B}}$	$*_{\mathbb{B}}$		$*_{\mathbb{B}}$

The truth-table for the non-strict operator can be written as:

$==$	0	1	2	$\ldots$	$*_{\mathbb{N}}$
0	**true**	**false**	**false**		**false**
1	**false**	**true**	**false**		**false**
2	**false**	**false**	**true**		**false**
$\vdots$					
$*_{\mathbb{N}}$	**false**	**false**	**false**		**true**

It should be clear that strong equality ($==$) is not monotonic: it is never used *within* function definitions.[6] The bound variables of quantifiers range only over the proper elements (not $*$) of sets.

Reverting again to the *subp* example, it is possible to explain in more detail the form of the inference rules which have to be created for partial functions. If the direct definition of *subp* itself is used in a proof, it would introduce a strong equality ($==$) and complicate the proof since the required result (cf. proof obligation 3.1) contains a weak equality ($=$) and the proof would be complicated by virtue of having to reason about two notions of equality. Formulating the inference rule for the non-recursive case of the conditional expression in the definition of *subp* presents no difficulty because all of the terms are defined.

Rule 3.23 Thus:

[6]The current formulation of LPF has gone to pains to avoid – in normal proofs – reasoning about two notions of equality. Also, other non-monotonic operators (e.g. Δ) are only used in meta-proofs such as those which justify the inference rule creation from function definitions.

from $i, j \in \mathbb{Z}$		
1	$i - 0 = i$	h, $\mathbb{Z}$
2	$subp(i, i) = 0$	h, R3.23
3	$subp(i, i - 0) = 0$	$=$-$subs$(1,2)
4	**from** $n \in \mathbb{N};\ subp(i, i - n) = n$	
4.1	$\quad i - (n + 1) \in \mathbb{Z}$	h, h4, $\mathbb{Z}$
4.2	$\quad i \neq i - (n + 1)$	h, h4, $\mathbb{Z}$
	infer $subp(i, i - (n + 1)) = n + 1$	h, 4.1, 4.2, ih4, R3.24
5	$\forall n \in \mathbb{N} \cdot subp(i, i - n) = n$	$\forall$-I($\mathbb{N}$-ind(3, 4))
6	**from** $i \geq j$	
6.1	$\quad (i - j) \in \mathbb{N}$	$\mathbb{N}$, h6
	infer $subp(i, j) = i - j$	$\forall$-E(5, 6.1), $\mathbb{Z}$
7	$\delta(i \geq j)$	h, $\mathbb{Z}$
infer $i \geq j \ \Rightarrow\ subp(i, j) = i - j$		$\Rightarrow$-I(6,7)

Proof about *subp* function

$$\boxed{\text{R3.23}}\ \frac{i \in \mathbb{Z}}{subp(i, i) = 0}$$

As mentioned on page 63, care is required with the recursive case. If the rule is presented with a weak equality in the hypothesis, it behaves exactly as the operational understanding of the function leads one to expect: conclusions about continued applications can only be drawn if the next lower case is actually defined.

Rule 3.24 Thus:

$$\boxed{\text{R3.24}}\ \frac{i_1, i_2 \in \mathbb{Z};\ i_1 \neq i_2;\ subp(i_1, i_2 + 1) = i_3}{subp(i_1, i_2) = i_3 + 1}$$

These rules can be used to create the proof on page 76 which can then be subjected to $\forall$-I to justify the expression which was considered problematic at the beginning of this section.

Exercise 3.3.1 Check that the truth tables for the propositional operators are monotonic.

Exercise 3.3.2 Propositional operators can be defined by conditional expressions as discussed in Exercise 1.1.3 on page 6. Draw up the truth tables for these operators. Contrast these truth tables with the symmetrical ones defined above; why cannot the conditional expressions form the symmetrical tables?

Exercise 3.3.3 Section 1.1 included an informal argument for the following sequent:

$$E_1 \wedge E_2 \vee \neg E_1 \wedge E_3 \vdash (E_1 \Rightarrow E_2) \wedge (\neg E_1 \Rightarrow E_3)$$

Produce a formal proof of this.

Consider the reverse sequent:

$$(E_1 \Rightarrow E_2) \wedge (\neg E_1 \Rightarrow E_3) \vdash E_1 \wedge E_2 \vee \neg E_1 \wedge E_3$$

why can this not be proved? What single additional assumption makes the proof possible?

Exercise 3.3.4 A true sequent of classical logic is:

$$E_1 \Rightarrow (E_2 \Rightarrow E_3) \vdash (E_1 \Rightarrow E_2) \Rightarrow (E_1 \Rightarrow E_3)$$

Show that this is not true in LPF and discuss what change makes it true.

Exercise 3.3.5 Consider the following sequents and indicate additional assumptions which permit their proofs (which should then be written):

$$E_1 \vee (E_2 \Leftrightarrow E_3) \vdash E_1 \vee E_2 \Leftrightarrow E_1 \vee E_3$$
$$E_1 \vee \neg (E_2 \Leftrightarrow E_3) \vdash \neg (E_1 \vee E_2 \Leftrightarrow E_1 \vee E_3)$$

3.4 Implicit specification of operations

Operations

The form of implicit specification introduced in Section 3.1 covers mathematical functions which manipulate numbers. In two respects, these specifications need to be extended in order to cope with the tasks faced by most programmers. The major extension is to cope with the fact that the majority of interesting programs manipulate complex data structures. It would be a mistake to write specifications in terms of the data types of some specific programming language; mathematical abstractions can be used to describe the function of a program without forcing the specification to handle the efficiency considerations which cause the programs themselves to become complicated. Chapter 4 introduces the first of these abstractions: set notation is shown to be a useful tool for writing some specifications. Further collections of notation are covered in Chapters 5 to 7. In contrast, the transition from mathematical functions to programs requires only a minor extension, which is described in this section, to the implicit specification notation of Section 3.1. Programs, as distinct from functions, can be characterized by observing that their execution is affected by, and in turn affects, a *state*. Enthusiasts for functional programming would argue that states and side effects bring much avoidable complexity. On the other hand, the restriction to functions necessitates making copies of those data structures which require modification. The efficiency implications of this copying are not acceptable to today's mainstream computing practitioners and this situation appears unlikely to change until new, special-purpose, machine architectures are developed. It is not the intention here to take a dogmatic position on functional versus procedural programming styles. This section shows that the notational extension from the former to cope with the latter is minor. More importantly, the more extensive material on data structures transcends the distinction.

Specifications can be written for whole programs, parts thereof, or even – as exercises in the notation – single statements. The most common practical use (i.e. not just for exercises) of such specifications is for something of about the size of a procedure in a programming language. A generic name is needed for these different objects – here the word *operation* is used to cover any piece of program-like text. The concern in this section is, then, with the implicit specification of operations.

Functions provide a fixed mapping from input to output. For example:

$$double(i) \quad \triangleq \quad 2 * i$$

yields 4 when applied to 2 whether it has previously been applied to 99 or not. Operations have a (hidden) state which can be used to record values which affect subsequent results. For example, an accumulator operation which outputs the sum of all inputs, might respond to the first input of 2 with 2; to 99 with 101; and to a second 2 with 103.

The *state* of an operation is the collection of external variables which it can access and change. Thus, for a Pascal procedure, it would be those non-local variables of the procedure which affect, or are affected by, execution of the procedure; for a whole program, the state might be a database.

Specifying a calculator

As an introductory example, consider a collection of operations for a simple calculator. The state here consists of a single external variable which is a register (*reg*) containing a natural number. This external variable is the link between the operations. An operation which stores its argument into this register is:

$LOAD\ (i{:}\,\mathbb{N})$
ext wr $reg\ :\ \mathbb{N}$

post $reg = i$

By convention, the names of operations are written in upper-case letters. The first line of an operation specification is similar to that for a function. The second part records those entities to which an operation has external (**ext**) access: variable names are preceded by an indication of whether access is read only (**rd**) or read and write (**wr**); the name of each variable is followed by its type. The post-condition is a truth-valued function of the parameters and the values of the external variables – in this case the value of *reg* after execution of the operation. Thus the post-condition requires that the *LOAD* operation stores the value of its parameter into the register.

An operation which requires read only access to the register is:

$SHOW\ ()\ r{:}\ \mathbb{N}$
ext rd $reg : \mathbb{N}$
post $r = \overleftarrow{reg}$

Here, the post-condition refers to the value of *reg* prior to the execution of the operation. Such values are marked with a backward-pointing hook. In this case, since the operation only has read access, it would have made no difference had the hook been omitted. The convention below is, in fact, to omit the hook on read-only variables, thus:

$SHOW\ ()\ r{:}\ \mathbb{N}$
ext rd $reg : \mathbb{N}$
post $r = reg$

In order to clarify the difference between the access modes (**rd**, **wr**) to external variables, the reader should understand that an equivalent specification would be:

$SHOW\ ()\ r{:}\ \mathbb{N}$
ext wr $reg : \mathbb{N}$
post $reg = \overleftarrow{reg} \wedge r = reg$

The first conjunct in the post-condition is necessary since the operation is marked here as having write access and the final value would otherwise be unconstrained.

A simple incrementing operation can be specified:

$ADD\ (i{:}\ \mathbb{N})$
ext wr $reg : \mathbb{N}$
post $reg = \overleftarrow{reg} + i$

$$
\begin{array}{l}
OP\ (p{:}\ Tp)\ r{:}\ Tr \\
\textbf{ext rd}\ v_1\ :\ T_1, \\
\quad\ \ \textbf{wr}\ v_2\ :\ T_2 \\
\textbf{pre}\ \cdots p \cdots v_1 \cdots v_2 \cdots \\
\textbf{post}\ \cdots p \cdots v_1 \cdots \overleftarrow{v_2} \cdots r \cdots v_2 \cdots
\end{array}
$$

Figure 3.3 Format of operation specification

None of the operations $LOAD$, $SHOW$ or ADD have pre-conditions. The convention that omitted pre-conditions are assumed to be **true** is adopted from function specifications. A pre-condition is required for the operation which performs integer division by its parameter – yielding the result as answer and leaving the remainder in the register:

$$
\begin{array}{l}
DIVIDE\ (d{:}\ \mathbb{N})\ r{:}\ \mathbb{N} \\
\textbf{ext wr}\ reg\ :\ \mathbb{N} \\
\textbf{pre}\ d \neq 0 \\
\textbf{post}\ d * r + reg = \overleftarrow{reg}\ \wedge\ reg < d
\end{array}
$$

The identifiers in the pre-condition are undecorated although they refer to the values prior to execution of the operation. If the pre-condition is thought of as being placed before the operation and the post-condition after the operation (cf. Figure 3.3), the undecorated values apply – in both cases – to the values of the variables at the position of the logical expression.[7]

The states referred to in a post-condition are those prior to and after execution of an operation: any internal states which arise are of no concern to the specification.

One danger with simple examples – in particular with deterministic operations – is that the post-conditions appear to be rather like assignment statements. It is important that *post-ADD* is read as a logical expression which asserts a relationship between values. Fortunately, more interesting examples make this point clear. In *post-DIVIDE* the technique of characterizing a result by conjoined conditions is adopted from function specifications.

The choice between making entities part of the state or of having additional parameters or results is up to the user. Essentially the state of a collection of operations is a hidden data type whose behaviour can be observed via the

[7]The other reasons for this convention become clear in Chapter 10 where proof obligations for operation decomposition are considered.

visible types used as input and output to the operations. So the use of the external clause is governed by the application: it facilitates a distinction in the specification between parameters and variables which are accessed by side effect. In the calculator example, it would be possible to replace the entities involved in the parameters/result by external variables. The decision as to where entities should appear is a pragmatic one. All parameters are assumed here to be passed by value.

The collection of operations for the calculator can be collected into a module. The syntax for modules is introduced in Chapter 9; for most of this book, such grouping is performed informally by the surrounding text. One thing that the module syntax provides is a way of defining initial states. In the textual descriptions of collections of operations, the initial state is normally identified by introducing a variable subscripted with zero. Thus, if the initial state for the calculator module contains a zero:

$$reg_0 = 0$$

A format for specifications of individual operations is indicated in Figure 3.3. Comparing this with Figure 3.4, the pre-condition defines the expected starting conditions for OP – it is, in general, a truth-valued function of the input parameters and the values of the external variables before the operation. None of these identifiers are decorated. The post-condition is a truth-valued function of the parameters, results, values of all external variables prior to execution of the operation and (for read/write variables) their values after the operation. Since there is, in post-conditions, a need to distinguish between two values for write variables, the value before execution of the operation is decorated with a hook. In both pre- and post-conditions, the undecorated identifiers refer to the values 'where the condition applies'. The identifiers within the pre- and post-conditions, whether hooked or not, become bound within the operation specification by the variable names.

It is conceded in Section 3.1 that the choice of the term 'post-condition' is not entirely apposite. With operations, the post-conditions are truth-valued functions of the values of the state before and after the operation: the use of 'post' suggests when it is expected to hold (i.e. after execution). It is as well to follow common usage rather than coin some new term like 'input/output relation'.

Many different sequences of operations can result in the same state. For example, reg would have the value 1 after any of the following sequences:

$LOAD(1)$
$LOAD(0); ADD(1)$
$LOAD(7); DIVIDE(3)$

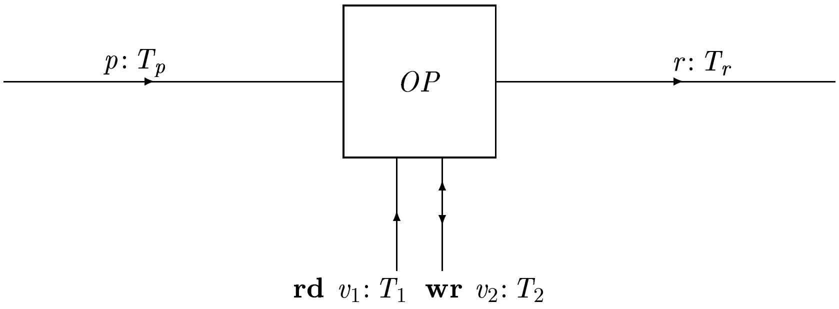

Figure 3.4 Picture of operation specification

Looking just at the state, there is no way of knowing which operations led to its current value. The important property is that this value determines the effect of the next operation: the history itself is not important.[8] As more complex applications are studied, the task of eliminating irrelevant detail (about the history of operations) from the state becomes important. It is precisely because the state contains the essential details of what does affect subsequent behaviour that it is an aid to perspicuous specifications. It is the need to refer to two states, in operation specifications, which necessitates some distinguishing decoration (here hooks for old values). With functions this could be avoided, as in $\textit{post-f}(n, f(n))$, because $f(n)$ is an expression for the result. It must be accepted that functions are more tractable mathematical objects than operations. One way to try to hide the difference is to regard operations as functions over the history of all state changes. This can be done. But this hides the fact that different histories give rise to situations which are not detectably different. The experience in the 'VDM school' is that clearer specifications result from a direct acceptance of the notion of state.

Specifying parts of a program

Another example (computing factorial) can be used to relate the specification format of Figure 3.3 to programs. Informally, it is clear that the specification:

[8] Mathematically, one could say that the state induces an equivalence class on histories. An extreme choice of state could just store all operations executed. The process of abstraction fixes what is irrelevant in the history and should yield a state which captures only that information which influences future operations.

$$
\begin{array}{l}
FACT \\
\textbf{ext wr } n \ : \mathbb{N}, \\
\qquad \textbf{wr } fn : \mathbb{N} \\
\textbf{post } fn = \overleftarrow{n}!
\end{array}
$$

is satisfied by the following fragment of program:

$$
\begin{array}{l}
fn := 1; \\
\textbf{while } n \neq 0 \textbf{ do} \\
\quad (fn := fn * n; \\
\quad n := n - 1)
\end{array}
$$

Notice that the program has write access to the variable n but that its final value is not constrained by the post-condition – it is, therefore, important that the initial value of n is used in the post-condition.

This informal notion of a piece of program satisfying a specification can be made completely formal: Chapter 10 gives rules for such proofs and extends the notion of satisfaction to cope with designs (or other specifications) satisfying specifications. As an exercise, one can show how the factorial program might be developed from its specification. The overall task ($FACT$) could be decomposed into an initialization ($INIT$) and a loop ($LOOP$); the initialization can be specified:

$$
\begin{array}{l}
INIT \\
\textbf{ext wr } fn : \mathbb{N} \\
\textbf{post } fn = 1
\end{array}
$$

Here, the variable n is not mentioned as an external. Showing which state variables are left unchanged by an operation is sometimes known as the *frame problem*. In this style of specification, a variable which is either not mentioned, or shown as read only, in the operation specification cannot be changed by that operation.

There are many possible specifications for $LOOP$. There is a temptation to use a pre-condition of $fn = 1$ but this is not really required. It is possible to write a more general specification (i.e. one involving fewer assumptions) which specializes to the required effect in the context of the above initialization:

$$
\begin{array}{l}
LOOP \\
\textbf{ext wr } n \ : \mathbb{N}, \\
\qquad \textbf{wr } fn : \mathbb{N} \\
\textbf{post } fn = \overleftarrow{fn} * \overleftarrow{n}!
\end{array}
$$

The techniques in Chapter 10 could be used to prove that the sequential combination of the specifications for *INIT* and *LOOP* satisfy that for *FACT*.

The next step of development would be to decompose *LOOP* into smaller steps. The design might be:

while $n \neq 0$ **do** *BODY*

It is possible to give, to the body of the loop, a specification which does not constrain implementation to the specific two statements used above. What is really required by the loop is that the product of the variable *fn* and the factorial of the value of n remains constant; it is also necessary to avoid the trivial implementation which does nothing: the second conjunct of the post-condition for *BODY* requires that the value of n decreases. In order to ensure that this is possible (given the type of n) the pre-condition is required:

BODY
ext wr $n \ : \mathbb{N}$,
    wr $fn \ : \mathbb{N}$
pre $n > 0$
post $fn * n! = \overleftarrow{fn} * \overleftarrow{n}! \wedge n < \overleftarrow{n}$

One important property of implicit specifications is to avoid implementation commitments. Even on this small example, *BODY* is specified so as to allow different implementations (e.g. n could be decreased by more than 1). However, the two statements in the code above can also be seen to satisfy the specification.

As might be expected, implicit specification brings certain problems – the need to find a suitable pre-condition for *BODY* demonstrates the need for a check that a specification is *satisfiable*. This point is picked up in Section 5.3, where it is treated as a formal proof obligation.

All of the arithmetic examples of the preceding chapter could be rewritten as operations rather than functions but many would be not be instructive. The greatest common divisor problem, however, does exhibit some useful points:

GCD
ext wr $m \ : \mathbb{N}_1$,
    wr $n \ : \mathbb{N}_1$
post $\textit{is-common-divisor}(\overleftarrow{m}, \overleftarrow{n}, m) \wedge$
    $\neg \exists d \in \mathbb{N}_1 \cdot \textit{is-common-divisor}(\overleftarrow{m}, \overleftarrow{n}, d) \wedge d > m$

Notice that the result is left as the final value of m. The usefulness of building up a post-condition from separate conjuncts can again be seen. It is also important to observe the interaction between the external clause and the post-condition: the final value of n is not constrained other than by its type. Any temptation to claim that a specification is inefficient must be resisted. The assertion *post-GCD* could be thought of as implying a massive search in an implementation. The purpose of a specification is to constrain the results; its efficiency should be measured in terms of its ease of comprehension.

The case for implicitly specifying operations, rather than giving their implementations, is loaded heavily towards specification. All of the reasons which make it clearer to use implicit specifications of functions (e.g. range of results, explicit pre-condition) recur. But for operations, there is an additional argument: sequences of statements are not normal mathematical expressions. In general, the equivalence of two such sequences has to be proved by mapping both of them to some common mathematical domain. For this reason, it is far easier to show that a sequence of statements satisfies a specification than it is to show that two sequences of statements compute the same result. Indeed, two programs could both satisfy the same specification but *not* be equivalent!

In this chapter there are several examples where specifications can be made clearer by defining a sequence of functions each in terms of the preceding ones. It is also desirable to structure the specifications of operations. This has been done above by using functions in pre- and post-conditions. It should be obvious that it is not possible to use an operation, as such, in the pre- or post-condition of another operation: these latter are logical expressions and a state-changing operation has no meaning in such a context. This having been said, Section 9.1 introduces a way in which the logical *specification* of one operation can be used in the specification of another.

The specification of non-deterministic operations takes over another technique from function specification and makes the role of post-conditions for operations clear. Loosely specified functions (i.e. those with post-conditions which do not determine a unique result) can only be implemented by deterministic functions. For operations, there is good reason to take the alternative position: implementations of operations can be non-deterministic. In languages with parallelism or explicit non-deterministic constructs, the need for this is obvious. There are also more subtle reasons for permitting the non-deterministic view. All that really need concern the reader for now is that the proof rules presented do cope with this interpretation.

Exercise 3.4.1 Specify an operation which subtracts the initial value of n from m, where both are treated as external variables.

Exercise 3.4.2 Specify an operation which has write access to two variables (say m and n); although both variables can be changed, it is required that the sum of their final values is the same as the sum of their initial values – furthermore, the operation should decrease the value in m. Assume that both variables are natural numbers.

Exercise 3.4.3 Specify the operation of integer division. There are to be three external variables. The initial value (integer) in m is to be divided by the initial value (integer) in n; the quotient is to be put into register q (integer) and the remainder left in m. Make restrictions on m and n to make the task easier. Do not use division or **mod** in the post-condition.

Respecify the operation so that n is a parameter and the quotient is given as output from the operation.

Exercise 3.4.4 Another program for factorial (using a temporary variable t and avoiding changes to n) is:

$fn := 1;$
$t := 0;$
while $t \neq n$ **do**
  $(t := t + 1;$
  $fn := fn * t)$

Sketch (as above) how it might have been developed.

4

Set Notation

> By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race.
> A. N. Whitehead

The numeric data considered so far provides an introduction to the key concepts of specification and proof. However, many systems programs, or commercial applications, do relatively little calculation; the essential purpose of such programs is the manipulation of data structures. Attention is now turned to specification and proof techniques relating to data structures. This, and the next three chapters show how abstraction on data structures plays a crucial part in writing succinct specifications.

Clearly, one would not wish to specify large systems at the bit and byte level. Most high-level programming languages tend to focus on those data structures which can be implemented efficiently: APL provides a rich set of array operations, LISP provides list-processing facilities. In a specification language, it would be a mistake to favour one specific programming language. Rather, a specification language should be rich enough to model a wide range of problems prior to any commitment to a particular programming language.

There is, however, a more important influence on the choice of data types in specification languages. Programming languages implement those structures which can be mapped efficiently onto the target machine. In writing specifications, concern should be focused on the task being specified and not on its eventual implementation. In general, it is possible to achieve concise specifications by using data types, like sets, which are more abstract than those, like arrays, which are governed by implementation efficiency considerations.

Such abstraction is, of course, possible only at the specification level – the eventual implementation must accept the constraints of the implementation machine. (Strictly, one should say here, the constraints of the implementation language. It is, however, true that most languages simply transmit the constraints of the underlying machines.) Data reification is considered in Chapter 8 – this can be seen as a process of making commitments which achieve efficiency by capitalizing on context. In writing specifications, whenever a trade-off between efficiency and clarity has to be made, preference is always given to the latter.

This chapter begins by reviewing set notation informally; Section 4.2 cements the understanding by developing a proof theory; Sections 4.3 and 4.4 develop the use of set notation in specifications.

4.1 Set notation

A spell-checker specification

Later sections show the use of set notation in specifications, but basic set notation should be familiar enough to make a simple introductory example readable. Consider the task of checking a large text file against a dictionary of known words. Such a program is useful in the location of possible spelling errors. There are, of course, many representation details about the text file to be resolved. But the crucial design decisions undoubtedly concern the representation of the dictionary. If this is to store tens of thousands of words in a way which facilitates efficient searching, some ingenuity is required in design – this issue is returned to in Chapter 8 as an example of data reification. Applying the dictum of abstraction, the representation issue can – and should – be postponed. For a specification the only concern is with a finite, unordered collection of distinct words – the state of this system can be presented as:

$$\mathit{Word}\text{-}\mathbf{set}$$

Even *Word* need not be further defined at this point. Apart from the detail that the required notation is not covered until Chapter 7, there is a positive advantage in postponing this implementation-specific information: the specification is thereby made more abstract. The operation which must be invoked once per word in the text should return **true** if, and only if, its argument is a member of the state; its specification is:

$CHECKWORD\ (w: Word)\ b: \mathbb{B}$
ext rd $dict\ :\ Word$-**set**
post $b \ \Leftrightarrow\ w \in dict$

Notice that $w \in dict$ yields a truth value and thus the equality with the Boolean result, b, is defined using the equivalence operator.

The initial state for the system might be the empty set of words:

$dict_0 = \{\,\}$

An operation to add one word to a dictionary can be specified:

$ADDWORD\ (w: Word)$
ext wr $dict\ :\ Word$-**set**
pre $w \notin dict$
post $dict = \overleftarrow{dict} \cup \{w\}$

This specification appears simple precisely because an apposite data type is used in defining its state. In terms of data structures usable in, for example, Pascal, the definition would be far longer and less clear. Of course, such representation details have to be faced in the design process but a concise specification is achieved by postponing implementation details.

It is interesting to note that the pre-condition of $ADDWORD$ is not necessary at the set level of description. It might, however, be important for the implementation and this justifies its being recorded. It can be a mistake to become so involved in the abstraction that the needs of the implementation are entirely ignored. In fact, the crucial decision here is whether or not the user accepts this limitation.

Notation

The specification above has used a little set notation – as have earlier chapters. It is now necessary to examine this notation in more detail. A *set* is an unordered collection of distinct objects; set values are marked by braces, thus:

$\{a, b\} = \{b, a\}$

The fact that the values are distinct means that there is no concept of the number of occurrences of an element in a set – elements are either present ($\in$) or absent ($\notin$). Thus:

$a \in \{a, b\}$
$c \notin \{a, b\}$

Notice that a set containing one element is distinct from that element:

$$\{a\} \neq a$$

The sets above are formed by simple enumeration of their elements; sets can also be defined by *set comprehension* – this defines a set which contains all elements satisfying some property – thus:

$$\{i \in \mathbb{Z} \mid 1 \leq i \leq 3\} = \{1, 2, 3\}$$
$$x \in \{y \in Y \mid p(y)\} \;\Leftrightarrow\; x \in Y \wedge p(x)$$

The need for a set containing an interval of the integers is common enough to justify a special notation:

$$\{i, \ldots, k\} = \{j \in \mathbb{Z} \mid i \leq j \leq k\}$$
$$\{1, \ldots, 3\} = \{1, 2, 3\}$$
$$\{2, \ldots, 2\} = \{2\}$$

But:

$$j < i \;\Rightarrow\; \{i, \ldots, j\} = \{\,\}$$

Where $\{\,\}$ is the empty set.
It is possible to relax the set comprehension notation in the case that types are obvious – write:

$$\{f(i) \mid p(i)\}$$

where f is a total function on D, meaning:

$$x \in \{f(i) \mid p(i)\} \;\Leftrightarrow\; \exists i \in D \cdot p(i) \wedge x = f(i)$$

A way of forming new set types is to use the '-**set**' constructor applied to (the names of) known sets, for example:

$$\mathbb{B}\text{-}\mathbf{set} = \{\{\,\}, \{\mathbf{true}\}, \{\mathbf{false}\}, \{\mathbf{true}, \mathbf{false}\}\}$$

Providing BS is finite:

$$BS\text{-}\mathbf{set} = \{S \mid S \subseteq BS\}$$

The X-**set** constructor yields only finite subsets of its base set[1] but, just as with natural numbers, there can be an infinite set of such subsets. One argument for this restriction is that it is rare, in writing specifications, that infinite sets – as such – are manipulated. Since computer stores are themselves finite it would only

[1] If the base set is infinite, this is not the same as the power set which yields the set of all subsets; for finite base sets, -**set** is identical with power set.

be possible to perform such manipulation indirectly via some finite representation. The restriction to finite values also facilitates inductive proofs (see page 96).

The distinction between sets and their elements is crucial. Notice that X-**set** defines a set of sets. The signature of *maxs* is:

$$maxs: \mathbb{N}\text{-}\mathbf{set} \to \mathbb{N}$$

This function can be applied to the elements of its domain, for example:

$$\{1, 7, 17\} \in \mathbb{N}\text{-}\mathbf{set}$$

It yields an element of its range:

$$17 \in \mathbb{N}$$

The operators which apply to operands which are sets are first discussed by example and logical expressions. Suppose e_1, e_2, etc. are expressions which evaluate to sets, ss evaluates to a set of sets, and:

$$S_1 = \{a, b, c\}$$
$$S_2 = \{c, d\}$$

The *union* of two sets is a set containing the elements of both sets (ignoring which set the elements come from and whether they are present in only one set or both):

$$S_1 \cup S_2 = \{a, b, c, d\}$$

It can be defined:

$$e_1 \cup e_2 = \{x \mid x \in e_1 \vee x \in e_2\}$$

A natural generalization of this operator is the *distributed union* of a set of sets. This unary (prefix) operator yields all of the elements present in any of the sets which are contained in its operand:

$$\bigcup\{S_1, \{e\}, S_2, \{\,\}\} = \{a, b, c, d, e\}$$

It can be defined:

$$\bigcup ss = \{x \mid \exists e \in ss \cdot x \in e\}$$

The *intersection* of two sets is a set which contains those elements common to the two sets:

$$S_1 \cap S_2 = \{c\}$$

It can be defined:

$$e_1 \cap e_2 = \{x \mid x \in e_1 \wedge x \in e_2\}$$

The *difference* of two sets is that set which contains the elements of the first operand that are not present in the second operand:

$$S_1 - S_2 = \{a, b\}$$

It can be defined:

$$e_1 - e_2 = \{x \mid x \in e_1 \wedge x \notin e_2\}$$

The operators above all yield values which are sets. Other operators yield Boolean results and can be used to test for properties of sets. Membership tests are used above:

$$a \in S_1$$
$$d \notin S_1$$

One set is a *subset* of (or is equal to) another if the second operand contains all elements of the first:

$$\{c\} \subseteq S_1$$
$$S_1 \subseteq S_1$$
$$S_1 \subseteq (S_1 \cup S_2)$$
$$\{\,\} \subseteq S_1$$

It can be defined:

$$e_1 \subseteq e_2 \;\Leftrightarrow\; (\forall x \in e_1 \cdot x \in e_2)$$

Unqualified use of the word 'subset' in this book implies that equality is subsumed. *Proper subset* excludes the case of equality:

$$\{\,\} \subset S_1$$
$$\{a, b\} \subset S_1$$
$$\neg\,(S_1 \subset S_1)$$

It can be defined:

$$e_1 \subset e_2 \;\Leftrightarrow\; e_1 \subseteq e_2 \wedge \neg\,(e_2 \subseteq e_1)$$

Set equality can be defined:

$$e_1 = e_2 \;\Leftrightarrow\; e_1 \subseteq e_2 \wedge e_2 \subseteq e_1$$

These operators are analogous to the ordering operators on numbers ($\leq$, $<$ and $=$). The subset operator is not, however, total: there are S_i and S_j such that:

$$\neg\,(S_i \subseteq S_j \vee S_j \subseteq S_i)$$

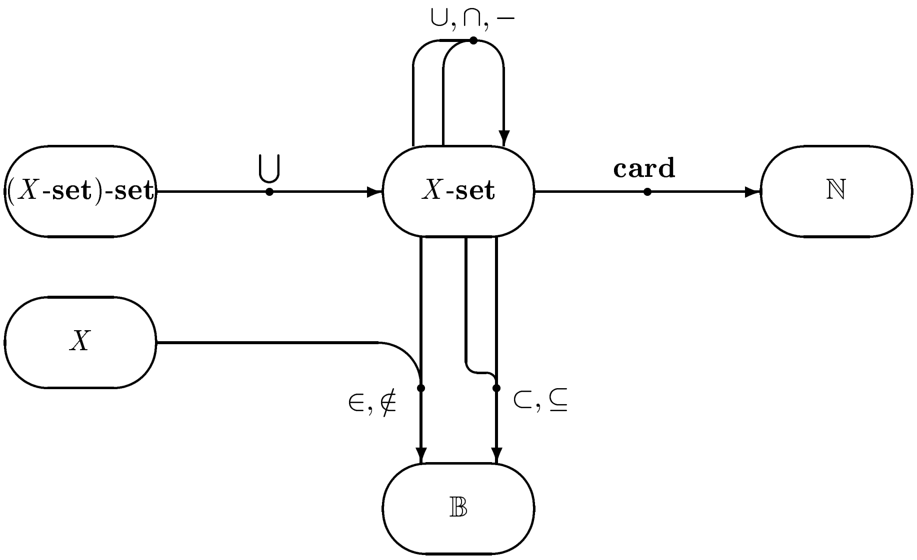

Figure 4.1 ADJ diagram of set operators

The *cardinality* of a (finite) set is the number of elements in the set:

$\mathbf{card}\, S_1 = 3$
$\mathbf{card}\, S_2 = 2$
$\mathbf{card}\, \{\,\} = 0$

A group of computer scientists who investigated an algebraic view of data types dubbed themselves the 'ADJ group'. They used a graphical notation for describing the signatures of operators and an *ADJ diagram* of the set operators is shown in Figure 4.1. In such diagrams, the ovals denote data types; the arcs from ovals to operators show the types of the operands; and those arcs from operators to ovals show the type of the result. Thus Figure 4.1 shows that $\in$ has the type $X \times X\text{-}\mathbf{set} \to \mathbb{B}$.

Priorities are placed on the logical operators in order to minimize the parentheses required in complex expressions. There is an obvious argument for mirroring the priority of $\wedge/\vee$ by making $\cap$ higher priority than $\cup$ – thus:

$S_1 \cap S_2 \cup S_3$

means:

$$(S_1 \cap S_2) \cup S_3$$

There is less agreement in textbooks about what should be done beyond this. The problem, which can be seen on the ADJ diagram, is that the operators yield results of different types. In general below, parentheses are used to make expressions like:

$$(A \cup B) \subseteq C$$
$$x \in (A \cup B)$$

clear. The set (or arithmetic) operators are assumed to be of higher priority than the logical operators.

Exercise 4.1.1 Write down the values of:

$$\{a, c\} \cap \{c, d, a\}$$
$$\{a, c\} - \{c, d, a\}$$
$$\mathbf{card}\, \{x^2 \mid x \in \{-1, \ldots, +1\}\}$$
$$5 \in \{3, \ldots, 7\}$$
$$\{7, \ldots, 3\}$$

$$\{i \in \mathbb{N} \mid i^2 \in \{4, 9\}\}$$
$$\{i \in \mathbb{Z} \mid i^2 = i\}$$
$$\bigcup \{\{a, b\}, \{\,\}, \{b, c\}, \{d\}\}$$
$$\bigcup \{\,\}$$

Exercise 4.1.2 Write set comprehension expressions for:

- the set of integers greater than 100 and less than 200 which are exactly divisible by 9;
- the set of prime numbers in the same range.

Show the subset relationships between $\mathbb{N}$, $\mathbb{Z}$ and $\mathbb{N}_1$.

Exercise 4.1.3 Complete the following by replacing the question mark so as to generate a true statement about sets (assume that the types are sensible):

$$e \cup e = ?$$
$$e \cap \{\,\} = ?$$
$$(e_1 \subseteq e_2) \;\Leftrightarrow\; (e_1 - e_2 = ?)$$
$$e \cap e = ?$$
$$e \cup \{\,\} = ?$$
$$e_1 \subseteq e_2 \land e_2 \subseteq e_3 \;\Rightarrow\; e_1 \; ? \; e_3$$
$$\{\,\} \; ? \; e$$

$$\mathbf{card}\,(e_1 \mathbin{?} e_2) = \mathbf{card}\; e_1 + \mathbf{card}\; e_2 - \mathbf{card}\,(e_1 \cap e_2)$$
$$(e_1 - e_2) \cap e_3 = (e_1 \mathbin{?} e_3) - e_2$$
$$e_1 - (e_1 - e_2) = e_1 \mathbin{?} e_2$$
$$\bigcup\{\bigcup es\} = \,?\; es$$

Exercise 4.1.4 Write out the commutative and associative laws for intersection; and the distributive laws for intersection over union.

Exercise 4.1.5 Define a predicate:

$$is\text{-}disj : X\text{-}\mathbf{set} \times X\text{-}\mathbf{set} \to \mathbb{B}$$
$$is\text{-}disj(s_1, s_2) \quad \triangleq \quad \cdots$$

which yields **true** if, and only if, the two sets have no common elements (i.e. they are disjoint).

Exercise 4.1.6 Define a distributed intersection operator – is a pre-condition required?

Exercise 4.1.7 (*) A symmetric difference operator can be defined:

$$s_1 \ominus s_2 = (s_1 \cup s_2) - (s_1 \cap s_2)$$

Complete the following expressions so that they are true:

$$s_1 \ominus s_2 = \{\,\} \;\Rightarrow\; s_1 \mathbin{?} s_2$$
$$s_1 \ominus s_1 = \,?$$
$$s_1 \mathbin{?} s_2 \subseteq s_1 \ominus s_2$$
$$s_1 \ominus s_2 = s_2 \mathbin{?} s_1$$
$$s_1 \ominus s_2 = (s_1 - s_2) \mathbin{?} (s_2 - s_1)$$
$$s_1 \ominus (s_1 \ominus s_2) = \,?$$

4.2 Reasoning about sets

Given the intuitive understanding of set operators from the preceding section, the next step is to be able to construct proofs about sets. As this section progresses, the proofs begin to contain less formal detail than in Section 3.2: the proofs are rigorous without being completely formal.

Induction based on set generators

Inductive proofs about the natural numbers are based on the generators (i.e. 0 and *succ*). Proofs about finite sets can be based on very similar inductive rules.[2] Here again the crucial step is to recognize the generators for sets – these are the empty set ($\{\,\}$) and an (infix) insertion operator ($\odot$) which adds an element to a set (its type is $X \times X\textbf{-set} \to X\textbf{-set}$). This insertion operator is *only* used in the construction of, and proofs about, the inductive structure of sets – one would normally use set union with a unit set. The intuition behind these generators is that any finite set can be represented by an expression of the form:

$$e_1 \odot (e_2 \odot (\ldots \odot \{\,\}))$$

Axiom 4.1 The fact that the elements of a set are unordered is reflected by the following commutativity property of the insertion operator:

$$\boxed{\odot\textit{-comm}}\ \frac{e_1, e_2 \in X;\ s \in X\textbf{-set}}{e_1 \odot (e_2 \odot s) = e_2 \odot (e_1 \odot s)}$$

Axiom 4.2 Similarly, the fact that sets do not contain duplicate elements is reflected by its property of absorption:

$$\boxed{\odot\textit{-abs}}\ \frac{e \in X;\ s \in X\textbf{-set}}{e \odot (e \odot s) = e \odot s}$$

Notice that these two properties imply that the intuitive representations of sets are not unique: syntactically different expressions (e.g. $e_1 \odot (e_2 \odot (e_2 \odot \{\,\}))$, $e_2 \odot (e_1 \odot \{\,\})$) stand for the same set value.

Axiom 4.3 The set induction rule which is suggested by the generators[3] is:

$$\boxed{\textit{Set-ind}}\ \frac{\begin{array}{l} p(\{\,\}); \\ e \in X,\ s \in X\textbf{-set},\ p(s) \vdash p(e \odot s) \end{array}}{s \in X\textbf{-set} \vdash p(s)}$$

This leads to proofs of the same shape as with $\mathbb{N}$-*ind* and it again relies on the finiteness of the possible values. As with the natural numbers, set operators can be defined by recursive functions over the generators. One way of making proofs about sets less tedious than those about natural numbers is to give the

[2] It is possible to prove many properties of sets by induction on their cardinality. This reduces induction on sets to induction on the natural numbers. But the consistent approach of studying the generators for each data type results in clearer proofs.

[3] This rule could be strengthened by adding $e \notin s$ as a hypothesis to the induction step; this is not done here since it is covered by the alternative rule given on page 102.

$$
\begin{array}{lll}
\textbf{from } s \in X\textbf{-set} & & \\
1 \quad \{\,\} \cup \{\,\} = \{\,\} & & \cup\text{-}b \\
2 \quad \textbf{from } e \in X,\ s_1 \in X\textbf{-set},\ s_1 \cup \{\,\} = s_1 & & \\
2.1 \qquad (e \odot s_1) \cup \{\,\} = e \odot (s_1 \cup \{\,\}) & & \cup\text{-}i \\
\quad\ \ \textbf{infer } (e \odot s_1) \cup \{\,\} = e \odot s_1 & & =\text{-}subs(\text{ih}2,2.1) \\
\textbf{infer } s \cup \{\,\} = s & & Set\text{-}ind(1,2)
\end{array}
$$

Lemma 4.5

information about operators directly in terms of inference rules. Thus, for $\cup$[4] the rules are as follows.

Rule 4.4 A basis ($\cup$-b) and an inductive rule ($\cup$-i) are given:

$$\boxed{\cup\text{-}b}\ \frac{s \in X\textbf{-set}}{\{\,\} \cup s = s}$$

$$\boxed{\cup\text{-}i}\ \frac{e \in X;\ s_1, s_2 \in X\textbf{-set}}{(e \odot s_1) \cup s_2 = e \odot (s_1 \cup s_2)}$$

Inductive proofs of set properties

Lemma 4.5 The rule $\cup$-b shows that $\cup$ absorbs empty sets as left operand; a proof must be given that the same happens on the right:

$$\boxed{\text{L4.5}}\ \frac{s \in X\textbf{-set}}{s \cup \{\,\} = s}$$

Lemma 4.6 ($\cup$-ass) Set union is associative:

$$\boxed{\cup\text{-}ass}\ \frac{s_1, s_2, s_3 \in X\textbf{-set}}{(s_1 \cup s_2) \cup s_3 = s_1 \cup (s_2 \cup s_3)}$$

Lemma 4.7 ($\cup$-$comm$) Set union is commutative:

[4] Strictly, the set operators are parameterized on the type of the set elements – this point is not treated formally here.

from $s_1, s_2, s_3 \in X$-**set**		
1	$(\{\,\} \cup s_2) \cup s_3$	
	$= s_2 \cup s_3$	$\cup$-b
2	$= \{\,\} \cup (s_2 \cup s_3)$	$\cup$-b
3	**from** $e \in X,\ s \in X$-**set**, $(s \cup s_2) \cup s_3 = s \cup (s_2 \cup s_3)$	
3.1	$((e \odot s) \cup s_2) \cup s_3$	
	$= (e \odot (s \cup s_2)) \cup s_3$	$\cup$-i
3.2	$= e \odot ((s \cup s_2) \cup s_3)$	$\cup$-i
3.3	$= e \odot (s \cup (s_2 \cup s_3))$	ih2
	infer $= (e \odot s) \cup (s_2 \cup s_3)$	$\cup$-i
infer $(s_1 \cup s_2) \cup s_3 = s_1 \cup (s_2 \cup s_3)$		*Set-ind*(2,3)

Lemma 4.6: $\cup$-*ass*

$$\boxed{\cup\text{-}comm}\ \frac{s_1, s_2 \in X\text{-}\mathbf{set}}{s_1 \cup s_2 = s_2 \cup s_1}$$

Lemma 4.8 ($\cup$-*idem*) Set union is idempotent:

$$\boxed{\cup\text{-}idem}\ \frac{s \in X\text{-}\mathbf{set}}{s \cup s = s}$$

Lemma 4.9 Distributed union distributes over union as follows:

$$\boxed{\text{L4.9}}\ \frac{ss_1, ss_2 \in (X\text{-}\mathbf{set})\text{-}\mathbf{set}}{\bigcup(ss_1 \cup ss_2) = \bigcup ss_1 \cup \bigcup ss_2}$$

The proof for Lemma 4.5 is straightforward and is given on page 97. Detail is omitted in these proofs by abbreviating the justifications: references to lines which provide type information, are dropped. Clearly, the writer of a proof should have checked the steps and a reviewer who is in doubt can ask for the details to be provided. Just as with the proofs in Section 3.2, the presentation here is given in the order for reading. This proof is actually best found by writing:

- the outer **from/infer**;
- line 1 and the inner **from/infer** (2) are generated by the induction rule; this now permits the final justification to be given;
- the justification of line 1;

- completion of the inner **from/infer**.

This way of generating proofs is quite general but it is not always immediately obvious which variable to use in the induction. A proof that *union* is associative (Lemma 4.6) is given (by induction on s_1) on page 98. Here again, the induction rule has been used to generate the sub-goals (1 and 3). What is more difficult in this proof is to choose the variable over which induction is to be performed. Often it is necessary to make a few experiments before it becomes clear which of the possible choices best decomposes the proof task. In addition to not referring to all of the type assumptions, another way of shortening proofs is used here. It is common in reasoning about data types to need many steps of substitution of equal expressions ($=$-*subs*). In the proof of Lemma 4.5 this is shown explicitly. Here, lines 1 and 2 follow by $=$-*subs* but only the subsidiary equality is cited. Lines 3.1 to the conclusion of the inner box represent another chain of equalities.

Using the commutative property of $\odot$, it is possible to prove that *union* is commutative (Lemma 4.7). A preliminary lemma and the main proof are given on page 100. The separation of the lemma avoids the need for a nested induction. The idempotence of union (Lemma 4.8), which relies on the absorptive property of $\odot$, is proved on page 101.

Exercise 4.2.1 Define (over the generators – as with union above) set intersection and prove:

$$s \cap \{\,\} = \{\,\}$$

Also prove its associativity, commutativity, and idempotence as well as the distribution of union over intersection and *vice versa*.

Exercise 4.2.2 Define the distributed union operator and prove:

$$\bigcup\{\bigcup ss\} = \bigcup ss$$

Also prove Lemma 4.9.

Exercise 4.2.3 Define set difference and prove:

$$(S_1 - S_2) \cap S_3 = (S_1 \cap S_3) - S_2$$

Exercise 4.2.4 (*) Define and develop a useful theory of the symmetric difference operator for sets (cf. Exercise 4.1.7 on page 95).

Exercise 4.2.5 (*) Exercise 2.1.6 on page 34 discusses the idea of reasoning about types. Rather than give signatures for the derived (set) operators, it would be possible to infer their types from the rules of generation. Based on $\cup$-b and $\cup$-i, infer that

$$
\begin{array}{lll}
\textbf{from } e \in X,\ s_1, s_2 \in X\textbf{-set} & & \\
1 \quad e \odot (\{\,\} \cup s_2) & & \\
\qquad = e \odot s_2 & & \cup\text{-}b \\
2 \qquad = \{\,\} \cup (e \odot s_2) & & \cup\text{-}b \\
3 \quad \textbf{from } e_2 \in X,\ s \in X\textbf{-set},\ e \odot (s \cup s_2) = s \cup (e \odot s_2) & & \\
3.1 \qquad e \odot ((e_2 \odot s) \cup s_2) & & \\
\qquad\quad = e \odot (e_2 \odot (s \cup s_2)) & & \cup\text{-}i \\
3.2 \qquad\quad = e_2 \odot (e \odot (s \cup s_2)) & & \odot\text{-}comm \\
3.3 \qquad\quad = e_2 \odot (s \cup (e \odot s_2)) & & \text{ih3} \\
\quad \textbf{infer} \quad = (e_2 \odot s) \cup (e \odot s_2) & & \cup\text{-}i \\
\textbf{infer } e \odot (s_1 \cup s_2) = s_1 \cup (e \odot s_2) & & Set\text{-}ind(2,3)
\end{array}
$$

$$
\begin{array}{lll}
\textbf{from } s_1, s_2 \in X\textbf{-set} & & \\
1 \quad \{\,\} \cup s_2 = s_2 \cup \{\,\} & & \cup\text{-}b,\ \text{L4.5} \\
2 \quad \textbf{from } e \in X,\ s \in X\textbf{-set},\ s \cup s_2 = s_2 \cup s & & \\
2.1 \qquad (e \odot s) \cup s_2 & & \\
\qquad\quad = e \odot (s \cup s_2) & & \cup\text{-}i \\
2.2 \qquad\quad = e \odot (s_2 \cup s) & & \text{ih2} \\
\quad \textbf{infer} \quad = s_2 \cup (e \odot s) & & \text{Lemma} \\
\textbf{infer } s_1 \cup s_2 = s_2 \cup s_1 & & Set\text{-}ind(1,2)
\end{array}
$$

Lemma 4.7: $\cup$-*comm*

$$\boxed{\cup\text{-sig}}\ \frac{s_1, s_2 \in X\textbf{-set}}{(s_1 \cup s_2) \in X\textbf{-set}}$$

Proofs about membership

It is possible to characterize the set membership operator by inference rules and thus provide the basis for formal proofs which include this operator. The basic facts about membership are:

$$\boxed{\in\text{-b}}\ \frac{}{\neg\,\exists e \in X \cdot e \in \{\,\}}$$

from $s \in X$-**set**		
1	$\{\,\} \cup \{\,\} = \{\,\}$	$\cup$-*b*
2	**from** $e \in X,\ s \in X$-**set**$,\ s \cup s = s$	
2.1	$(e \odot s) \cup (e \odot s)$	
	$= e \odot (s \cup (e \odot s))$	$\cup$-*i*
2.2	$= e \odot ((e \odot s) \cup s)$	$\cup$-*comm*
2.3	$= e \odot (e \odot (s \cup s))$	$\cup$-*i*
2.4	$= e \odot (s \cup s)$	$\odot$-*abs*
	infer $= e \odot s$	ih2
infer $s \cup s = s$		*Set-ind*(1,2)

Lemma 4.8: $\cup$-*idem*

$$\boxed{\in\text{-i}}\ \frac{e_1, e_2 \in X,\ s \in X\text{-}\mathbf{set}}{e_1 \in (e_2 \odot s) \;\Leftrightarrow\; e_1 = e_2 \vee e_1 \in s}$$

It is now possible to prove properties like:

Lemma 4.10

$$\boxed{\text{L4.10}}\ \frac{x \in (s_1 \cup s_2)}{x \in s_1 \vee x \in s_2}$$

Below, it is necessary to prove properties of the form:

$$\forall x \in \{x \in X \mid p(x)\} \cdot q(x)$$

It should be clear that this is equivalent to:

$$\forall x \in X \cdot p(x) \;\Rightarrow\; q(x)$$

Similarly:

$$\exists x \in \{x \in X \mid p(x)\} \cdot q(x)$$

is equivalent to:

$$\exists x \in X \cdot p(x) \wedge q(x)$$

With the natural numbers, a second form of the induction rule is available once subtraction has been introduced ($\mathbb{N}$-*indp*). The rule has an inductive step which shows that p inherits from $n-1$ to n. It is not the intention here to develop the whole of the set notation formally, but – once set difference has been covered – the following induction rule can be used.

Axiom 4.11

$$\boxed{\textit{Set-ind2}}\ \frac{p(\{\,\});\quad s \in X\text{-}\mathbf{set},\ e \in s,\ p(s - \{e\})\ \vdash\ p(s)}{s \in X\text{-}\mathbf{set}\ \vdash\ p(s)}$$

Notice that the validity of this rule relies on $\odot$-*abs* and $\odot$-*comm*. It would also be possible to present a complete induction rule for sets.

4.3 Theories of data types

Importance of theories

The preceding section has established a theory of sets which can be used throughout the remainder of this book. Whenever a new class of objects arises, it is worth investigating its properties. In effect, a *theory* of the new objects is created which gathers together useful results about the objects. Of course, for the well-known basic types like sets, standard mathematical texts may be consulted. The advantage of building such a theory for other types, as they arise, is that the collection of results is then available for any use of that type. Several authors (including [Jon79]) have recognized the crucial role that the development of theories will play in making more widespread the use of formal methods.

Partitions

As an example of such a theory, this section outlines some results about the concept of *Partition*. This theory is used in a specification in the next section; there, a motivation for the specific example is given. In this section, the theory is developed abstractly. If this makes the material too difficult to absorb, the reader should skim it now and then return when the results are needed in Section 4.4.

A set (of, say, $\mathbb{N}$) is *partitioned* if it is split into (a set of) disjoint subsets. Thus:

$$\textit{Partition} = \{\, p \in (\mathbb{N}\text{-}\mathbf{set})\text{-}\mathbf{set} \mid \textit{inv-Partition}(p)\}$$

Where:

$$\begin{array}{l} \textit{inv-Partition} : (\mathbb{N}\text{-}\mathbf{set})\text{-}\mathbf{set} \to \mathbb{B} \\ \textit{inv-Partition}(p) \quad \triangleq \quad \textit{is-prdisj}(p) \wedge \{\,\} \notin p \end{array}$$

Pairwise disjointness is defined by:

$$\begin{array}{l} \textit{is-prdisj} : (\mathbb{N}\textbf{-set})\textbf{-set} \to \mathbb{B} \\ \textit{is-prdisj}(ss) \quad \triangleq \quad \forall s_1, s_2 \in ss \cdot s_1 = s_2 \vee \textit{is-disj}(s_1, s_2) \end{array}$$

(A full discussion of, and notation for, such data type invariants is contained in Section 5.2. The exclusion of the empty set is a technicality which is explained in Chapter 11: for now, it should just be accepted.)

An example of a *Partition* is:

$$\{\{3,6\},\{5\},\{1,2,7\}\}$$

Notice that elements of *Partition* are sets of sets – the collection of all partitions is, of course, a set of such objects. Thus:

$$\begin{array}{l} \{p_a, p_b\} \subseteq \textit{Partition} \\ p_a = \{\{1\},\{2\}\} \\ p_b = \{\{1,2\}\} \end{array}$$

In p_a, which is a 'fine' partition, each element is in a unit set; in the 'coarse' p_b, all elements are in the same set. But:

$$\{\{1,2\},\{1\}\} \in (\mathbb{N}\textbf{-set})\textbf{-set}$$

is not a *Partition* because it fails to satisfy *inv-Partition*.

Given the definition of *Partition* it is possible to prove that certain properties hold.

Lemma 4.12 The trivial empty partition satisfies the invariant:

$$\boxed{\text{L4.12}}\ \frac{}{\{\,\} \in \textit{Partition}}$$

Lemma 4.13 A simple way of extending partitions is given by:

$$\boxed{\text{L4.13}}\ \frac{p \in \textit{Partition};\ e \in \mathbb{N};\ e \notin \bigcup p}{(p \cup \{\{e\}\}) \in \textit{Partition}}$$

Although these results might appear obvious, it is interesting to see how their proofs can be formalized. Both boxed proofs follow the same pattern: firstly the type of the required expression is established; then it is shown that the expression satisfies each clause of *inv-Partition*. Notice, in the proof of Lemma 4.13 on page 105, how lines 2 and 3 establish the need for the double set of braces around the element e. In the same proof, one can observe how properties of the more basic data types are brought into play. Line 9 for example relies on the property of *Set* theory that:

$$\frac{e \notin s_1;\ e \notin s_2}{e \notin (s_1 \cup s_2)}$$

While line 5 uses:

$$\frac{e \notin \bigcup p}{\forall s \in p \cdot \textit{is-disj}(\{e\}, s)}$$

In a mechanized theorem proving system each of these properties would be spelled out.

Partitions can be generated from one another by merging sets which satisfy truth-valued functions:

$$\begin{array}{l} \textit{merge} : \textit{Partition} \times (\mathbb{N}\text{-}\mathbf{set} \to \mathbb{B}) \to \textit{Partition} \\ \textit{merge}(p, t) \quad \triangleq \quad \{s \in p \mid \neg\, t(s)\} \cup \{\bigcup\{s \in p \mid t(s)\}\} \end{array}$$

So, for example, if:

$$t(s) \quad \triangleq \quad \neg\, \textit{is-disj}(s, \{2, 3\})$$

then:

$$\begin{array}{l} \textit{merge}(\{\{1,2,7\},\{5\},\{6,3\}\}, t) \\ \qquad = \{\{5\}\} \cup \{\bigcup\{\{1,2,7\},\{6,3\}\}\} \\ \qquad = \{\{5\}\} \cup \{\{1,2,3,6,7\}\} \\ \qquad = \{\{5\},\{1,2,3,6,7\}\} \end{array}$$

In order to know that this works in general, it is necessary to show that the following lemma holds.

Lemma 4.14 Merging preserves the property of being a *Partition*:

$$\boxed{\text{L4.14}}\ \frac{p \in \textit{Partition};\ (t \colon \mathbb{N}\text{-}\mathbf{set} \to \mathbb{B});\ \exists s \in p \cdot t(s);\ p' = \textit{merge}(p, t)}{p' \in \textit{Partition}}$$

Notice that the third hypothesis avoids the danger of generating an empty set in p'. No proof of this is given here but a closely related proof (Lemma 11.1) is given on page 273.

It would be dishonest to camouflage the fact that this 'theory' was actually extracted from an initial attempt at the specification of the equivalence relation problem which is discussed in the next section. This admission does not undermine the arguments for collecting together such bodies of knowledge. Only when extensive collections are available will it be reasonable to expect that new problems will be encountered which gain major support from what others have done.

Exercise 4.3.1 (*) This exercise concerns the theory *Partition*.

from	*defns*	
1	$\{\,\} \in (\mathbb{N}\text{-}\mathbf{set})\text{-}\mathbf{set}$	*Set*
2	$\forall s_1, s_2 \in \{\,\} \cdot s_1 = s_2 \vee \textit{is-disj}\,(s_1, s_2)$	$\forall$
3	$\textit{is-prdisj}\,(\{\,\})$	*is-prdisj*,2
4	$\forall e \in \mathbb{N}\text{-}\mathbf{set} \cdot e \notin \{\,\}$	$\{\,\}$
5	$\{\,\} \in \mathbb{N}\text{-}\mathbf{set}$	*Set*
6	$\{\,\} \notin \{\,\}$	$\forall$-E(4,5)
infer	$\{\,\} \in \textit{Partition}$	*Partition*,1,3,6

Lemma 4.12

from	$p \in \textit{Partition},\ e \in \mathbb{N},\ e \notin \bigcup p$	
1	$p \in (\mathbb{N}\text{-}\mathbf{set})\text{-}\mathbf{set}$	h,*Partition*
2	$\{\{e\}\} \in (\mathbb{N}\text{-}\mathbf{set})\text{-}\mathbf{set}$	h,*Set*
3	$(p \cup \{\{e\}\}) \in (\mathbb{N}\text{-}\mathbf{set})\text{-}\mathbf{set}$	1,2,$\cup$
4	$\textit{is-prdisj}\,(p)$	h,*Partition*
5	$\forall s \in p \cdot \textit{is-disj}\,(\{e\}, s)$	h,*Set*
6	$\textit{is-prdisj}\,(p \cup \{\{e\}\})$	4,5,*is-prdisj*
7	$\{\,\} \notin p$	h,*Partition*
8	$\{\,\} \notin \{\{e\}\}$	*Set*
9	$\{\,\} \notin (p \cup \{\{e\}\})$	7,8,*Set*
infer	$(p \cup \{\{e\}\}) \in \textit{Partition}$	*Partition*,3,6,9

Lemma 4.13

- Specify a function which, given a set of objects from $\mathbb{N}$, will return a set containing a partition of the input set into two sets whose sizes differ by at most one.

- Show that the coarsest partition of any finite subset of $\mathbb{N}$ satisfies *inv-Partition*.

- Argue informally that $\{\,\} \notin p \wedge \forall x \in \bigcup p \cdot \exists!\, s \in p \cdot x \in s$ is an equivalent formulation of *inv-Partition*.

- Define a function which can split sets (with two or more elements) of a *Partition* and show that it preserves *inv-Partition*.

4.4 Specifications

The reader should now have a thorough grasp of set notation and some facility with its manipulation in proofs. It would be worth looking back at the specification of the spelling checker in Section 4.1 to ensure that its details are fully understood.

A buffer pool

Another simple specification which uses only sets is for a resource manager program. Suppose that the resource is a pool of buffers. Each buffer might be identified by a buffer identifier which could, in the actual implementation, be an address. This level of detail need not be decided in the initial specification and the buffer identifiers are shown as a set *Bid*. Again, in the likely representation, the free buffers might be organized into a free list. The specification can ignore such representation details and build around an unused set (*us*). An operation which resets the collection of free buffers is:

$SETUP\ (s{:}\, Bid\text{-}\mathbf{set})$
$\mathbf{ext\ wr}\ us\ :\ Bid\text{-}\mathbf{set}$
$\mathbf{post}\ us = s$

A free buffer can be obtained by the operation:

$OBTAIN\ ()\ r{:}\, Bid$
$\mathbf{ext\ wr}\ us\ :\ Bid\text{-}\mathbf{set}$
$\mathbf{pre}\ us \neq \{\,\}$
$\mathbf{post}\ r \in \overleftarrow{us} \wedge us = \overleftarrow{us} - \{r\}$

Notice that this post-condition does not determine which buffer is to be allocated: the specification is non-deterministic. The operation which releases a buffer is:

$RELEASE\ (b{:}\, Bid)$
ext wr $us\ :\ Bid$-**set**
pre $b \notin us$
post $us = \overleftarrow{us} \cup \{b\}$

Census data base

This example illustrates how properties of operations are important in understanding specifications. A database is to be set up which records people's sex and marital status. One possible way of modelling the information is to have three sets: one each for male, female and married names. (*Name* is used as a primitive set – in a real system some form of unique identifier would be used. Thus, no name change is shown on marriage.) In the initial state, all three sets would be empty. One interrogation operation, and two which update the database, are specified:

$MARMALE\ ()\ rs{:}\, Name$-**set**
ext rd $male\ :\ Name$-**set**,
  rd $married\ :\ Name$-**set**
post $rs = male \cap married$

$NEWFEM\ (f{:}\, Name)$
ext wr $female\ :\ Name$-**set**,
  rd $male\ :\ Name$-**set**
pre $f \notin (female \cup male)$
post $female = \overleftarrow{female} \cup \{f\}$

$MARRIAGE\ (m{:}\, Name, f{:}\, Name)$
ext rd $male\ :\ Name$-**set**,
  rd $female\ :\ Name$-**set**,
  wr $married\ :\ Name$-**set**
pre $m \in (male - married) \wedge f \in (female - married)$
post $married = \overleftarrow{married} \cup \{m, f\}$

In each of these operations, external variables are marked as 'read only' where they cannot be changed.

There are certain properties of the operations in this model. For example, the *married* set is always a subset of the union of the other two sets – the *male* and *female* sets are always disjoint. Such properties are *invariants* on the state and are discussed in Section 5.2.

Another point which is taken up in subsequent chapters is the choice of the most appropriate model for a particular specification. That given above, for example, is chosen for pedagogic reasons – the notation of Chapter 6 makes it possible to provide a model with simpler invariants. Even with the set notation alone, other models could be employed – one such is suggested in Exercise 4.4.4 below.

Exercise 4.4.1 The spell checking program of Section 4.1 would probably need an operation which inserted many words into a dictionary at once. Specify an operation which takes a set of words as arguments, adds all new ones to the dictionary and returns all duplicates as result.

Exercise 4.4.2 A system is to be specified which keeps track of which people are in a secure area – ignore how the operations are invoked (perhaps via a badge reader?) and assume that no two people have the same name. Specify operations for *ENTER*, *EXIT*, *ISPRESENT*. Also show the initial state.

Exercise 4.4.3 A system is to be specified which keeps track of which students have done an example class. Specify operations which can be used to:

- record the enrollment of a student (only enrolled students can have the next operation performed);
- record the fact that a student has successfully completed the examples;
- output the names of those students who have, so far, completed the examples.

Also show the initial state.

Exercise 4.4.4 Respecify the three operations in the text relating to the recording of people based on a model:

$singfem: Name\text{-}\mathbf{set}$
$marfem: Name\text{-}\mathbf{set}$
$singmale: Name\text{-}\mathbf{set}$
$marmale: Name\text{-}\mathbf{set}$

What invariants hold over these sets?

Recording equivalence relations

An interesting example which can be handled with sets alone concerns the creation and interrogation of a database which records equivalence relations. Before discussing the specification, some motivation is offered. Compilers for high-level languages of the ALGOL family frequently have to map programs with many variables onto machines in which some store access times (e.g. for registers) are much faster than others. Storing variables in registers can considerably improve the performance of the created object programs especially if they are used to index arrays. There is, however, a trap which must be carefully avoided. Distinct variable names can be made to refer to the same location in store. This happens when variables are passed by location in Pascal (i.e. to *var* parameters) or 'by name' in ALGOL. Any change made to one variable must be reflected in that variable's surrogates. A compiler writer therefore might need to keep track of a relation between variables which might be known as 'could share storage' and to ensure that appropriate register-to-store operations follow updates. The use of 'could' indicates that this check should be fail-safe. Now, if both variable pairs (A and B) and (B and C) could share storage then clearly (A and C) could also share storage. This is one of the properties of an equivalence relation.

The form of relation being considered here records connections over elements of a set.[5] If R is a relation, xRy can be written to state that the pair of elements (x and y) stand in the relation.[6] There are a number of properties which are, or are not, possessed by different kinds of relations. A relation R is said to be *transitive* if when xRy and yRz, then xRz necessarily holds. Figure 4.2 shows which relations over the integers possess the properties being discussed; the reader should use these to confirm the intuition of the properties (note, in particular, that inequality is not transitive). A relation R is *symmetric* if whenever xRy, then yRx. A relation R is *reflexive* if for all elements x, then xRx. A relation is an *equivalence relation* if it is reflexive, symmetric and transitive. Referring to Figure 4.2, it can be seen that equality is the only equivalence relation shown there. The reader should be able to see that the 'could share storage' relation over variables is an equivalence relation.

There are very many applications of such relations in computing including, for example, codebreaking. The applications in graph processing involve relations over very large sets. (The compiler example might involve relatively small sets.) The reader might like to spend some time thinking about how to represent the relation so that it can be queried and updated efficiently. But for now, the real concern is to obtain a clear specification which defines exactly what the system does without getting involved in the implementation problems. The key to such a specification is to use a state containing a *Partition* (named p). The initial value of this variable stores no elements: $p_0 = \{\,\}$. The fact that p_0 is a *Partition*

[5] Mathematically, such a relation is a subset of the Cartesian product of two instances of the set.

[6] Other notational styles for stating this include $(x, y) \in R$ and $R: x \mapsto y$.

Property	Definition	Examples
Reflexive	xRx	$=, \leq, \geq$
Symmetric	$xRy \;\Rightarrow\; yRx$	$=, \neq$
Transitive	$xRy \wedge yRz \;\Rightarrow\; xRz$	$=, <, \leq, >, \geq$

Figure 4.2 Properties of relations over integers

is the import of Lemma 4.12 on page 103. An operation which gives as its result the set of elements which currently occur in any equivalence group is:

$ELS\ ()\ r{:}\,\mathbb{N}\text{-}\mathbf{set}$
$\mathbf{ext\ rd}\ p\ :\ Partition$
$\mathbf{post}\ r = \bigcup p$

Notice that this operation only has read access to p. Its satisfiability therefore relies only on the types matching in the post-condition of ELS: since $\bigcup p$ does yield a $\mathbb{N}$-**set** the operation is satisfiable. A simple state changing operation is one which adds an element as an isolated equivalence group:

$ADD\ (e{:}\,\mathbb{N})$
$\mathbf{ext\ wr}\ p\ :\ Partition$
$\mathbf{pre}\ e \notin \bigcup p$
$\mathbf{post}\ p = \overleftarrow{p} \cup \{\{e\}\}$

Here the satisfiability consideration is less obvious. To know that the combination of the type information ($Partition$) and the post-condition for ADD do not contradict, needs the result in Lemma 4.13 on page 103 from which it follows that:

$$\forall e \in \mathbb{N}, \overleftarrow{p} \in Partition \cdot$$
$$\quad pre\text{-}ADD(e, \overleftarrow{p}) \;\Rightarrow\; \exists p \in Partition \cdot post\text{-}ADD(e, \overleftarrow{p}, p)$$

Another operation which only has read access to p shows the equivalent elements to any given element:

$GROUP\ (e{:}\,\mathbb{N})\ r{:}\,\mathbb{N}\text{-}\mathbf{set}$
$\mathbf{ext\ rd}\ p\ :\ Partition$
$\mathbf{pre}\ e \in \bigcup p$
$\mathbf{post}\ r \in p \wedge e \in r$

Its satisfiability proof obligation is:

$$\forall e \in \mathbb{N}, p \in Partition \cdot$$
$$\quad pre\text{-}GROUP(e,p) \;\Rightarrow\; \exists r \in \mathbb{N}\textbf{-set} \cdot post\text{-}GROUP(e,p,r)$$

This again requires only a type check. The operation which *EQUATEs* two elements (along with their equivalent elements) is:

$EQUATE\ (e_1{:}\mathbb{N}, e_2{:}\mathbb{N})$
ext wr $p\ :\ Partition$
pre $e_1, e_2 \in \bigcup p$
post $p = \{s \in \overleftarrow{p} \mid e_1 \notin s \wedge e_2 \notin s\} \cup \{\bigcup\{s \in \overleftarrow{p} \mid e_1 \in s \vee e_2 \in s\}\}$

Its satisfiability proof obligation is:

$$\forall e_1, e_2 \in \mathbb{N}, \overleftarrow{p} \in Partition \cdot$$
$$\quad pre\text{-}EQUATE(e_1, e_2, \overleftarrow{p})$$
$$\quad \Rightarrow\; \exists p \in Partition \cdot post\text{-}EQUATE(e_1, e_2, \overleftarrow{p}, p)$$

This relies on Lemma 4.14 which was stated – but not proved – on page 104. (Again, see Chapter 11 for a closely analogous example which is proved.) Notice that the pre-condition establishes the third hypothesis of the lemma.

One virtue of this set-based specification is that it is much more succinct than a description based on an implementation. But a more important property is that, because the algebra of the underlying objects is established, it is possible to make deductions about a specification more readily than reasoning about contorted details of a particular representation. It is, for example, easy to prove:

$$p \in Partition;\ e \in \bigcup p;\ post\text{-}GROUP(e,p,r) \;\vdash\; e \in r$$

which asserts that the argument given to *GROUP* will also be a member of the set returned as a result. Or, again:

$$p_1, p_2 \in Partition;\ e, e' \in \mathbb{N};\ e \notin \bigcup p_1;$$
$$e' \in \bigcup p_1;\ post\text{-}ADD(e, p_1, p_2);\ post\text{-}GROUP(e', p_2, r) \;\vdash$$
$$\quad e \notin r$$

The collection and verification of such properties goes some way towards validating the formal specification against the (informal) understanding of the requirements for the system.

Exercise 4.4.5 Express the last inference rule in words and write some inference rules which express other properties of (combinations of) the operations. Do not feel obliged to provide formal proofs at this time.

Exercise 4.4.6 Respecify the equivalence relation problem so that the *EQUATE* and *GROUP* operations take a set of elements as input.

5

Composite Objects and Invariants

We always require an outside point to stand on, in order to apply the lever of criticism.
C. G. Jung

Sets are only one item in the collection from which abstract descriptions of objects can be built. Chapters 6 and 7 introduce further familiar mathematical constructs. In this chapter, a way of forming multicomponent objects is described. In many respects these composite objects are like the records of Pascal or the structures of PL/I; since, however, the properties of composite objects are not exactly the same as for records, a syntax is chosen which differs from that used in programming languages. As with the objects discussed above, an (inductive) proof method is given which facilitates proofs about composite objects. In Section 5.2, data type invariants are discussed in detail. Section 5.3 provides amplification of the concept of states and some related proof obligations.

5.1 Notation

Constructors

Whereas instances of set objects are written using braces, the composite values considered in this chapter are created by so-called *make-functions*. A *composite object* has a number of fields; each such field has a value. A *make-function*, when applied to appropriate values for the fields, yields a value of the composite type. The notation to define composite types is explained below. Suppose, for now, that some composite type has been defined such that each object contains a form of date. The type is called *Datec*; the first field contains a day and the second the year; the relevant make-function might have the signature:

$$mk\text{-}Datec: \{1, \ldots, 366\} \times \mathbb{N} \to Datec$$

A make-function is specific to a type: its name is formed by prefixing *mk-* to the name of the type.

A useful property of make-functions is that they yield a tagged value[1] such that no two different make-functions can ever yield the same value. Thus if two sorts of temperature measurements are to be manipulated, one might have:

$$mk\text{-}Fahrenheit: \mathbb{R} \to Fahrenheit$$
$$mk\text{-}Celsius: \mathbb{R} \to Celsius$$

Even though each of these types has one field, and the field contains a real number in each case, the types *Fahrenheit* and *Celsius* are disjoint (i.e. $mk\text{-}Fahrenheit(0) \neq mk\text{-}Celsius(0)$). It is then possible to form the union type containing both *Fahrenheit* and *Celsius* without them becoming confused.

A particular make-function yields distinct results (composite values) for different arguments (i.e. $mk\text{-}Celsius(0) \neq mk\text{-}Celsius(1)$).

Decomposing objects

One way of decomposing composite values is by selectors. The definitions of such selectors are described below with the notation for defining the composite type itself. For now, assume that the selectors *day* and *year* have been associated with the two fields of *Datec* – then:

$$day(mk\text{-}Datec(7, 1979)) = 7$$
$$year(mk\text{-}Datec(117, 1989)) = 1989$$

Such *selectors*[2] are functions which can be applied to composite values to yield the component values. Thus their signatures are:

$$day: Datec \to \{1, \ldots, 366\}$$
$$year: Datec \to \mathbb{N}$$

There are several other ways of decomposing composite values; each uses the name of the make-function in a context which makes it possible to associate

[1] In fact, a reasonable model for VDM's composite objects is a Cartesian product with a tag. The explanation of the properties of composite objects avoids the need to discuss the model. In particular, the selectors of composite objects can be given more meaningful names than the numeric selectors of tuples.

[2] The selectors serve as projection functions and make-functions as injections.

names with the sub-components of a value. A notation used above for defining local values is:

$$\mathbf{let}\ i = \cdots\ \mathbf{in}\ \cdots i \cdots$$

The expression to the right of the equality sign is evaluated and its value is associated with i, this value of i is used in evaluating the expression to the right of **in**; the **let** construct provides a binding for free occurrences of i in the final expression. This notation can be extended in an obvious way so that it might be said to decompose composite values. Suppose that a function is to be defined whose domain is *Datec* and the definition of the function requires names for the values of the components. The function could be defined, using selectors:

$$\begin{array}{l} \textit{inv-Datec} : \textit{Datec} \to \mathbb{B} \\ \textit{inv-Datec}(dt) \ \ \triangleq \ \ \textit{is-leapyr}(\textit{year}(dt)) \lor \textit{day}(dt) \leq 365 \end{array}$$

Using the extension of **let**, this can be written:

$$\begin{array}{l} \textit{inv-Datec}(dt) \ \ \triangleq \\ \quad \mathbf{let}\ \textit{mk-Datec}(d, y) = dt\ \ \mathbf{in}\ \ \textit{is-leapyr}(y) \lor d \leq 365 \end{array}$$

The **let** construct, in a sense, decomposes dt by associating names with the values of its fields. The frequency with which such decompositions occur on parameters of functions prompts the use of the make-functions directly in the parameter list. Thus an equivalent effect can be achieved by writing:

$$\textit{inv-Datec}(\textit{mk-Datec}(d, y)) \ \ \triangleq \ \ \textit{is-leapyr}(y) \lor d \leq 365$$

The tagging property of make-functions can be used to support a useful **cases** construct. A function which reduces either form of temperature to a *Celsius* value might be written:

$$\begin{array}{lll} \textit{norm-temp} : (\textit{Fahrenheit} \cup \textit{Celsius}) \to \textit{Celsius} & & \\ \textit{norm-temp}(t) & \triangleq & \mathbf{if}\ t \in \textit{Fahrenheit} \\ & & \mathbf{then\ let}\ \textit{mk-Fahrenheit}(v) = t\ \ \mathbf{in}\ \ \textit{mk-Celsius}((v - 32) * 5/9) \\ & & \mathbf{else}\ t \end{array}$$

This is rather cumbersome and an obvious '**cases**' notation can be used which, as in a parameter list, names the components of a composite object:

$$\begin{array}{lll} \textit{norm-temp}(t) & \triangleq & \mathbf{cases}\ t\ \mathbf{of} \\ & & \quad \textit{mk-Fahrenheit}(v) \to \textit{mk-Celsius}((v - 32) * 5/9), \\ & & \quad \textit{mk-Celsius}(v) \qquad \to t \\ & & \mathbf{end} \end{array}$$

At first sight, the range of ways for decomposing composite objects might appear excessive. However, it is normally easy to choose the most economical alternative. For example, it is briefer to use selector functions than decompose an object with **let** if only a few fields of a multicomponent object are referred to within the function; if, on the other hand, all fields are referred to, it is simpler to name them all at once in a **let**. If no reference is made in the body of a function to the value of the entire object, such a **let** can be avoided and the decomposition made in the parameter list. Decomposition via the **cases** construct is obviously of use when several options are to be resolved. Although the notations can be used interchangeably, brevity and clarity result from careful selection.

Defining composite types

The definition of composite types is now considered. While classes of values of type set are defined by the **-set** constructor, the *composite type* is defined – for the *Datec* example above:

compose *Datec* **of**
  day : $\{1, \ldots, 366\}$,
  year : $\mathbb{N}$
end

In general, the name of the type (and thus of its make-function) is written between the **compose** and **of**; after the **of** is written the information about fields – for each field, the name of its selector is followed by the type of value. Similarly:

compose *Fahrenheit* **of**
  v : $\mathbb{R}$
end

compose *Celsius* **of**
  v : $\mathbb{R}$
end

If it is clear that values in a composite type are never going to be decomposed by selectors, the selector names can be omitted altogether in the definition. Thus, it is possible to write:

compose *Celsius* **of** $\mathbb{R}$ **end**

The corresponding sets of objects defined are:

$$\{mk\text{-}Datec(d, y) \mid d \in \{1, \ldots, 366\} \land y \in \mathbb{N}\}$$
$$\{mk\text{-}Fahrenheit(v) \mid v \in \mathbb{R}\}$$
$$\{mk\text{-}Celsius(v) \mid v \in \mathbb{R}\}$$

From the properties of make-functions, it follows that:

$$\textit{is-disj}\,(\textit{Fahrenheit}, \textit{Celsius})$$

Definitions of composite types can be used in any suitable context. Thus, one could write:

$$(\mathbf{compose}\ \textit{Datec}\ \mathbf{of}\ \cdots\ \mathbf{end})\text{-}\mathbf{set}$$

However, the most common context is just to associate a name with the set:

$$\textit{Datec} = \mathbf{compose}\ \textit{Datec}\ \mathbf{of}\ \cdots\ \mathbf{end}$$

This name is often the same as the constructor name. The frequency of this special case justifies an abbreviation. The above definition can be written:

$$\begin{array}{lll} \textit{Datec} :: & \textit{day} & : \{1, \ldots, 366\} \\ & \textit{year} & : \mathbb{N} \end{array}$$

The :: symbol can be read as 'is composed of'; the following two definitions are equivalent:

$$\textit{Name} :: \cdots$$

$$\textit{Name} = \mathbf{compose}\ \textit{Name}\ \mathbf{of}\ \cdots\ \mathbf{end}$$

The :: is actually used far more often than the **compose** form in the sequel.

Names for types can be introduced in definitions to add clarity. For example, the definition given above could be written:

$$\begin{array}{lll} \textit{Datec} :: & \textit{day} & : \textit{Day} \\ & \textit{year} & : \textit{Year} \end{array}$$

$$\textit{Day} = \{1, \ldots, 366\}$$

$$\textit{Year} = \mathbb{N}$$

Since these are simple set equalities, the definitions of *Day* and *Year* have not, however, been tagged by constructors – thus:

$$7 \in (\textit{Day} \cap \textit{Year})$$

Modifying composite objects

The functions associated with composite objects (make-functions and selectors so far) are unlike the operators on sets in that the latter are general whereas those for composite objects are specific to a type. Thus the ADJ diagram given

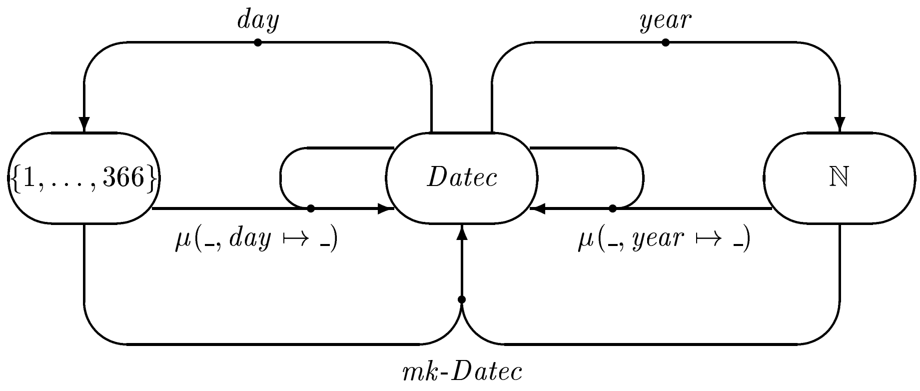

Figure 5.1 ADJ diagram of *Datec* operators

in Figure 5.1 relates solely to the *Datec* example. Only one other function[3] is defined for composite objects: the μ function provides a way of creating a composite value, which differs only in one field, from another; thus:

$dt = mk\text{-}Datec(17, 1927)$
$\mu(dt, day \mapsto 29) = mk\text{-}Datec(29, 1927)$
$\mu(dt, year \mapsto 1937) = mk\text{-}Datec(17, 1937)$

Concrete syntax notations (e.g. BNF) which can be used to define the set of strings of a language are discussed in Section 1.2. An *abstract syntax* is similar in many respects but defines a set of objects which contain only the essential information but do not retain the syntactic marks (e.g. := , ;) which play a part in parsing strings. The definition of the semantics of programming languages uses an abstract syntax in order to avoid irrelevant detail. In fact, one of the reasons that the uniqueness property of make-functions had to be adopted was to simplify the description of the abstract syntax of programming languages. Both the **-set** and **compose** constructs are used in describing abstract syntax

[3]Strictly, there is a whole family of μ functions – one for each composite type. However, since a μ function cannot change the type of a composite object, no confusion arises if μ is used as a generic name. The μ function could be generalized to change more than one field at a time. This is not needed in the current book.

and many examples occur below. In spite of the differences, certain aspects of concrete syntax notation carry over naturally to the description of abstract syntax. The $[\cdots]$ notation for marking things as optional is taken over from concrete syntax along with the idea of distinguishing elementary values by fount change (here set in SMALL CAPS). Thus:

$$Month = \{\textsc{Jan}, \textsc{Feb}, \textsc{Mar}, \textsc{Apr}, \textsc{May}, \textsc{Jun}, \textsc{Jul}, \textsc{Aug}, \textsc{Sep}, \textsc{Oct}, \textsc{Nov}, \textsc{Dec}\}$$

$$\begin{array}{lll} Record :: & day & : \{1, \ldots, 366\} \\ & year & : \mathbb{N} \\ & valid & : [\textsc{Error}] \end{array}$$

The brackets denoting optional items can be read as:

$$[Set] = Set \cup \{\mathbf{nil}\}$$

Thus, an omitted field is marked by the **nil** value and:

$$mk\text{-}Record(366, 1984, \mathbf{nil}) \in Record$$
$$mk\text{-}Record(366, 1983, \textsc{Error}) \in Record$$

Notice also that in the *Record* example the concrete syntax convention of letting a single value be interpreted as a set has been adopted.

Naming conventions

A number of naming conventions are being followed in the examples in this book – although not strictly part of the notation, conformance to some stated set of conventions can significantly aid the readability of large specifications. The conventions here are:

- names of types are printed in italics and have their first letter in upper case and the rest of the name in lower case (e.g. *Datec*) – exceptions are standard names for certain mathematical sets (e.g. $\mathbb{N}$) which are distinguished by being in special founts;
- names of functions (and thus selectors) are in all lower case italic letters;
- names of operations are in all upper case italic letters;
- elementary values (e.g. ERROR) are in a 'small caps' fount.

Data type invariants

The topic of data type invariants which is touched upon above, is now explored more fully. The *day* field of *Datec* is restricted to show that, for instance, 399 can never be a value. This sub-range concept is useful but does not solve the problem of restricting values of composite objects. In several places above (and very many below) it is necessary to show that certain combinations of field values cannot arise. *Data type invariants* are truth-valued functions which can be used to record such restrictions. The function *inv-Datec* discussed above is an obvious invariant on dates. It is convenient to write such restrictions as part of the type definition with a keyword (**inv**) to separate the invariant – thus:

$$\begin{aligned} &\mathit{Datec} :: \mathit{day} : \mathit{Day} \\ &\qquad\quad\ \mathit{year} : \mathit{Year} \\ &\mathbf{inv}\ (\mathit{mk\text{-}Datec}(d, y)) \triangleq \mathit{is\text{-}leapyr}(y) \lor d \le 365 \end{aligned}$$

defines the set:

$$\{\mathit{mk\text{-}Datec}(d, y) \mid d \in \mathit{Day} \land y \in \mathit{Year} \land \mathit{inv\text{-}Datec}(\mathit{mk\text{-}Datec}(d, y))\}$$

Where:

$$\mathit{inv\text{-}Datec}(\mathit{mk\text{-}Datec}(d, y)) \quad \triangleq \quad \mathit{is\text{-}leapyr}(y) \lor d \le 365$$

Here, just as with pre- and post-conditions, the keyword gives rise to a truth-valued function *inv-Datec* which can be used elsewhere. The *valid* objects of *Datec* are those which, as well as belonging to the composite type, also satisfy *inv-Datec*. Thus:

$$d \in \mathit{Datec}$$

is taken to imply that the invariant is satisfied.[4]

Referring back to the example of Section 4.3 which was written:

$$\mathit{Partition} = \{\, p \in (\mathbb{N}\text{-}\mathbf{set})\text{-}\mathbf{set} \mid \mathit{inv\text{-}Partition}(p)\}$$

This can be given in the keyword form as:

$$\begin{aligned} &\mathit{Partition} = (\mathbb{N}\text{-}\mathbf{set})\text{-}\mathbf{set} \\ &\mathbf{inv}\ \mathit{inv\text{-}Partition}(p) \end{aligned}$$

The *Datec* example is typical of the way in which data type invariants arise. Neat mathematical abstractions tend to fit regular situations; some objects which are to be modelled are ragged and do not immediately fit such an abstraction. The truth-valued function which is used as the data type invariant cuts out those

[4]This has a profound consequence for the type mechanism of the notation. In programming languages, it is normal to associate type checking with a simple compiler algorithm. The inclusion of a sub-typing mechanism which allows truth-valued functions forces the type checking here to rely on proofs. The next section shows how such proof obligations are generated and discharged.

elements which do not arise in reality. Section 5.3 shows how invariants are also useful on composite objects used as states.

Some interesting data types can be defined with the aid of recursion. It is possible to write *recursive abstract syntax definitions* such as:

$$Llist = [Llistel]$$

$$\begin{array}{lll} Llistel :: & hd & : \mathbb{N} \\ & tl & : Llist \end{array}$$

These objects are reminiscent of the simplest lists in a list programming language; elements of *Llist* can be **nil**; non-**nil** elements are of type *Llistel* and contain a head and a tail where the latter is a (**nil** or non-**nil** element of) *Llist*. Just as with sets, there is a clear argument for restricting attention to finite objects and it is assumed that all objects satisfying a recursive composite object definition are finite (but, of course, there is an infinite set of such objects because their size is unbounded). It can be useful to think of such objects as trees (notice that cycles cannot be generated by the chosen constructors); Figure 5.2 pictures some elements of *Llist*.

A function which sums the elements of such a list can be written:

$$\begin{array}{ll} lsum : Llist \to \mathbb{N} & \\ lsum(t) \;\; \triangleq & \textbf{cases } t \textbf{ of} \\ & \quad \begin{array}{ll} \textbf{nil} & \to 0, \\ mk\text{-}Llistel(hd, tl) & \to hd + lsum(tl) \end{array} \\ & \textbf{end} \end{array}$$

Notice that this recursive function is well-defined (i.e. it terminates) only because all elements of *Llist* are finite.

Further examples of recursive definitions are given in the next section; these are presented with their invariants and a discussion of the relevant proof methods.

Exercise 5.1.1 Given:

$$\begin{array}{lll} Date :: & year & : \mathbb{N} \\ & month & : Month \\ & day & : \{1, \ldots, 31\} \end{array}$$

- Write the signature of *mk-Date* and of the selectors.

- Use *mk-Date* to construct an object with an interesting date.

- Define a truth-valued function which determines whether the first of two

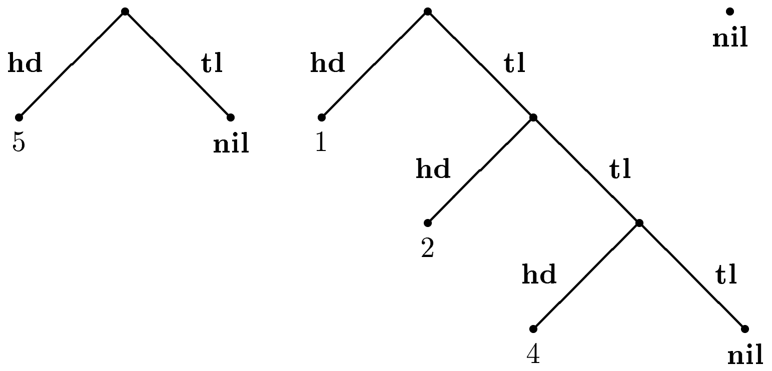

Figure 5.2 Elements of *Llist*

Dates is earlier than a second. Three versions should be given using (respectively) selectors, **let**, and putting *mk-Date* in the parameter list.

- Write a data type invariant for *Date*.
- Use a μ function to modify the 'interesting date'.

Exercise 5.1.2 Define a composite object which could be used to store the time of day to the nearest second. Why is no data type invariant required? Give the signature of a μ function which modifies the minute field of *Time*.

Exercise 5.1.3 Given a specification of (UK) traffic lights:

$Light = Colour\text{-}\mathbf{set}$
$Colour = \{\text{RED}, \text{GREEN}, \text{AMBER}\}$

limit the possible values with a data type invariant.

Exercise 5.1.4 Suppose a hotel requires a system which stores information about room numbers. Floors are numbered 1 to 25 and rooms are numbered 0 to 63. Define a composite object *Roomno* and an invariant to reflect the facts that:

- there is no floor number 13;
- level 1 is an open area and has only room number 0;

- the top five floors consist of large suites and these are numbered with even integers.

Exercise 5.1.5 Write expressions corresponding to the elements of *Llist* pictured in Figure 5.2. Use a μ function to insert a new tail (*tl*) into the first of these objects. Define a (recursive) function *ljoin* which places one list at the end of (i.e. in the **nil** position) of another.

Exercise 5.1.6 This exercise develops the form of list closer to those known in LISP as 'dotted pairs'. Define a set of objects (*Pllist*) which have fields named *car* and *cdr*; these fields can contain either integers or lists. Define one function which gathers the set of numbers in such an object and another function which sums all of the numbers.

Exercise 5.1.7 (*) Given

$S = T$

$\mathbf{inv}\ (s) \triangleq \cdots$

then:

$$(\forall s \in S \cdot p(s)) \;\Leftrightarrow\; (\forall s \in T \cdot \mathit{inv}\text{-}S(s) \;\Rightarrow\; p(s))$$
$$(\exists s \in S \cdot p(s)) \;\Leftrightarrow\; (\exists s \in T \cdot \mathit{inv}\text{-}S(s) \wedge p(s))$$

Explain this using de Morgan's law.

5.2 Structural induction and invariants

Creating induction rules

Recursive definitions of composite objects define infinite sets of (finite) objects; induction rules are needed to prove properties of such sets. Induction rules are given above for natural numbers and sets. For composite objects, in contrast, there is no single induction rule. Instead, it is necessary to generate an induction rule for each recursively defined class of objects. *Structural induction* provides a way of generating the appropriate induction rules.

The fact that such induction rules exist depends on the finiteness of objects which satisfy recursive type definitions. As with the other induction rules, those for structural induction relate to the ways in which objects are generated.

Axiom 5.1 For *Llist* of the preceding section the appropriate induction rule is:

$$\boxed{Llist\text{-}ind}\ \frac{p(\mathbf{nil});\quad hd \in \mathbb{N},\ tl \in Llist,\ p(tl) \vdash p(mk\text{-}Llistel(hd, tl))}{l \in Llist \vdash p(l)}$$

Inspection of this example, should make clear how induction rules are generated for recursively defined objects. The basis comes from the non-recursive (e.g. **nil**) case(s) and the induction step from the recursive case(s).

To illustrate how the induction rule can be used, some proofs about the following function can be performed:

$$\begin{array}{l} ldbl : Llist \to Llist \\ ldbl(t) \ \ \triangleq \\ \quad \textbf{cases } t \textbf{ of} \\ \quad \textbf{nil} \to \textbf{nil}, \\ \quad mk\text{-}Llistel(hd, tl) \to mk\text{-}Llistel(2 * hd, ldbl(tl)) \\ \quad \textbf{end} \end{array}$$

Lemma 5.2 A simple property to prove is that:

$$l \in Llist \vdash 2 * lsum(l) = lsum(ldbl(l))$$

The proof is shown on page 125.

Exercise 5.2.1 Using the definitions above (including Exercise 5.1.5 on page 123), prove by induction:

$$l_1, l_2 \in Llist \vdash lsum(ljoin(l_1, l_2)) = lsum(l_1) + lsum(l_2)$$

Exercise 5.2.2 Give an induction rule for *Pllist* (as in Exercise 5.1.6 on page 123).

- Define a function *flatten* which places the elements of a *Pllist* into a *Llist*.
- Prove $ll \in Pllist \vdash sumll(ll) = lsum(flatten(ll))$

Where *sumll* is the function defined in Exercise 5.1.6.

Invariants in recursive definitions

In order to present more interesting examples of proofs, invariants are now added to recursive definitions. Chapter 8 addresses the problem of finding representations of abstract objects like sets and maps (see Chapter 6): it is necessary to create such representations either because the abstractions are unavailable in the implementation language or to enhance the efficiency of an implementation. One example is finding representations of sets. The sets of *Words* required in the

$$
\begin{array}{lll}
\textbf{from } l \in \mathit{Llist} & & \\
1 & 2 * \mathit{lsum}(\textbf{nil}) & \\
 & \quad = 0 & \mathit{lsum} \\
2 & \quad = \mathit{lsum}(\textbf{nil}) & \mathit{lsum} \\
3 & \quad = \mathit{lsum}(\mathit{ldbl}(\textbf{nil})) & \mathit{ldbl} \\
4 & \textbf{from } \mathit{hd} \in \mathbb{N},\ \mathit{tl} \in \mathit{Llist},\ 2 * \mathit{lsum}(\mathit{tl}) = \mathit{lsum}(\mathit{ldbl}(\mathit{tl})) & \\
4.1 & \quad 2 * \mathit{lsum}(\mathit{mk\text{-}Llistel}(\mathit{hd}, \mathit{tl})) & \\
 & \qquad = 2 * \mathit{hd} + 2 * \mathit{lsum}(\mathit{tl}) & \mathit{lsum},\mathbb{N} \\
4.2 & \qquad = 2 * \mathit{hd} + \mathit{lsum}(\mathit{ldbl}(\mathit{tl})) & \text{ih4} \\
4.3 & \qquad = \mathit{lsum}(\mathit{mk\text{-}Llistel}(2 * \mathit{hd}, \mathit{ldbl}(\mathit{tl}))) & \mathit{lsum} \\
 & \textbf{infer} \qquad = \mathit{lsum}(\mathit{ldbl}(\mathit{mk\text{-}Llistel}(\mathit{hd}, \mathit{ldbl}(\mathit{tl})))) & \mathit{ldbl} \\
\textbf{infer } 2 * \mathit{lsum}(l) = \mathit{lsum}(\mathit{ldbl}(l)) & & \mathit{Llist\text{-}ind}(3,4)
\end{array}
$$

Lemma 5.2: *lsum*

$$
\begin{array}{lll}
\textbf{from } \mathit{lt}, \mathit{rt} \in \mathit{Setrep},\ \mathit{mv} \in \mathbb{N}, & & \\
\quad \mathit{inv\text{-}Node}(\mathit{mk\text{-}Node}(\mathit{lt}, \mathit{mv}, \mathit{rt})),\ i \in \mathit{retrns}(\mathit{mk\text{-}Node}(\mathit{lt}, \mathit{mv}, \mathit{rt})) & & \\
1 & \textbf{from } i < \mathit{mv} & \\
1.1 & \quad \mathit{retrns}(\mathit{mk\text{-}Node}(\mathit{lt}, \mathit{mv}, \mathit{rt})) = & \mathit{retrns} \\
 & \qquad \mathit{retrns}(\mathit{lt}) \cup \{\mathit{mv}\} \cup \mathit{retrns}(\mathit{rt}) & \\
1.2 & \quad i \neq \mathit{mv} & \text{h1} \\
1.3 & \quad i \notin \mathit{retrns}(\mathit{rt}) & \text{h},\mathit{inv\text{-}Node},\text{h1} \\
 & \textbf{infer } i \in \mathit{retrns}(\mathit{lt}) & 1.1,1.2,1.3 \\
2 & \delta(i < \mathit{mv}) & \\
\textbf{infer } i < \mathit{mv} \ \Rightarrow\ i \in \mathit{retrns}(\mathit{lt}) & & \Rightarrow\text{-}I(2,1)
\end{array}
$$

Lemma 5.3

spell-checking application is a particular example studied below. Here, the problem of representing a set of natural numbers is considered. A large set of numbers can be stored in a binary tree to facilitate efficient updating and checking. Such a *binary tree*:

- has two (possibly **nil**) branches and a number at each node;
- is arranged so that all numbers in the left branch of a node are less than (and all numbers in the right branch are greater than) the number in the node;
- is balanced to increase efficiency.

The relevant data structure is defined:

$$Setrep = [Node]$$

$$\begin{array}{lll} Node :: & lt & : Setrep \\ & mv & : \mathbb{N} \\ & rt & : Setrep \end{array}$$
$$\textbf{inv } (mk\text{-}Node(lt, mv, rt)) \triangleq$$
$$\quad (\forall lv \in retrns(lt) \cdot lv < mv) \wedge (\forall rv \in retrns(rt) \cdot mv < rv)$$

A function which retrieves the set of numbers in a tree is:

$$retrns : Setrep \to \mathbb{N}\textbf{-set}$$
$$retrns(sr) \quad \triangleq$$
$$\begin{array}{ll} \quad \textbf{cases } sr \textbf{ of} & \\ \quad \textbf{nil} & \to \{\,\}, \\ \quad mk\text{-}Node(lt, mv, rt) & \to retrns(lt) \cup \{mv\} \cup retrns(rt) \\ \quad \textbf{end} & \end{array}$$

The invariant captures the second requirement above; the third requirement is discussed in Exercise 5.2.4 on page 127. Notice that writing the invariant with *Node* requires that it applies to *all* occurrences of *Node* within the tree, not just the root. If this were not done the invariant would have to be a recursive function; moreover, proofs involving *Nodes* would be more complicated.

Lemma 5.3 The invariant results in the following simple Lemma about *Nodes*:

$$\boxed{\text{L5.3}}\ \frac{i \in retrns(mk\text{-}Node(lt, mv, rt))}{i < mv \;\Rightarrow\; i \in retrns(lt)}$$

The proof is shown on page 125. Notice how the fact that the antecedent is defined is used in order to prove that the implication holds.

A function which checks whether a number is in such a set representation can be defined. Direct definitions are being used here rather than implicit specifications. This is often the case as design steps tackle implementation details.

Assuming *inv-Node* is true, a function which tests whether values are in *Setrep* can be defined:

$$
\begin{array}{l}
isin : \mathbb{N} \times Setrep \to \mathbb{B} \\
isin(i, sr) \quad \triangleq \\
\quad \textbf{cases } sr \textbf{ of} \\
\quad \textbf{nil} \hspace{7em} \to \textbf{false}, \\
\quad mk\text{-}Node(lt, mv, rt) \to \textbf{if } i = mv \\
\hspace{11em} \textbf{then true} \\
\hspace{11em} \textbf{else if } i < mv \textbf{ then } isin(i, lt) \textbf{ else } isin(i, rt) \\
\quad \textbf{end}
\end{array}
$$

Axiom 5.4 The induction rule for *Setrep* is:

$$
\boxed{Setrep\text{-}ind}\;\frac{\begin{array}{l} p(\textbf{nil}); \\ mv \in \mathbb{N},\ lt, rt \in Setrep, \\ \ inv\text{-}Node(mk\text{-}Node(lt, mv, rt)),\ p(lt),\ p(rt) \vdash \\ \hspace{14em} p(mk\text{-}Node(lt, mv, rt)) \end{array}}{sr \in Setrep \vdash p(sr)}
$$

This can be used to prove:

Lemma 5.5

$$
\boxed{\text{L5.5}}\;\frac{i \in \mathbb{N};\ sr \in Setrep}{isin(i, sr) \ \Leftrightarrow \ i \in retrns(sr)}
$$

A proof is shown on page 128.

Exercise 5.2.3 Define a function which inserts a number into a *Setrep* and prove that the function preserves the invariant (it will be necessary to conjoin a property about the result in order to make the induction work). Do not bother to preserve the 'balanced tree' property (yet).

Exercise 5.2.4 (*) Define a function which deletes a number from a *Setrep* and show that the function preserves the invariant and has the expected effect on the set of numbers. (Deletion is significantly harder than insertion.) Do not, in the first attempt, try to preserve the 'balanced tree' property.

The property of a tree being (height) balanced has not been formalized yet. Write a suitable invariant. Use this to give an implicit specification of a delete function which does preserve the property.

from	$i \in \mathbb{N},\ sr \in Setrep$	
1	$\neg\, isin(i, \mathbf{nil})$	$isin$
2	$retrns(\mathbf{nil}) = \{\,\}$	$retrns$
3	$i \notin retrns(\mathbf{nil})$	Set,2
4	$isin(i, \mathbf{nil}) \;\Leftrightarrow\; i \in retrns(\mathbf{nil})$	$\Leftrightarrow$-I(1,3)
5	**from** $mv \in \mathbb{N},\ lt, rt \in Setrep, inv\text{-}Node(mk\text{-}Node(lt, mv, rt)),$ $(isin(i, lt) \;\Leftrightarrow\; i \in retrns(lt)), (isin(i, rt) \;\Leftrightarrow\; i \in retrns(rt))$	
5.1	$i < mv \vee i = mv \vee i > mv$	$\mathbb{N}$
5.2	**from** $i = mv$	
5.2.1	$isin(i, mk\text{-}Node(lt, mv, rt))$	$isin$,h5.2
5.2.2	$i \in retrns(mk\text{-}Node(lt, mv, rt))$	$retrns$,h5.2
	infer $isin(i, mk\text{-}Node(lt, mv, rt)) \;\Leftrightarrow\; i \in retrns(mk\text{-}Node(lt, mv, rt))$	$\Leftrightarrow\text{-}I$(5.2.1,5.2.2)
5.3	**from** $i < mv$	
5.3.1	$isin(i, mk\text{-}Node(lt, mv, rt)) \;\Leftrightarrow\; isin(i, lt)$	h5.3,$isin$
5.3.2	$i \in retrns(lt) \;\Leftrightarrow\; i \in retrns(mk\text{-}Node(lt, mv, rt))$	L5.3,ih5,h5.3
	infer $isin(i, mk\text{-}Node(lt, mv, rt)) \;\Leftrightarrow\; i \in retrns(mk\text{-}Node(lt, mv, rt))$	$\Leftrightarrow\text{-}trans$(5.3.1,ih5,5.3.2)
5.4	**from** $i > mv$	
	similar	
	infer $isin(i, mk\text{-}Node(lt, mv, rt)) \;\Leftrightarrow\; i \in retrns(mk\text{-}Node(lt, mv, rt))$	
	infer $isin(i, mk\text{-}Node(lt, mv, rt)) \;\Leftrightarrow\; i \in retrns(mk\text{-}Node(lt, mv, rt))$	$\vee\text{-}E$(5.1,5.2,5.3,5.4)
infer	$isin(i, sr) \;\Leftrightarrow\; i \in retrns(sr)$	$Setrep\text{-}ind$(4,5)

Lemma 5.5

5.3 States and proof obligations

Satisfiability

The process of design proceeds, normally in several stages, from specification to implementation. At each stage of design, a claim is being made that the design coincides, in some way, with what has gone before – for example some piece of code satisfies a module specification. In an informal development method, such claims are often only implicit; they are not capable of formalization since the specifications, etc. are informal. In the rigorous approach, such claims are made explicit: they give rise to *proof obligations*. Such proof obligations are in the form of sequents to be proved. The formality of the specification makes these proof obligations quite precise. The level of detail to be employed in a particular proof depends on judgement – thus the method is rigorous rather than completely formal. The virtue of recognizing proof obligations is to ensure that issues like satisfiability are not overlooked and to provide a hook for extra formality if required.

Even when specifications alone are considered, there are proof obligations. It is possible to write implicit specifications which cannot be satisfied. For example, a post-condition can be written which requires a number such that it and its successor are even, or a function can be specified to produce the 'largest prime number'.

Proof obligation 5.6 The proof obligation of *satisfiability* requires that, for any function or operation, some result must exist for each valid input. For example, for:

$$f\,(i{:}\,D)\,d{:}\,R$$
$$\textit{pre-f}: D \to \mathbb{B}$$
$$\textit{post-f}: D \times R \to \mathbb{B}$$

the condition is:

$$\forall d \in D \cdot \textit{pre-f}\,(d) \;\Rightarrow\; \exists r \in R \cdot \textit{post-f}\,(d, r)$$

This states that there must exist an f which satisfies the specification. It is, however, the case that the need to establish satisfiability can frequently be discharged with a minimum of work.

Theorem 5.7 For example, the appropriate sequent for proof obligation 5.6 for the *pi* function of Section 3.2 is:

$$x \in \mathbb{R} \;\vdash\; \exists r \in \mathbb{R} \cdot abs(\pi - r) \leq 10^{-2}$$

This expression is obviously true. Since, however, this is the first proof which requires $\exists$-I, its form is shown (notice how the bound variable r is substituted for the 3.141):

from $x \in \mathbb{R}$		
1	$3.141 \in \mathbb{R}$	$\mathbb{R}$
2	$abs(\pi - 3.141) \le 10^{-2}$	$\mathbb{R}$
infer $\exists r \in \mathbb{R} \cdot abs(\pi - r) \le 10^{-2}$		$\exists$-$I(2)$

Theorem 5.7: *pi*

Even in the case of some of the more complex explicit function definitions given above, the satisfiability proof obligation is straightforward.

Theorem 5.8 For example, the square function requires:

$$i \in \mathbb{N} \vdash \exists r \in \mathbb{N} \cdot r = i^2$$

Which is obviously true from knowledge of the natural numbers.

Some appreciation of the need for satisfiability can be seen from an example where it does not hold. Suppose that square root were specified so as to require:

$$\forall i \in \mathbb{N} \cdot \exists rt \in \mathbb{N} \cdot rt^2 = i$$

This is obviously not true, as can be shown by a simple counter example:

$$\neg \exists rt \in \mathbb{N} \cdot rt^2 = 2$$

There are cases where the satisfiability proof obligation is not at all obvious and it is no easier to prove than simply creating the implementation. In such cases, the proof obligation should be used as an item on a checklist in a review and – given a strong feeling that it is satisfied – work on the implementation should proceed.

It must be kept in mind that type information interacts with the pre- and post-conditions when considering satisfiability. Thus an operation with a pre-condition of $x < 2$ and a post-condition of $x = \overleftarrow{x} - 2$ is satisfiable for integers (or reals) but not where x is constrained to be a natural number.

Such satisfiability constraints carry over in an obvious way from functions to operations. Since it is only necessary to fix an order for the parameters of the pre- and post-conditions when they are taken out of their context.

Influence of invariants

Invariants – which are a part of the type discipline – also play a part in satisfiability. An operation which has write access to a variable of type *Datec*, must not generate a value like $mk\text{-}Datec(366, 1923)$.

Proof obligation 5.9 No operation specification must be written which rules out all valid elements of *Datec*. So:

$$\begin{array}{l} OP\ (i{:}\,D)\ o{:}\,R \\ \textbf{ext wr}\ dt\ :\ Datec \\ \textbf{pre}\ p(i, dt) \\ \textbf{post}\ q(i, \overleftarrow{dt}, o, dt) \end{array}$$

must satisfy:

$$\begin{array}{l} \forall i \in D, \overleftarrow{dt} \in Datec \cdot \\ \qquad pre\text{-}OP(i, \overleftarrow{dt}) \ \Rightarrow\ \exists o \in R, dt \in Datec \cdot post\text{-}OP(i, \overleftarrow{dt}, o, dt) \end{array}$$

Examples involving *Setrep* or *Partition* behave in exactly the same way and it should now be clear why emphasis was placed on invariant preservation lemmas when these objects were introduced. The concept of satisfiability provides a way of identifying rules for different contexts. In each case, the requirement is to see that a specification does not preclude all possible implementations.

The idea of recording the external variables of an operation makes it possible to avoid mentioning any irrelevant variables. There is an obvious way in which an operation can be used in a state which has, at least, all of the required external variables. There is, of course, also a requirement that the types match. A state[5] can be defined as a composite object and can have an invariant. The satisfiability proof obligation for an operation which is to be used in such a state must reflect the invariant on that state. Consider the example, from Section 4.4, which controls information about people. The state could be:

$$\begin{array}{lll} World\ ::\ male & : & Name\textbf{-set} \\ \qquad\qquad female & : & Name\textbf{-set} \\ \qquad\qquad married & : & Name\textbf{-set} \\ \multicolumn{3}{l}{\textbf{inv}\ (mk\text{-}World(m, f, e)) \triangleq is\text{-}disj(m, f) \wedge e \subseteq (m \cup f)} \end{array}$$

No operation which has only read access to the state can disturb the invariant. However, the operation:

[5]Section 9.1 introduces the module concept which binds operations together with a specific state.

$$
\begin{array}{l}
BIRTHM\ (n{:}\,Name)\\
\textbf{ext wr}\ male\ \ :\ Name\text{-}\textbf{set},\\
\quad\ \textbf{rd}\ female\ :\ Name\text{-}\textbf{set}\\
\textbf{pre}\ n \notin (male \cup female)\\
\textbf{post}\ male = \overleftarrow{male} \cup \{n\}
\end{array}
$$

poses a non-trivial satisfiability proof obligation.

Theorem 5.10 The basic form is:

$$
\begin{array}{l}
\forall n \in Name, \overleftarrow{w} \in World \cdot\\
\qquad pre\text{-}BIRTHM(n, male(\overleftarrow{w}), female(\overleftarrow{w})) \ \Rightarrow\\
\qquad\quad \exists w \in World \cdot\\
\qquad\qquad post\text{-}BIRTHM(n, male(\overleftarrow{w}), female(\overleftarrow{w}), male(w)) \wedge\\
\qquad\qquad\quad female(w) = female(\overleftarrow{w}) \wedge married(w) = married(\overleftarrow{w})
\end{array}
$$

The two final conjuncts come from the fact that the externals show that *BIRTHM* cannot change these values. The set *World* is constrained by *inv-World* such that:

$$
\begin{array}{l}
World = \{mk\text{-}World(m, f, e) \mid\\
\qquad\qquad m, f, e \in Name\text{-}\textbf{set} \wedge inv\text{-}World(mk\text{-}World(m, f, e))\}
\end{array}
$$

Proofs about quantifiers ranging over such set comprehensions are discussed in Exercise 5.1.7 on page 123. From the equivalences there, it can be seen that the proof obligation becomes:

$$
\begin{array}{l}
\forall n \in Name, \overleftarrow{m}, \overleftarrow{f}, \overleftarrow{e} \in Name\text{-}\textbf{set} \cdot\\
\qquad inv\text{-}World(mk\text{-}World(\overleftarrow{m}, \overleftarrow{f}, \overleftarrow{e})) \ \Rightarrow\\
\qquad\quad (\,pre\text{-}BIRTHM(n, \overleftarrow{m}, \overleftarrow{f}\,) \ \Rightarrow\\
\qquad\qquad \exists m \in Name\text{-}\textbf{set} \cdot\\
\qquad\qquad\ inv\text{-}World(mk\text{-}World(m, \overleftarrow{f}, \overleftarrow{e})) \wedge post\text{-}BIRTHM(n, \overleftarrow{m}, \overleftarrow{f}, m))
\end{array}
$$

Using Lemma 1.20 (page 27) and the usual translation into a sequent, the proof is shown on page 133. It is not normally necessary to produce such formal versions of satisfiability proofs. It is done here by way of illustration.

The role of invariants on states can perhaps best be visualized by considering them as some form of global (or 'meta') pre- and post-condition: an invariant on a state is an assertion which can be thought of as having been conjoined to the pre- and post-conditions of all operations on that state.

This raises the question of why it is thought worth separating data type invariants. There are three main arguments:

- for consistency checking;

$$
\begin{array}{llr}
\textbf{from } n \in \mathit{Name},\ \overleftarrow{m}, \overleftarrow{f}, \overleftarrow{e} \in \mathit{Name}\textbf{-set} & & \\
1 \quad \textbf{from } \mathit{is\text{-}disj}(\overleftarrow{m}, \overleftarrow{f}) \wedge \overleftarrow{e} \subseteq (\overleftarrow{m} \cup \overleftarrow{f}) \wedge n \notin (\overleftarrow{m} \cup \overleftarrow{f}) & & \\
1.1 \qquad \overleftarrow{m} \cup \{n\} \in \mathit{Name}\textbf{-set} & & \text{h},\cup \\
1.2 \qquad \mathit{is\text{-}disj}(\overleftarrow{m} \cup \{n\}, \overleftarrow{f}) & & \text{h1,h1,}\mathit{is\text{-}disj} \\
1.3 \qquad \overleftarrow{e} \subseteq (\overleftarrow{m} \cup \{n\} \cup \overleftarrow{f}) & & \text{h1},\cup \\
1.4 \qquad \overleftarrow{m} \cup \{n\} = \overleftarrow{m} \cup \{n\} & & \\
\quad\ \ \textbf{infer } \exists m \in \mathit{Name}\textbf{-set} \cdot & & \exists\text{-}I(\wedge\text{-}I(1.2,1.3,1.4),1.1) \\
\qquad\qquad \mathit{is\text{-}disj}(m, \overleftarrow{f}) \wedge \overleftarrow{e} \subseteq (m \cup \overleftarrow{f}) \wedge m = \overleftarrow{m} \cup \{n\} & & \\
\textbf{infer } \mathit{is\text{-}disj}(\overleftarrow{m}, \overleftarrow{f}) \wedge \overleftarrow{e} \subset (\overleftarrow{m} \cup \overleftarrow{f}) \wedge n \notin (\overleftarrow{m} \cup \overleftarrow{f}) \ \Rightarrow & & \Rightarrow\text{-}I \\
\qquad \exists m \in \mathit{Name}\textbf{-set} \cdot & & \\
\qquad\qquad \mathit{is\text{-}disj}(m, \overleftarrow{f}) \wedge \overleftarrow{e} \subseteq (m \cup \overleftarrow{f}) \wedge m = \overleftarrow{m} \cup \{n\} & &
\end{array}
$$

Theorem 5.10: *BIRTHM*

- to guide subsequent revisions; and
- to ease implementation.

It is not possible to prove formally that a specification matches a user's wishes since these latter are inherently informal but the more that can be done to postulate and prove theorems about a specification, the greater is the chance of discovering any unexpected properties of the chosen specification. Thus the obligation to prove results about invariants can be seen as an opportunity to increase confidence in the consistency of a specification.

The techniques described in this book were originally developed in an industrial environment. The sort of application considered was rarely stable; specifications often had to be updated. Recording data type invariants is one way in which the authors of a specification can record assumptions about the state on which their operations work. An explicit assumption, and its attendant proof obligation, are likely to alert someone making a revision to an error which could be missed if the reliance were left implicit. The task of showing that representations are adequate for abstractions used in specifications is addressed in Section 8.1. It should, however, be intuitively clear that the search for representations is facilitated by limits to the abstraction.

Long invariants can provide a warning. Different states with different invariants can be used to define exactly the same behaviour of a collection of operations. How is one to choose between alternative models? Although there are these advantages in recording invariants, it is also true that their presence – or complexity – can provide a hint that a simpler state model might be more appropriate. This point is pursued below when other data-structuring mechanisms are available. But it is generally true that a state with a simpler invariant is to be preferred in a specification to one with a complex invariant.

The process of designing representations frequently forces the inclusion of redundancy; typically, this might be done to make some operation efficient. Such redundancy (e.g. a doubly-linked list) gives rise to invariants. Thus, as in the *Setrep* example above, more complex invariants do tend to arise in the design process.

As can be seen, data type invariants provide information about any single state which can arise. They do not provide information about the way in which states change (e.g. a constraint that a variable does not increase in value). Knowledge about single states (e.g. $fn = t!$ in the factorial example used in Exercise 3.4.4 on page 86) and between states (e.g. the greatest common divisor of i and j is the same in each succeeding state) both have parts to play in the implementation proofs of Chapter 10. In specifications themselves, however, it is data type invariants which are most useful.

Exercise 5.3.1 Write out the satisfiability proof obligation (without proof) for:

- *double* (cf. Exercise 3.2.1 on page 57);
- *choose* (cf. Exercise 3.2.3 on page 58);
- *mult* (cf. Exercise 3.2.6 on page 61).

Exercise 5.3.2 Outline the proof of the first part of Exercise 5.3.1 – this is very simple but shows the overall idea.

Exercise 5.3.3 Exercise 4.4.3 on page 108 can be specified in (at least) two ways. The different models are distinguished by their invariants. Document the invariant used in answering that exercise and prove that the operations are satisfiable with respect to it. Then find another model and record its invariant.

Data types

The notion of data type is very important in modern programming methods. The view taken in this book is that a *data type* characterizes a behaviour. The *behaviour* is the relationship between the results of the operators of the data type. The importance of this relationship is that a value is exposed in other, more basic, data types. Thus, in the *World* example above, *Name* is taken as a basic type and the behaviour of the operations can be observed via their inputs and outputs.

Clearly, if one knows all about the behaviour of a data type, one need know nothing else in order to use the data type. The fact that it is realized (or implemented) in some particular way is unimportant. For the specification of the operations around the *World* example (*BIRTHM*, etc.) the choice of the specific state is an artifact of the specification. This focuses the discussion on how data types can be specified. For interesting data types, the behaviours are infinite and it is clear that they have to be specified other than by enumeration. Section 9.4 shows how the properties themselves can sometimes be used to form a specification. The approach followed in the body of this book is to specify data types via models. Not only is a particular composite object (containing sets) chosen as the model for *World*, but also the map objects in the next chapter can be modelled by sets of pairs. This *model-oriented* approach appears to be appropriate for the specification of larger computer systems. There are some dangers in the approach and these are discussed in Section 9.3. Basically, the model must be seen as a way of describing the essential behaviour and implementation choices must be avoided.

There is another distinction about data types which is worth clarifying since it often confuses discussions about their specification. Data types like sets or integers have operators which are purely functional in the sense that their results depend only on their arguments. In contrast, the results of operations (in an example like the calculator of Section 3.4) depend on the state. This distinction is made here by referring to *functional data types* and *state-based data types*. In the main, the specifications of computer systems are state-based data types. In the model-oriented approach to specifications, the states themselves are built using functional data types (e.g. sets).

A model-oriented specification of a state-based data type comprises:

- a definition of the set of states (normally including invariants);
- a definition of possible initial states (often exactly one); and
- a collection of operations whose external variables are parts of the state: these operations must be satisfiable.

Section 9.1 describes a fixed concrete syntax for presenting a whole data-type specification. This is not used in the body of this book because of the wish to focus on concepts rather than details of syntax. In a state-based data type, the history of the operations plays a part in governing the behaviour. Even so, the behaviour can be seen as the essence of the data type. The model is a convenient way of defining the behaviour. To a user of the data type, internal details of the state are important only in so far as they affect the observable behaviour. Those details which are not made visible by operations should be ignored.[6]

[6]Section 3.4 explains why one operation cannot, as such, be used in the specification of another. It is, however, clear that the separation provided by data types is very useful in structuring specifications. There is, therefore, a need to be able to use, in some way, even state-based data types in the specifications of others. This topic is taken up in Section 9.1.

6

Map Notation

If you are faced by a difficulty or a controversy in science, an ounce of algebra is worth a ton of verbal argument.
J. B. S. Haldane

Functions define a mapping between their domain and range sets – a result can be computed by evaluating the expression in the direct definition with particular arguments substituted for the parameter names. Their definitions use powerful concepts which make it – in general – impossible to answer even simple questions about functions such as whether they yield a result for some particular argument value. When a mapping is required in a specification, it is often sufficient to construct a finite map; the virtue of explicitly recognizing the more restricted case is that more powerful operators can be defined. The maps which are described in this chapter are, however, similar to functions in many respects and the terminology and notation adopted reflects the similarities. The differences result from the fact that the argument/result relationship is explicitly constructed for maps. Building a map is like building a table of pairs; application of a map requires table look-up rather than evaluation of a defining expression. Furthermore, whereas functions are defined by a fixed rule, maps are often created piecemeal.

Access to information via keys is very common in computer applications and poses significant implementation problems. A powerful abstract notation for maps provides a crucial tool for the construction of concise specifications. Consequently, maps are the most common structure used in large specifications.

6.1 Notation

Representing equivalence relations

In order to provide an introduction to the notation, a specification is shown – in terms of maps – which defines the same behaviour for the operations as that for the equivalence relation problem in Section 4.4. It should be remembered that elements of $\mathbb{N}$ have to be separated into partitions; partitions can be merged by an $EQUATE$ operation; another operation makes it possible to find the $GROUP$ of elements in the same partition as some given element. In the definition to be given here, the property of being in the same partition is captured by a map: equivalent elements are mapped to the same partition identifier (the set of which is Pid). The required map type is defined:

$$Partrep = \mathbb{N} \xrightarrow{m} Pid$$

Thus the partition:

$$\{\{3, 6\}, \{5\}, \{7, 2, 1\}\}$$

might be represented by a table of $\mathbb{N}/Pid$ values:

3	pid_1
6	pid_1
5	pid_2
7	pid_3
2	pid_3
1	pid_3

A linear presentation of *map* values can be used: individual pairs are known as *maplets* and the elements are separated by a special arrow ($\mapsto$); the collection of pairs is contained in set braces. Thus:

$$\{3 \mapsto pid_1, 6 \mapsto pid_1, 5 \mapsto pid_2, 7 \mapsto pid_3, 2 \mapsto pid_3, 1 \mapsto pid_3\}$$

The map is shown as a set of maplets or element pairs. Their order is unimportant and a natural model for finite maps is a finite set of ordered pairs. *Arbitrary* sets of such pairs would, however, be too general. In order for maps to be used with a function style of notation, they must satisfy the restriction that no two pairs have the same left-hand value. In other words, a map represents a *many-to-one* mapping.

The information about variables, etc. for the $GROUP$ operation can be rewritten:

$GROUP\ (e{:}\,\mathbb{N})\ r{:}\,\mathbb{N}\text{-}\mathbf{set}$
ext rd m : $Partrep$
pre $\cdots$
post $\cdots$

The post-condition must require that the set r contains all elements which map to the same *Pid* as e. Application of a map is just like function application and the same notation is used. Thus *post-GROUP* is:

$$r = \{e' \in \cdots \mid m(e') = m(e)\}$$

Completing the post-condition – and writing the pre-condition – requires that the domain of the map be known because the definition of a map fixes the maximum set of values and each instance of such a map value has a domain (**dom**) which is a subset of the maximum set. Using this operator, the specification of *GROUP* can be completed:

$GROUP\ (e{:}\,\mathbb{N})\ r{:}\,\mathbb{N}\text{-}\mathbf{set}$
ext rd m : $Partrep$
pre $e \in \mathbf{dom}\, m$
post $r = \{e' \in \mathbf{dom}\, m \mid m(e') = m(e)\}$

The post-condition of the *EQUATE* operation must describe how m changes. There is a mapping override operator ($\dagger$) which enables pairs from its second operand to take precedence over any pairs from its first operand for the same key – thus:

$$\{a \mapsto 1, b \mapsto 2\} \dagger \{a \mapsto 3, c \mapsto 4\} = \{a \mapsto 3, b \mapsto 2, c \mapsto 4\}$$

It would be possible to write in *post-EQUATE*:

$$m = \overleftarrow{m} \dagger \{e_1 \mapsto \overleftarrow{m}(e_2)\}$$

but this would be wrong! By changing only one key, other members of the e_1 partition would not be updated (and the transitivity property would be lost). A comprehension notation, like that for sets, can be used for maps. The correct specification of *EQUATE* is:

$EQUATE\ (e_1{:}\,\mathbb{N}, e_2{:}\,\mathbb{N})$
ext wr m : $Partrep$
post $m = \overleftarrow{m} \dagger \{e \mapsto \overleftarrow{m}(e_2) \mid e \in \mathbf{dom}\, \overleftarrow{m} \wedge \overleftarrow{m}(e) = \overleftarrow{m}(e_1)\}$

The second operand of the override contains all pairs from the old value of m which have the same key as e_1 did in the old value of m.

The initial value of *Partrep* is defined to be the empty map: $m_0 = \{\,\}$.

Continuing in this way would result in there being two specifications of the equivalence relation problem. Chapter 8 introduces the methods by which one can be shown to model the other. (The choice of the name *Partrep* was made to suggest its being a representation of *Partition*.) Chapter 11 takes a variant of this problem through the process of data reification (and operation decomposition down to code).

Now that the collection of data type constructors is larger, it is necessary to spend more time considering which model best suits the task to be specified and this is taken up in Section 9.2. Abstraction is interesting – but not always easy.

Operators

The remainder of this section takes a closer look at the notation for maps. Maps are associations between two sets of values; within a pair (maplet), the key and value are separated by $\mapsto$; a map value contains a collection of such pairs where no two pairs have the same first element. For example:

$$\{1 \mapsto 1, 2 \mapsto 4, -1 \mapsto 1, 0 \mapsto 0\}$$

The pairs can be written in any order within the braces:

$$\{1 \mapsto 1, 2 \mapsto 4, -1 \mapsto 1, 0 \mapsto 0\} = \{-1 \mapsto 1, 0 \mapsto 0, 1 \mapsto 1, 2 \mapsto 4\}$$

Map values can also be defined by comprehension in a way which reflects the fact that maps are simply sets of pairs. Thus:

$$\{i \mapsto i^2 \in \mathbb{N} \times \mathbb{N} \mid i \in \{-1, \ldots, 2\}\}$$

is the same map value as above. The general form is:

$$\{x \mapsto f(x) \in X \times Y \mid p(x)\}$$

But, since it is normally obvious, the constraint is frequently omitted. Such expressions must be written so as to generate only finite[1] maps. With care, one can also write map comprehension as:

$$\{x \mapsto y \mid q(x, y)\}$$

but, in order to be able to look up values, it is essential that q does not associate two different y values with the same x value.

The examples which follow use the values:

$$m_1 = \{a \mapsto 1, c \mapsto 3, d \mapsto 1\}, \qquad m_2 = \{b \mapsto 4, c \mapsto 5\}$$

[1]This restriction is required – as with other objects – to admit induction.

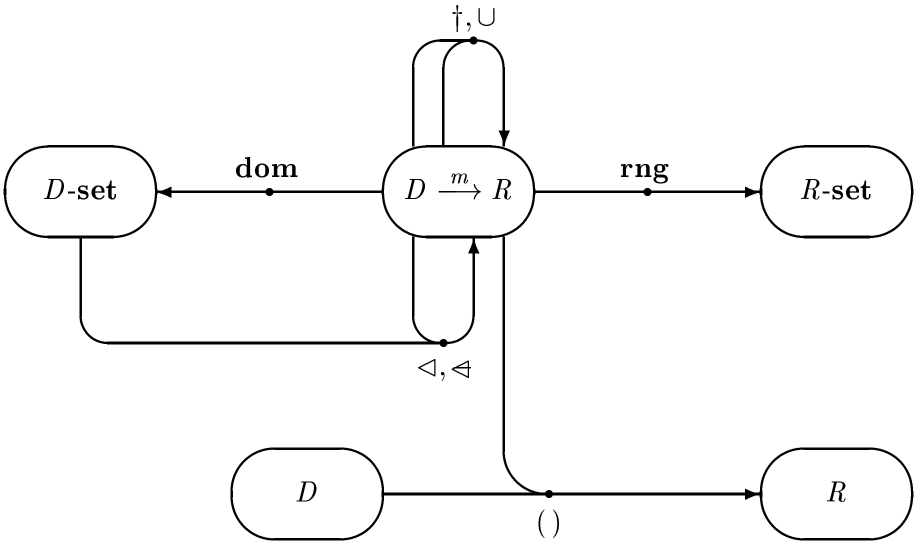

Figure 6.1 ADJ diagram of map operators

The domain operator yields, when applied to a map value, the set of first elements of the pairs in that map value. Thus:

$\mathbf{dom}\, m_1 = \{a, c, d\}$
$\mathbf{dom}\, m_2 = \{b, c\}$

and for the empty map:

$\mathbf{dom}\, \{\,\} = \{\,\}$

A map value can be applied to a value for which it is defined (is in the set given by **dom**) – thus:

$m_1(a) = 1$
$m_2(c) = 5$

and for maps defined by comprehension:

$$m = \{x \mapsto f(x) \mid p(x)\} \land p(x_0) \;\Rightarrow\; m(x_0) = f(x_0)$$

Given an understanding of these operators, all other map operators (see Figure 6.1 for the ADJ diagram) can be defined.

The set of values on the right of the pairs contained in a map can be determined by the range operator:

$$\begin{aligned}&\mathbf{rng}\, m_1 = \{1, 3\}\\&\mathbf{rng}\, m_2 = \{4, 5\}\\&\mathbf{rng}\, \{\,\} = \{\,\}\end{aligned}$$

which is defined:

$$\mathbf{rng}\, m = \{m(d) \mid d \in \mathbf{dom}\, m\}$$

Notice that, as a result of the many-to-one property, for any map value m:

$$\mathbf{card\, rng}\, m \leq \mathbf{card\, dom}\, m$$

The map override operator yields a map value which contains all of the pairs from the second (map) operand and those pairs of the first (map) operand whose first elements are not in the domain of the second operand. Thus:

$$\begin{aligned}&m_1 \dagger m_2 = \{a \mapsto 1, b \mapsto 4, c \mapsto 5, d \mapsto 1\}\\&m_2 \dagger m_1 = \{a \mapsto 1, b \mapsto 4, c \mapsto 3, d \mapsto 1\}\\&m \dagger \{\,\} = m = \{\,\} \dagger m\end{aligned}$$

The types of all of the map operators can be read from Figure 6.1; map override is defined:

$$\begin{aligned}&ma \dagger mb \triangleq\\&\quad \{d \mapsto\\&\qquad (\mathbf{if}\ d \in \mathbf{dom}\, mb\ \mathbf{then}\ mb(d)\ \mathbf{else}\ ma(d)) \mid\\&\qquad\quad d \in (\mathbf{dom}\, ma \cup \mathbf{dom}\, mb)\}\end{aligned}$$

Notice that the domain of the second operand can contain elements which are not in the domain of the first operand.

The override operator is not commutative. When the domains of two map values are disjoint, the values can be combined by a union operator:

$$m_2 \cup \{a \mapsto 7\} = \{a \mapsto 7, b \mapsto 4, c \mapsto 5\}$$

for which:

$$\textit{is-disj}(\mathbf{dom}\, ma, \mathbf{dom}\, mb) \;\Rightarrow\; ma \cup mb = mb \cup ma$$

The definition of map union is identical with that for override, so:

$$\textit{is-disj}(\mathbf{dom}\, ma, \mathbf{dom}\, mb) \;\Rightarrow\; ma \cup mb = ma \dagger mb$$

The advantage of identifying – with a distinct operator – the special case of disjoint domains is that the commutativity property can be used in proofs. Remember, however, that the union operator is undefined if the domains of the operands overlap.

The union symbol is used in two distinct contexts. Strictly, set union and map union are two different operators. The same symbol is used because of their similarity. Such overloading is familiar both in mathematics and in programming languages. For example, Pascal uses the same plus operator for integer and real numbers (as well as for set union!).

A restriction operator ($\lhd$) is defined with a first operand which is a set value and a second operand which is a map value; the result is all of those pairs in the map value whose first element is in the set value. Thus:

$$\{a, d, e\} \lhd m_1 = \{a \mapsto 1, d \mapsto 1\}$$
$$\{\,\} \lhd m_1 = \{\,\}$$
$$s \lhd \{\,\} = \{\,\}$$

Map domain restriction is defined:

$$s \lhd m \quad \triangleq \quad \{d \mapsto m(d) \mid d \in (s \cap \mathbf{dom}\, m)\}$$

and for any map:

$$(\mathbf{dom}\, m) \lhd m = m$$

Similarly, a domain deletion operator ($\mathrel{\triangleleft\!\!\!-}$), with the same type as restriction, yields those pairs whose first elements are not in the set:

$$\{a, d, e\} \mathrel{\triangleleft\!\!\!-} m_1 = \{c \mapsto 3\}$$

Map deletion is defined:

$$s \mathrel{\triangleleft\!\!\!-} m \quad \triangleq \quad \{d \mapsto m(d) \mid d \in (\mathbf{dom}\, m - s)\}$$

and for any map values:

$$\{\,\} \mathrel{\triangleleft\!\!\!-} m = m$$
$$ma \dagger mb = (\mathbf{dom}\, mb \mathrel{\triangleleft\!\!\!-} ma) \cup mb$$

A type whose values are to be maps each with maximum domain D and maximum range R is defined by:

$$T = D \xrightarrow{m} R$$

Any value of type T is a map whose (finite) domain is a *subset* of D and whose range is a *subset* of R. Thus:

$$\{a, b\} \xrightarrow{m} \{1, 2\}$$

denotes a set of maps whose *elements* are:

$$\{\,\}, \{a \mapsto 1\}, \{a \mapsto 2\}, \{b \mapsto 1\}, \{b \mapsto 2\},$$
$$\{a \mapsto 1, b \mapsto 1\}, \{a \mapsto 1, b \mapsto 2\}, \{a \mapsto 2, b \mapsto 1\}, \{a \mapsto 2, b \mapsto 2\}$$

Thus:

$$\{a \mapsto 1, b \mapsto 2\} \in (\{a, b\} \xrightarrow{m} \{1, 2\})$$

It should be clear from this example that the type (given by **map**) defines the maximum possible domain for a map. The domain operator determines the domain set of a particular map value. Thus:

$$\mathbf{dom}\,\{b \mapsto 2\} = \{b\} \subseteq \{a, b\}$$
$$\mathbf{dom}\,\{\,\} = \{\,\} \subseteq \{a, b\}$$

Because of the restriction that maps be many-to-one, the inverse of a map is not – in general – a map. Only if a map is *one-to-one* is its inverse also a map. Although it is needed less, this type can be shown by:

$$D \overset{m}{\longleftrightarrow} R$$

where:

$$(D \overset{m}{\longleftrightarrow} R) = \{m \in (D \xrightarrow{m} R) \mid \textit{is-oneone}(m)\}$$

$$\textit{is-oneone} : (D \xrightarrow{m} R) \to \mathbb{B}$$
$$\textit{is-oneone}(m) \quad \triangleq \quad \mathbf{card\,rng}\, m = \mathbf{card\,dom}\, m$$

If:

$$m \in D \overset{m}{\longleftrightarrow} R$$

then the inverse, which is defined:

$$m^{-1} = \{r \mapsto d \mid d \in \mathbf{dom}\, m \wedge r = m(d)\}$$

is of type:

$$m^{-1} \in R \overset{m}{\longleftrightarrow} D$$

Exercise 6.1.1 Given:

$$m_1 = \{a \mapsto x, b \mapsto y, c \mapsto x\}$$
$$m_2 = \{b \mapsto x, d \mapsto x\}$$

what is the value (if defined) of:

$m_1(c)$
$\mathbf{dom}\ m_1$
$\mathbf{rng}\ m_2$
$m_1(x)$

$m_1 \dagger m_2$
$m_2 \dagger m_1$
$m_1 \cup m_2$

$\{a, e\} \lhd m_1$
$\{d, e\} \mathbin{\lhd\!\!\!-} m_2$

Exercise 6.1.2 Complete the following expressions (m_i are arbitrary maps):

$m \dagger \{\,\} = ?$
$\{\,\} \dagger m = ?$
$m_1 \dagger (m_2 \dagger m_3) = (m_1 \dagger m_2)\ ?\ m_3$
$\mathbf{dom}\,(m_1 \dagger m_2) = ?\ m_1\ ?\ \ ?\ m_2$
$\mathbf{rng}\,(m_1 \dagger m_2) = ?$
$\mathbf{dom}\,\{x \mapsto f(x) \mid p(x)\} = ?$
$\mathbf{rng}\,(m_1 \dagger m_2)\ ?\ (\mathbf{rng}\ m_1 \cup \mathbf{rng}\ m_2)$

Exercise 6.1.3 Sketch a map value ($\in$ *Floor* $\xrightarrow{m}$ *Roomno*-**set**) which shows which rooms are on which floors of the hotel mentioned in Exercise 5.1.4 on page 122.

Exercise 6.1.4 The reader should now look back at the introductory example of the equivalence relation specification built on *Partrep*. To check the understanding of the way maps are used:

- Specify, on *Partrep*, the *ELS* and *ADD* operations of Section 4.4.
- Reformulate *post-EQUATE* in a way which leaves open the choice of whether the key of e_1 or e_2 is used in the update.
- Respecify the *GROUP* and *EQUATE* operations (as in Exercise 4.4.6 on page 111) to take sets as arguments.

A model of maps

In the description of the notation given above, all of the operators are defined formally except **dom** and *application*: the other operators are defined in terms of these two. The reliance on explanation by examples can also be eliminated for these basic operators. The general style of specification in this book is to provide a model for any new data type; the model being defined using data types which are already understood. Maps can be defined in this way. The essence of the definition is to find a model for ordered pairs. If a pair is formed by the function *pr* and *first* and *second* are functions which decompose a pair, the key properties are:

$$\begin{array}{l} \mathit{first}(pr(a,b)) = a \\ \mathit{second}(pr(a,b)) = b \\ (pr(a,b) = pr(c,d)) \;\Leftrightarrow\; (a = c \wedge b = d) \end{array}$$

Either of the data type constructors from the preceding chapters can be used to construct a suitable model. Using composite objects, for given types D and R:

$$\begin{array}{lll} \mathit{Pair} :: & \mathit{first} & : D \\ & \mathit{second} & : R \end{array}$$

This satisfies the required properties with:

$$pr(a,b) \;\;\triangleq\;\; \mathit{mk\text{-}Pair}(a,b)$$

Notice that, by choosing the selectors appropriately, the decomposition functions come automatically.

It is also possible to model pairs solely in terms of sets – though this takes some thought. In order to be able to decompose the pair and to obtain the uniqueness property, it is necessary to define:

$$pr(a,b) \;\;\triangleq\;\; \{\{a\},\{a,b\}\}$$

There is a problem with naming[2] the results of the decomposition functions. This is overcome here by writing implicit specifications:

$\mathit{first}\ (p\colon \mathit{Pair})\ v\colon D$
post $\{v\} \in p$

$\mathit{second}\ (p\colon \mathit{Pair})\ v\colon R$
post $\exists u \in \bigcup p \cdot p = \{\{u\},\{u,v\}\}$

If the reader finds these definitions contorted, a few moments should be spared trying out values like:

$$pr(1,1)$$

[2] Here again, the iota (ι) operator could be used in a direct definition.

Either of these models suffices and only the properties of pairs are important. A map can be modelled by a set of pairs in which no two elements have the same *first* value:

$$Map = Pair\text{-}\mathbf{set}$$
$$\mathbf{inv}\ (s) \triangleq \forall p_1, p_2 \in s \cdot p_1 = p_2 \vee \mathit{first}(p_1) \neq \mathit{first}(p_2)$$

It is then straightforward to define:

$$\mathbf{dom}\, m = \{\mathit{first}(p) \mid p \in m\}$$

Application is again defined implicitly but, because it is an infix operator, this is written:

$$m(v) = r \;\Rightarrow\; \exists p \in m \cdot v = \mathit{first}(p) \wedge r = \mathit{second}(p)$$

All of the map notation has now been defined in terms of other types and thus, in some sense, it could be avoided by writing everything in terms of one of the models of *Pair*. As subsequent examples show, however, the map notation is one of the main tools for achieving concise specifications and it is much more convenient to use the special operators.

Exercise 6.1.5 In the text of the chapter, operators like **rng** and $\cup$ are defined in terms of **dom** and *application*. Redefine all of the map operators directly in terms of sets of pairs (use the composite object model of *Pair*).

6.2 Reasoning about maps

Map induction

As with other data types, the interesting proofs about maps require induction. It would be possible to conduct such proofs by using set induction on the domain of the map. Rather than do this, specific induction rules are given for maps. As above, these rules rely on the operators which generate finite maps. The ones chosen are very like those for sets. The empty map ($\{\,\}$) is a map and there is a (ternary) operator ($\odot$) which inserts one new pair into a map (its signature is: $D \times R \times (D \xrightarrow{m} R) \to (D \xrightarrow{m} R)$). A few detailed points are worth making here. The 'pun' on $\{\,\}$ being both the empty set and the empty map should cause no confusion. It would be more confusing to index each value with its type (although they are coded differently in the LaTeXsource files!). Furthermore, the insertion operator ($\odot$) is used only in the definitions of – and proofs about – the normal map operators: the specifications in Section 6.3 and subsequent chapters use the normal operators which are introduced in the previous section.

The generators provide an intuitive representation for any finite map:

$$\{d_1 \mapsto r_1\} \odot (\ldots \odot \{\,\})$$

The absorption and commutativity properties are slightly different from those for sets.

Axiom 6.1 Of two insertions for the same key, only the outer one has effect:

$$\boxed{\odot\text{-}pri}\ \frac{d \in D;\ r_1, r_2 \in R;\ m \in D \xrightarrow{m} R}{\{d \mapsto r_1\} \odot (\{d \mapsto r_2\} \odot m) = \{d \mapsto r_1\} \odot m}$$

Axiom 6.2 However, for different keys, insertions can be commuted:

$$\boxed{\odot\text{-}comm}\ \frac{d_1, d_2 \in D;\ r_1, r_2 \in R;\ m \in D \xrightarrow{m} R;\ d_1 \neq d_2}{\{d_1 \mapsto r_1\} \odot (\{d_2 \mapsto r_2\} \odot m) = \{d_2 \mapsto r_2\} \odot (\{d_1 \mapsto r_1\} \odot m)}$$

The intuitive representation given above is, therefore, not unique.

Axiom 6.3 (*Map-ind*) More important for the current purpose is the fact that the full induction rule reflects the absorption:

$$\boxed{\textit{Map-ind}}\ \frac{\begin{array}{l} p(\{\,\}); \\ d \in D,\ r \in R,\ m \in (D \xrightarrow{m} R),\ p(m),\ d \notin \mathbf{dom}\, m \vdash \\ \hfill p(\{d \mapsto r\} \odot m) \end{array}}{m \in (D \xrightarrow{m} R) \ \vdash\ p(m)}$$

Thus it is necessary to prove that a property holds for the empty map and that it inherits over insertion in order to conclude that the property holds for any map. The final hypothesis of the induction step shows that any map can be generated with no key occurring more than once. However, one of the operators which is discussed below is the domain operator; in proving its properties the final hypothesis is not used.

Map application is defined over the generators. The rules are given here less formally than for sets (i.e. types are not shown as antecedents in the inference rules – they are suggested by the choice of identifiers).

$$(\{d \mapsto r\} \odot m)(d) = r$$
$$d_2 \in \mathbf{dom}\, m \ \vdash\ d_1 \neq d_2 \ \Rightarrow\ (\{d_1 \mapsto r\} \odot m)(d_2) = m(d_2)$$

These rules do not permit the empty map to be applied to any value.

Proofs about override

The override operator can be defined in terms of the generators.

Rule 6.4 ($\dagger$-b) The base case:

$$\boxed{\dagger\text{-}b}\ \frac{m \in (D \xrightarrow{m} R)}{m \dagger \{\,\} = m}$$

Rule 6.5 ($\dagger$-i) The induction case:

$$\boxed{\dagger\text{-}i}\ \frac{d \in D;\ r \in R;\ m_1, m_2 \in (D \xrightarrow{m} R)}{m_1 \dagger (\{d \mapsto r\} \odot m_2) = \{d \mapsto r\} \odot (m_1 \dagger m_2)}$$

It is worth noticing that, in the case of set union, the first operand is the one which is analyzed by the cases of the definition. Here, it is necessary to analyze the second argument because of the priority given to values of the second operand. The $\dagger$-i rule essentially decomposes the second operand and generates a series of inserts around the first operand. This process could generate a string of insertions with duplicate keys (one instance coming from each operand). In conjunction with the – limited – commutativity of insertion, these can be eliminated by the

$$
\begin{array}{lll}
\textbf{from } m \in (D \xrightarrow{m} R) & & \\
1 & \{\,\} \dagger \{\,\} = \{\,\} & \dagger\text{-}b \\
2 & \textbf{from } d \in D,\ r \in R,\ m \in (D \xrightarrow{m} R),\ \{\,\} \dagger m = m & \\
2.1 & \quad \{\,\} \dagger (\{d \mapsto r\} \odot m) & \\
 & \quad\quad = \{d \mapsto r\} \odot (\{\,\} \dagger m) & \dagger\text{-}i \\
 & \textbf{infer} \quad = \{d \mapsto r\} \odot m & \text{ih2} \\
\textbf{infer } \{\,\} \dagger m = m & & \mathit{Map\text{-}ind}(1,2)
\end{array}
$$

Lemma 6.6

absorption rule ($\odot$-*pri*) for keys given above.

Lemma 6.6 The first proof about maps (see page 150) shows that the empty map is absorbed also when used as left operand of override.

$$\boxed{\text{L6.6}}\ \frac{m \in (D \xrightarrow{m} R)}{\{\,\} \dagger m = m}$$

Lemma 6.7 ($\dagger$-*ass*) The associativity of override:

$$\boxed{\dagger\text{-}\mathit{ass}}\ \frac{m_1, m_2, m_3 \in (D \xrightarrow{m} R)}{m_1 \dagger (m_2 \dagger m_3) = (m_1 \dagger m_2) \dagger m_3}$$

is also proved on page 151.

From the development of sets, the next property to consider is commutativity. It is made clear in Section 6.1 that override is not commutative. Consulting the proof of the property for set union (see page 100), it can be seen that the lack of this property for override results from the restriction placed on the commutativity of insert ($\odot$-*comm*).

Lemma 6.8 A useful result is:

$$\boxed{\text{L6.8}}\ \frac{m_1, m_2 \in (D \xrightarrow{m} R);\ \mathit{is\text{-}disj}(\mathbf{dom}\, m_1, \mathbf{dom}\, m_2)}{m_1 \dagger m_2 = m_1 \cup m_2}$$

Proofs about the domain operator

The definition of the domain operator can also be given in terms of the generators.

Rule 6.9 (**dom**-b) The basis:

$$
\begin{array}{lll}
\textbf{from } m_1, m_2, m_3 \in (D \xrightarrow{m} R) & & \\
1 \quad m_1 \dagger (m_2 \dagger \{\,\}) & & \\
\qquad = m_1 \dagger m_2 & & \dagger\text{-}b \\
2 \qquad = (m_1 \dagger m_2) \dagger \{\,\} & & \dagger\text{-}b \\
3 \quad \textbf{from } d \in D,\ r \in R,\ m \in (D \xrightarrow{m} R), & & \\
\qquad m_1 \dagger (m_2 \dagger m) = (m_1 \dagger m_2) \dagger m & & \\
3.1 \qquad m_1 \dagger (m_2 \dagger (\{d \mapsto r\} \odot m)) & & \\
\qquad\quad = m_1 \dagger (\{d \mapsto r\} \odot (m_2 \dagger m)) & & \dagger\text{-}i \\
3.2 \qquad\quad = \{d \mapsto r\} \odot (m_1 \dagger (m_2 \dagger m)) & & \dagger\text{-}i \\
3.3 \qquad\quad = \{d \mapsto r\} \odot ((m_1 \dagger m_2) \dagger m) & & \text{ih3} \\
\quad \textbf{infer} \quad = (m_1 \dagger m_2) \dagger (\{d \mapsto r\} \odot m) & & \dagger\text{-}i \\
\textbf{infer } m_1 \dagger (m_2 \dagger m_3) = (m_1 \dagger m_2) \dagger m_3 & & \mathit{Map\text{-}ind}(2,3)
\end{array}
$$

Lemma 6.7: $\dagger$-*ass*

$$\boxed{\textbf{dom}\text{-}b}\ \frac{}{\textbf{dom}\,\{\,\} = \{\,\}}$$

Rule 6.10 (**dom**-i) The inductive step:

$$\boxed{\textbf{dom}\text{-}i}\ \frac{d \in D;\ r \in R;\ m \in (D \xrightarrow{m} R)}{\textbf{dom}\,(\{d \mapsto r\} \odot m) = \{d\} \cup \textbf{dom}\, m}$$

Notice how the insert case relies on the absorption property of set union.

Lemma 6.11 The relationship between the domain and override operators:

$$\boxed{\text{L6.11}}\ \frac{m_1, m_2 \in (D \xrightarrow{m} R)}{\textbf{dom}\,(m_1 \dagger m_2) = \textbf{dom}\, m_1 \cup \textbf{dom}\, m_2}$$

is proved on page 152.

The development of the results for maps is – given an understanding of the proofs about sets – routine. A number of further results are considered in the exercises.

Exercise 6.2.1 The proofs in this section are presented in less detail than in earlier chapters. In particular, note all of the line numbers are referenced. To show that the process of completing such proof sketches is made possible by their overall structure, complete the details of the proof of Lemma 6.7 on page 151.

$$
\begin{array}{llr}
\textbf{from } m_1, m_2 \in (D \xrightarrow{m} R) & & \\
1 \quad \textbf{dom}\,(m_1 \dagger \{\,\}) & & \\
\qquad\quad = \textbf{dom}\, m_1 & & \dagger\text{-}b \\
2 \qquad\quad = \textbf{dom}\, m_1 \cup \{\,\} & & \text{L4.5} \\
3 \qquad\quad = \textbf{dom}\, m_1 \cup \textbf{dom}\,\{\,\} & & \textbf{dom}\text{-}b \\
4 \quad \textbf{from } d \in D,\ r \in R,\ m \in (D \xrightarrow{m} R), & & \\
\qquad\quad \textbf{dom}\,(m_1 \dagger m) = \textbf{dom}\, m_1 \cup \textbf{dom}\, m & & \\
4.1 \qquad \textbf{dom}\,(m_1 \dagger (\{d \mapsto r\} \odot m)) & & \\
\qquad\qquad = \textbf{dom}\,(\{d \mapsto r\} \odot (m_1 \dagger m)) & & \dagger\text{-}i \\
4.2 \qquad\quad = \textbf{dom}\,(m_1 \dagger m) \cup \{d\} & & \textbf{dom}\text{-}i \\
4.3 \qquad\quad = \textbf{dom}\, m_1 \cup \textbf{dom}\, m \cup \{d\} & & \text{ih4} \\
\qquad \textbf{infer} \quad = \textbf{dom}\, m_1 \cup \textbf{dom}\,(\{d \mapsto r\} \odot m) & & \textbf{dom}\text{-}i \\
\textbf{infer } \textbf{dom}\,(m_1 \dagger m_2) = \textbf{dom}\, m_1 \cup \textbf{dom}\, m_2 & & \textit{Map-ind}(3,4)
\end{array}
$$

Lemma 6.11

Exercise 6.2.2 Define, in terms of the generators for maps, the map operators ($\lhd$, $\unlhd$ and $\cup$). It will prove convenient for Exercise 6.2.3 to analyze the first operand when writing the last definition.

Exercise 6.2.3 Prove (showing any necessary assumptions):

$\{\,\} \lhd m = \{\,\}$
$m \cup \{\,\} = m$
$(m_1 \cup m_2) \cup m_3 = m_1 \cup (m_2 \cup m_3)$
$\{d \mapsto r\} \odot (m_1 \cup m_2) = m_1 \cup (\{d \mapsto r\} \odot m_2)$
(This splits out the equivalent of the lemma used in set union.)
$m_1 \cup m_2 = m_2 \cup m_1$
$m_1 \dagger m_2 = m_1 \cup m_2$

Exercise 6.2.4 (*) Develop further results about map operators including links to application.

Exercise 6.2.5 (*) Prove the properties of Exercise 4.4.5 on page 111 about the equivalence relation specification on the *Partrep* model.

6.3 Specifications

Bank example

It is claimed above that maps are the most ubiquitous of the basic data types. In order to indicate why this is so, a simple bank system is specified: the need to locate information by keys is typical of many computing applications. The example is also just complicated enough to rehearse some arguments which must be considered when choosing a model to underlie a specification. This is done on the level of alternative states before the operations are specified in detail.

The customers of the bank to be modelled are identified by customer numbers (Cno); accounts are also identified by numbers ($Acno$). One customer may have several accounts whose balances must be kept separately. A customer has an overdraft limit which applies to each account – a credit in one account cannot be set against a debit elsewhere.

There are, then, two sorts of information to be stored for each customer: the relevant overdraft and the balance information. Both pieces of information can be located by maps whose domains are customer numbers. But should there be one map or two? There are advantages in either solution. Separating the maps into:

$$\begin{aligned} &odm\colon Cno \xrightarrow{m} Overdraft \\ &acm\colon Cno \xrightarrow{m} \ldots \end{aligned}$$

makes it possible for some operations to reference (in their **ext** clause) only one of the maps. With separate maps, however, there is the need to define a data type invariant which requires that the domains of the two maps are always equal. The need for this invariant is avoided by using one map to composite objects:

$$Bank = Cno \xrightarrow{m} Acinf$$

$$\begin{aligned} Acinf \;::\; & od \;:\; Overdraft \\ & ac \;:\; Acno \xrightarrow{m} Balance \end{aligned}$$

$$\textbf{inv } (mk\text{-}Acinf(od, m)) \triangleq \forall acno \in \textbf{dom } m \cdot -od \leq m(acno)$$

$$Overdraft = \mathbb{N}$$

$$Balance = \mathbb{Z}$$

Invariants, as seen above, can complicate the satisfiability proof obligation. It is therefore worth avoiding gratuitous complexity and the second model is used here.[3]

Before considering other general issues raised by this specification, some minor points about interpretation should be cleared up. Both *Overdraft* and *Balance* concern sums of money. The temptation to treat these as real numbers should be resisted. Although most currencies do have fractional parts, π is an unusual balance! The fractions are there for human use and a whole number of the lowest denomination is clearly appropriate in a computer system. *Balances* can be negative – it is necessary to choose how to show overdrafts. The decision here can be seen clearly from the invariant on *Acinf* (representing the overdraft information as a minimum balance would be a possibility which would avoid a minus sign).

A larger and more general point surrounds the uniqueness of account numbers. Most banks make account numbers unique to a customer. An invariant can be used to show that no two different customers can have the same account number:

$$
\begin{aligned}
& \mathit{Bank} = \mathit{Cno} \xrightarrow{m} \mathit{Acinf} \\
& \textbf{inv}\ (m) \triangleq \\
& \quad \forall \mathit{cno}_1, \mathit{cno}_2 \in \textbf{dom}\ m \cdot \\
& \qquad \mathit{cno}_1 \neq \mathit{cno}_2 \ \Rightarrow\ \mathit{is\text{-}disj}(\textbf{dom}\ \mathit{ac}(m(\mathit{cno}_1)), \textbf{dom}\ \mathit{ac}(m(\mathit{cno}_2)))
\end{aligned}
$$

However, this suggests that the account information could be organized in a totally different way. Consider:

$$
\begin{aligned}
& \mathit{Bank} :: \mathit{am} \ : \mathit{Acno} \xrightarrow{m} \mathit{Acdata} \\
& \qquad\quad\ \mathit{odm} : \mathit{Cno} \xrightarrow{m} \mathit{Overdraft} \\
& \textbf{inv}\ (\mathit{mk\text{-}Bank}(\mathit{am}, \mathit{odm})) \triangleq \\
& \quad \forall \mathit{mk\text{-}Acdata}(\mathit{cno}, \mathit{bal}) \in \textbf{rng}\ \mathit{am} \cdot \mathit{cno} \in \textbf{dom}\ \mathit{odm} \wedge \mathit{bal} \geq -\mathit{odm}(\mathit{cno})
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{Acdata} :: \mathit{own} : \mathit{Cno} \\
& \qquad\qquad\ \mathit{bal} \ : \mathit{Balance}
\end{aligned}
$$

The invariant is not too complex and the many-to-one relationship between accounts and customers has been fitted naturally onto a map. This model looks plausible enough to justify attempting some operation specifications.

The operation to introduce a new customer into the system can be specified:

$$
\begin{aligned}
& \mathit{NEWC}\ (\mathit{od}: \mathit{Overdraft})\ r: \mathit{Cno} \\
& \textbf{ext wr}\ \mathit{odm} \ : \ \mathit{Cno} \xrightarrow{m} \mathit{Overdraft} \\
& \textbf{post}\ r \notin \textbf{dom}\ \overleftarrow{\mathit{odm}} \wedge \mathit{odm} = \overleftarrow{\mathit{odm}} \cup \{r \mapsto \mathit{od}\}
\end{aligned}
$$

[3]Once the material in Chapter 8 on relating models is understood, it is possible to work with more than one model in the case where advantages of different contending models are desired (one *can* 'have one's cake and eat it'!).

Notice that this operation *allocates* the new customer number. It is also worth observing that both *post-NEWC* and *inv-Bank* rely on the LPF. Since many map operators are partial, the reliance on the non-strict propositional operators is even greater than in earlier chapters.

An operation to introduce a new account is:

$$
\begin{array}{l}
NEWAC\ (cu\colon Cno)\ r\colon Acno \\
\textbf{ext rd } odm\ :\ Cno \xrightarrow{m} Overdraft, \\
\quad\ \textbf{wr } am\ \ :\ Acno \xrightarrow{m} Acdata \\
\textbf{pre } cu \in \textbf{dom } odm \\
\textbf{post } r \notin \textbf{dom } \overleftarrow{am} \wedge am = \overleftarrow{am} \cup \{r \mapsto mk\text{-}Acdata(cu, 0)\}
\end{array}
$$

Both of the foregoing operations trivially preserve the invariants. A simple enquiry operation is:

$$
\begin{array}{l}
ACINF\ (cu\colon Cno)\ r\colon Acno \xrightarrow{m} Balance \\
\textbf{ext rd } am\ :\ Acno \xrightarrow{m} Acdata \\
\textbf{post } r = \{acno \mapsto bal(am(acno)) \mid acno \in \textbf{dom } am \wedge own(am(acno)) = cu\}
\end{array}
$$

The chosen model stands up to the test of defining these operations. Things are rarely so easy and it is only the restriction to a very simplified system which gives this slightly unrealistic outcome. In large specifications, the writer must be prepared to revise the underlying model. Time spent in ensuring that the state matches the problem can lead to a vastly clearer specification than results from simply using one's first guess.

There is also another trade-off which is worth mentioning here. As richer sets of operations are required, it often becomes tempting to add redundant information into the state to shorten their specifications. This redundancy would of course result in further data type invariants and is to be avoided. It is in general better to define auxiliary functions which extract the necessary information from a minimal state.

Exercise 6.3.1

For the banking system specify operations which:

- close an account;
- remove a customer;
- transfer money between accounts;
- change an overdraft limit.

What changes need to be made to the model if each account has a separate overdraft limit? The informal descriptions of each of these operations can be interpreted in different ways – record any assumptions which are made in formalizing the specification.

Specifying bags

The next specification is of a different type. The preceding section showed that maps can be modelled on other types. Here, another type is modelled on maps. A *bag* (sometimes known as a *multiset*) can contain multiple elements but the order of elements is not preserved. Bags thus share the unordered property with sets and the possibility to store duplicates with sequences. The model of a bag (over some set X) is:

$$Bag = X \xrightarrow{m} \mathbb{N}_1$$

This can be viewed as associating the multiplicity with each element which has a non-zero multiplicity. The initial object – the empty bag – is:

$$b_0 = \{\,\}$$

Clarity can be heightened in this specification if an auxiliary function (mpc) is identified to compute (possibly zero) multiplicities.

$$mpc : X \times Bag \to \mathbb{N}$$
$$mpc(e, m) \quad \triangleq \quad \textbf{if } e \in \textbf{dom}\, m \textbf{ then } m(e) \textbf{ else } 0$$

The operation which shows how many occurrences of an element are in a bag is specified:

$COUNT\ (e{:}\,X)\ c{:}\,\mathbb{N}$
ext rd $b\ :\ Bag$
post $c = mpc(e, b)$

An operation to update a bag is specified:

$ADD\ (e{:}\,X)$
ext wr $b\ :\ Bag$
post $b = \overleftarrow{b} \dagger \{e \mapsto mpc(e, \overleftarrow{b}) + 1\}$

Exercise 6.3.2 Specify an operation to remove an occurrence of an element from a bag and show that it is satisfiable. (Hint: notice the range of Bag).

Describing virtual storage

As an example of the use of maps in describing a feature of machine architecture, the concept known as *virtual store* is considered. A virtual store is one which provides multiple users each with an apparent addressing space larger than the real store which is actually available to the user – perhaps even larger than the real store of the whole machine. This is achieved by paging inactive portions of store onto a backing store with slower access. The penalty is, of course, that a reference to a page which is not in fast store must be delayed while the page fault is handled.

This specification provides a good example of how abstraction can be used to explain concepts in an orderly way. The first step is to obtain a clear understanding of the basic role of store. This has nothing, as yet, to do with virtual store. The following should be easily understood by the reader:

$$Store = Addr \xrightarrow{m} Val$$

$RD\ (a{:}\ Addr)\ v{:}\ Val$
ext rd $s\ :\ Store$
pre $a \in \mathbf{dom}\ s$
post $v = s(a)$

There is an overhead in a virtual store system: the current position (i.e. in fast or slow store) of each addressable value has to be tracked. In order to reduce this overhead, addresses are grouped into pages which are always moved between levels of store as a unit. The *Addr* set has not so far been defined. It is now assumed to contain a page number and an offset (i.e. position within its page) and a *Page* maps *Offset* to *Val*:

$$\begin{aligned} Addr\ ::\ & p\ :\ Pageno \\ & o\ :\ Offset \end{aligned}$$

$Page = Offset \xrightarrow{m} Val$
inv $(m) \triangleq \mathbf{dom}\ m = Offset$

The invariant (*inv-Page*) records that the domain of any particular $m \in (Offset \xrightarrow{m} Val)$ is a subset of *Offset*. The virtual store system can now be defined to have front and backing stores, each of which contain pages:

$$\begin{aligned} Vstore\ ::\ & fs\ :\ Pageno \xrightarrow{m} Page \\ & bs\ :\ Pageno \xrightarrow{m} Page \end{aligned}$$
inv $(mk\text{-}Vstore(fs, bs)) \triangleq is\text{-}disj(\mathbf{dom}\ fs, \mathbf{dom}\ bs)$

The read operation can be respecified on *Vstore*. At this level, the concept of page faulting is introduced by showing that the relevant page must be in *fs* after the read operation. Any consideration of a specific algorithm (e.g. least recently used) to choose which page to move out is deferred. The post-condition only shows that no pages are lost and leaves open how much paging activity occurs. This non-determinism is being used as an abstraction to postpone design decisions.

$$
\begin{array}{l}
RDVS\ (a{:}\ Addr)\ v{:}\ Val \\
\textbf{ext wr } fs\ :\ Pageno \xrightarrow{m} Page, \\
\qquad \textbf{wr } bs\ :\ Pageno \xrightarrow{m} Page \\
\textbf{pre } p(a) \in (\textbf{dom}\, fs \cup \textbf{dom}\, bs) \\
\textbf{post } fs \cup bs = \overleftarrow{fs} \cup \overleftarrow{bs} \wedge p(a) \in \textbf{dom}\, fs \wedge v = fs(\,p(a))(o(a))
\end{array}
$$

There are systems in which a very clean abstraction can be given of nearly all of the functionality but where some detail distorts the final model. If one is involved in designing such an architecture, one can use this as a prompt to check whether the complexity could be avoided. If an established architecture is being described, there is no choice but to accept the extra complexity in the final model. This virtual store system provides a basis for an example. Virtual store systems in actual computers need many extra features: it is often possible to lock pages into fast store; some operations might allow access to values which cross the page boundaries. In such a case, it is good practice to record the simplified versions so as to convey the basic concepts. For some architectures this process of approximating to the final functionality can require several stages but can enormously help the comprehension of systems whose entire 'architecture' is opaque.

LISP-like lists

Chapter 5 introduces various forms of lists as occur in list-processing languages. The most LISP-like of these is shown (cf. Exercise 5.1.6 on page 123) as:

$$Pllist = [Node]$$

$$
\begin{array}{lll}
Node\ ::\ car & : & Pllist \cup \mathbb{N} \\
 cdr & : & Pllist \cup \mathbb{N}
\end{array}
$$

This fails to reflect the possibility – which exists in most dialects of LISP – that sub-lists are shared. Handling this possibility is an example of the need to introduce an intermediate link. Thus, one model which covers sharing is:

$$
\begin{array}{lll}
Lisp1\ ::\ l & : & Lisplist \\
 nm & : & Nid \xrightarrow{m} Node
\end{array}
$$

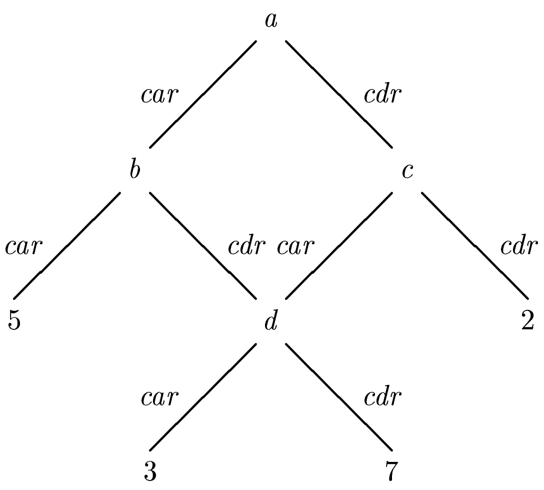

Figure 6.2 LISP list

$$Lisplist = [Nid]$$

$$\begin{aligned} Node \ :: \ & car \ : \ Lisplist \cup \mathbb{N} \\ & cdr \ : \ Lisplist \cup \mathbb{N} \end{aligned}$$

Given the basic idea of intermediate links, there are various ways in which it can be employed. It would, for example, be possible to define:

$$\begin{aligned} Lisp2 \ :: \ & carrel \ : \ Nid \xrightarrow{m} (Nid \cup \mathbb{N}) \\ & cdrrel \ : \ Nid \xrightarrow{m} (Nid \cup \mathbb{N}) \end{aligned}$$

Figure 6.2 pictures a structure. The two possible representations are:

$$\begin{aligned} mk\text{-}Lisp1(a, \{ & a \mapsto mk\text{-}Node(b, c), \\ & b \mapsto mk\text{-}Node(5, d), \\ & c \mapsto mk\text{-}Node(d, 2), \\ & d \mapsto mk\text{-}Node(3, 7)\}) \end{aligned}$$

$$\begin{aligned} mk\text{-}Lisp2(& \{a \mapsto b, b \mapsto 5, c \mapsto d, d \mapsto 3\}, \\ & \{a \mapsto c, c \mapsto 2, b \mapsto d, d \mapsto 7\}) \end{aligned}$$

The drawback of the second is the need for a relatively complicated invariant.

Well-founded relations

Sections 4.4 and 6.1 (and Chapter 11) address various representations of equivalence relations. General relations cannot be represented so compactly. The obvious model for general relations over D is sets of *Pairs*.

$$Rel = Pair\text{-}\mathbf{set}$$

$$\begin{aligned} Pair :: \ & f : D \\ & t : D \end{aligned}$$

Such a relation is said to be 'over D' because both the domain and range elements are chosen from that set. The claim that a particular pair of (D) elements stand in the R relation is written $e_1 R e_2$. These more general relations also have interesting properties: the topic of *well-foundedness* affects several examples below and has a key part to play in the proof obligations (cf. Chapter 10) for loop constructs. It is therefore worth spending a little time on the topic. Intuitively, a well-founded relation is one which has no loops. For R to be well-founded, eRe must obviously be prohibited but so also must any indirect loops like $e_1 R e_2$ and $e_2 R e_1$. In order to capture this with a predicate, one might try to trace along the relation collecting the elements that are encountered. This can be done but some care is necessary in order to make sure that the function does not become undefined in precisely the cases where the invariant should be false. Of the alternative approaches, the most straightforward is to require that, in any non-empty subset of the (potential) domain of the relation, there must be an element which is not related to an element in that subset. Thus:

$$\begin{aligned} & \forall s \subseteq D \cdot \\ & \qquad s \neq \{\,\} \ \Rightarrow \ \exists e \in s \cdot \neg\,(\exists e' \in s \cdot eRe') \end{aligned}$$

Observe that:

$$\{mk\text{-}Pair(i, i-1) \mid i \in \mathbb{N}_1\}$$

is well-founded (over $\mathbb{N}$), as also is:

$$\bigcup\{\{mk\text{-}Pair(i, j) \mid j \in \mathbb{N} \wedge j < i\} \mid i \in \mathbb{N}_1\}$$

The concept of well-foundedness plays a significant part in other branches of mathematics and it is interesting to compare the above definition with the more common mathematical definition:

$$\neg\,(\exists f : \mathbb{N} \to D \cdot \forall i \in \mathbb{N} \cdot f(i) R f(i+1))$$

This is a direct way of stating that there must be no infinite descending paths but it does require the use of higher-order quantification over functions.

Exercise 6.3.3 (*) An alternative model for relations (over D) is to view them as:

$$Rel = D \xrightarrow{m} D\text{-}\mathbf{set}$$

Define the concept of well-foundedness over this model.

Other applications

Exercise 6.3.4 Repeat Exercise 4.4.3 on page 108 using a map as a state:

$$Studx = Studnm \xrightarrow{m} \{\text{YES}, \text{NO}\}$$

What is the advantage of this state?

Exercise 6.3.5 Write the specification of a system which keeps track of which rooms people at a conference are in. Assume that operations *ARRIVE*, *MOVE* and *WHO* (giving all names in a given room) are automatically triggered.

Exercise 6.3.6 Assume that a state is available for a hotel system which shows the set of possible room numbers and the current occupancy:

$$\begin{array}{lll} Hotel :: rooms & : & Roomno\text{-}\mathbf{set} \\ occupancy & : & Roomno \xrightarrow{m} Name \end{array}$$
$$\mathbf{inv}\ (mk\text{-}Hotel(rms, occ)) \triangleq \mathbf{dom}\ occ \subseteq rms$$

Specify some useful operations such as allocating a room, checking out and determining if there are empty rooms.

Exercise 6.3.7 A simple 'bill of materials' system uses a database which, for each assembly, keeps track of the immediate components or sub-assemblies required in its construction. In this first – simplified – system, no attempt is made to record the number of each component required. Some way is needed of distinguishing basic components (no sub-assemblies). An 'explosion' can trace recursively from some assembly down to its basic components.

- Define a suitable data type with invariant for the bill of materials. (Hint: use well-founded relations.)
- Define a function which shows all sub-assemblies and components required to produce some given assembly.

- Define a function similar to the preceding one which yields only the basic components required.
- Specify an operation (say, *WHEREUSED*) which looks up in the database all of the assemblies which need a given part number as an immediate component.

Exercise 6.3.8 (*) Write a specification for a bill of materials system which counts the number of required components. Obviously, the basic data type must include the number of components per part. Furthermore, the required number of parts must be computed by multiplying the number of assemblies required by the number of components. This, and the requirement to sum such counts, will best be achieved by developing some theory of such maps.

Exercise 6.3.9 (*) Specify some operations relating to a database for an employment agency. The database should record people and their skills (more than one per person) as well as the required skills for available jobs. Operations should include showing people suitable for jobs and various updates.

7

Sequence Notation

> Various models of the same objects are possible, and these may differ in various respects. We should at once denote as inadmissible all models which contradict our laws of thought. We shall denote as incorrect any permissible models, if their essential relations contradict the relations of the external things. But two permissible and correct models of the same external objects may yet differ in respect of appropriateness. Of two models of the same object ... the more appropriate is the one which contains the smaller number of superfluous or empty relations; the simpler of the two.
> Heinrich Hertz

The concept of a sequence is both familiar to programmers and something whose manipulation is very intuitive – almost tactile. The notation developed in this chapter is, however, abstract in the sense that useful mathematical properties, rather than implementation efficiency, are taken as guidance to the choice of operators and their definitions. As a consequence, a specification written in terms of this sequence notation will need to be subjected to design steps (i.e. data reification) before it can be used as the basis for a program.

The basic collection of specification notation (sets, composite objects, maps and sequences) is completed by this chapter. It is possible to specify large systems with the help of such a tool kit; on the other hand, careful thought has to be given to the choice of an appropriate model for an application since the range of choices is now wide. Section 7.3 explores some interesting examples of such specification choices.

7.1 Notation

Sequences can be viewed as maps with a restricted domain. The advantage in recognizing sequences as a special case is that operators, such as concatenation, which are natural for sequences can be defined.

Modelling queues

The description of the notation itself is, as in previous chapters, preceded by an introductory example: this specification concerns queues. Operations are to be defined, for this first-in-first-out data structure, which enqueue, dequeue, and test whether a queue is empty. The state must record the collection of elements which are in the queue. It is possible for multiple occurrences of a *Qel* to be present and the order of elements is clearly important. These are exactly the properties of sequences. Thus:

$$Queue = Qel^*$$

The queue elements *Qel* are not further defined. *Queue* is a type whose values are sequences of *Qel*. The initial queue object is an empty sequence – sequence brackets are square – thus:

$$q_0 = [\,]$$

The operator for forming larger sequences from smaller ones is concatenation ($\frown$). Both operands of a concatenation operator must be sequences so the post-condition of the enqueue operation has to use a unit-sequence containing the new element:

$$\begin{array}{l} \mathit{ENQUEUE}\ (e\colon Qel) \\ \textbf{ext wr}\ q\ :\ Queue \\ \textbf{post}\ q = \overleftarrow{q} \frown [e] \end{array}$$

This operation requires no pre-condition. In contrast, it is only possible to remove an element from a non-empty queue. A pleasing symmetry with *post-ENQUEUE* is shown by the following specification:

$$\begin{array}{l} \mathit{DEQUEUE}\ ()\ e\colon Qel \\ \textbf{ext wr}\ q\ :\ Queue \\ \textbf{pre}\ q \neq [\,] \end{array}$$

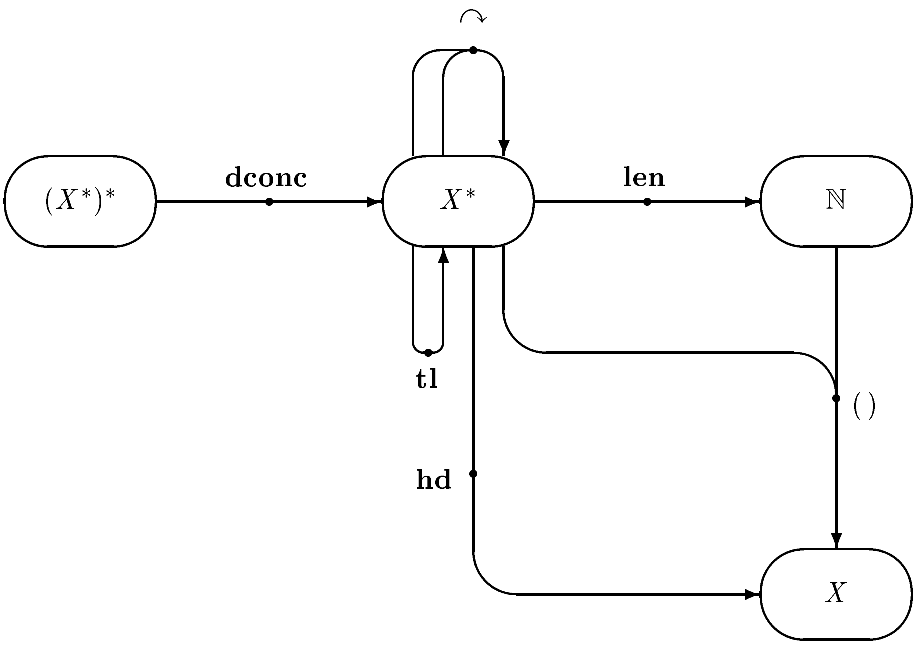

Figure 7.1 ADJ diagram of sequence operators

$\quad$ **post** $\overleftarrow{q} = [e] \frown q$

Alternatively, the post-condition could be written:

$\quad q = \mathbf{tl}\,\overleftarrow{q} \wedge e = \mathbf{hd}\,\overleftarrow{q}$

This shows the operators which yield the first element – or head – of a sequence (**hd**) and the rest of a sequence – or tail – after its head is removed(**tl**).

The operation which can be used to check whether a queue is empty is specified:

ISEMPTY () $r{:}\,\mathbb{B}$
ext rd q : *Queue*
post $r \Leftrightarrow (\mathbf{len}\, q = 0)$

The operator which yields the length of a sequence (**len**) is used in a comparison to check for an empty sequence.

Sequence operators

The first topic to be considered in the more formal treatment of sequence notation is the creation of sequence values. As is indicated above, these are written in square brackets. With sequences, both the position of values and the occurrence of duplicate values is important, thus:

$$[b, a] \neq [a, b]$$
$$[a, b] \neq [a, b, b]$$

The examples which follow use the sequences:

$$s_1 = [b, b, c]$$
$$s_2 = [a]$$

The length operator counts the number of (occurrences of) elements in a sequence, thus:

$$\textbf{len}\, s_2 = 1$$
$$\textbf{len}\, s_1 = 3$$

and for the empty sequence:

$$\textbf{len}\, [\,] = 0$$

The signatures of the sequence operators are shown in the ADJ diagram in Figure 7.1.

Sequences can be applied to valid indices – the validity of indices can be determined via the length operator – *indexing* has the properties:

$$s \in X^* \wedge 1 \leq i \leq \textbf{len}\, s \;\Rightarrow\; s(i) \in X$$
$$s_1(1) = s_1(2) = b$$

All of the other sequence operators can be defined in terms of **len** and indexing. These two basic sequence operators can be defined if the sequence type is viewed as a particular form of map:

$$\mathit{Sequence} = \mathbb{N}_1 \xrightarrow{m} X$$
$$\textbf{inv}\ (s) \triangleq \exists n \in \mathbb{N} \cdot \textbf{dom}\, s = \{1, \ldots, n\}$$

Thus:

$$\textbf{len}\, s = \textbf{card dom}\, s$$

and sequence indexing is simply map application. The set of valid indices to a sequence is given by its domain but a special operator (**inds**) is defined:

$$\begin{aligned}
&\textbf{inds}\ s = \{1, \ldots, \textbf{len}\ s\}\\
&\textbf{inds}\ s_1 = \{1, 2, 3\}\\
&\textbf{inds}\ s_2 = \{1\}\\
&\textbf{inds}\ [\,] = \{\,\}
\end{aligned}$$

The collection of elements contained in a sequence can be determined by the **elems** operator. Naturally, the set which results from this operator loses any duplications of elements:

$$\begin{aligned}
&\textbf{elems}\ s = \{s(i) \mid i \in \textbf{inds}\ s\}\\
&\textbf{elems}\ s_2 = \{a\}\\
&\textbf{elems}\ s_1 = \{b, c\}\\
&\textbf{elems}\ [\,] = \{\,\}
\end{aligned}$$

Equality over sequences must take account of the position and duplications of elements and cannot, therefore, be defined in terms of the **elems** operator. Instead:

$$s_a = s_b \;\Leftrightarrow\; \textbf{len}\ s_a = \textbf{len}\ s_b \land \forall i \in \textbf{inds}\ s_a \cdot s_a(i) = s_b(i)$$

Sequence values can be *concatenated* (i.e. joined together) by:

$$\begin{aligned}
&concat\ (s_a \colon X^*, s_b \colon X^*)\ rs \colon X^*\\
&\textbf{post}\ \textbf{len}\ rs = \textbf{len}\ s_a + \textbf{len}\ s_b \land\\
&\qquad (\forall i \in \textbf{inds}\ s_a \cdot rs(i) = s_a(i)) \land (\forall i \in \textbf{inds}\ s_b \cdot rs(i + \textbf{len}\ s_a) = s_b(i))
\end{aligned}$$

But this is written as an infix operator ($s_a \frown s_b$ rather than $concat(s_a, s_b)$). Thus:

$$\begin{aligned}
&s_1 \frown s_2 = [b, b, c, a]\\
&s_2 \frown s_1 = [a, b, b, c]\\
&s_2 \frown s_2 = [a, a]\\
&s_2 \frown [\,] = s_2
\end{aligned}$$

Notice that concatenation is neither commutative nor absorptive. A distributed concatenation operator is also available which concatenates all of the sequences within a sequence of sequences. This is defined by a recursive function:

$$\begin{aligned}
&dconc : (X^*)^* \to X^*\\
&dconc(ss) \quad \triangleq \quad \textbf{if}\ ss = [\,]\ \textbf{then}\ [\,]\ \textbf{else}\ (\textbf{hd}\ ss) \frown \textbf{dconc}\ (\textbf{tl}\ ss)
\end{aligned}$$

This is written as a prefix operator ($\textbf{dconc}\ ss$ rather than $dconc(ss)$). Thus:

$$\textbf{dconc}\ [s_1, [\,], s_2, s_2] = [b, b, c, a, a]$$

The head of a non-empty sequence is given by:

$$hd\ (s \colon X^*)\ r \colon X$$

$$\begin{array}{l}\textbf{pre } s \neq [\,] \\ \textbf{post } r = s(1)\end{array}$$

Notice that this operator yields the first *element* of a sequence whereas the tail operator yields a sequence:

$$\begin{array}{l}tl\ (s{:}\, X^*)\ rs{:}\, X^* \\ \textbf{pre } s \neq [\,] \\ \textbf{post } s = [\textbf{hd}\, s] \curvearrowright rs\end{array}$$

Both are treated as operators and are thus written in the keyword fount without parentheses:

$$\begin{array}{l}\textbf{hd}\, s_1 = b \\ \textbf{hd}\, s_2 = a \\ \textbf{tl}\, s_1 = [b, c] \\ \textbf{tl}\, s_2 = [\,]\end{array}$$

A useful operator for extracting a contiguous sub-sequence (from i to j – inclusive) of a sequence is:

$$\begin{array}{l}subseq\ (s{:}\, X^*, i{:}\, \mathbb{N}_1, j{:}\, \mathbb{N})\ rs{:}\, X^* \\ \textbf{pre } i \leq j+1 \wedge i \leq \textbf{len}\, s + 1 \wedge j \leq \textbf{len}\, s \\ \textbf{post } \exists s_1, s_2 \in X^* \cdot \\ \qquad\qquad \textbf{len}\, s_1 = i - 1 \wedge \textbf{len}\, s_2 = \textbf{len}\, s - j \wedge s = s_1 \curvearrowright rs \curvearrowright s_2\end{array}$$

Although it rather overloads the parenthesis symbol, $subseq(s, i, j)$ is written as $s(i, \ldots, j)$. The pre-condition of this operation is chosen to permit the extraction of empty sequences:

$$\begin{array}{l}s_1(2, \ldots, 2) = [b] \\ s_1(1, \ldots, 3) = [b, b, c] \\ s_1(1, \ldots, 0) = [\,] \\ s_1(4, \ldots, 3) = [\,]\end{array}$$

The reader should study the other boundary conditions of this operator. Notice that:

$$\begin{array}{l}\textbf{len}\, rs = \textbf{len}\, s - (i - 1 + (\textbf{len}\, s - j)) \\ \qquad\quad = (j - i) + 1\end{array}$$

Amongst other useful properties, careful consideration of such end cases simplifies the construction of a delete function:

$$del(t, i) \quad \triangleq \quad t(1, \ldots, i-1) \curvearrowright t(i+1, \ldots, \textbf{len}\, t)$$

A sequence type, X^*, defines values of the type to be any finite sequence all of whose elements are members of X. Thus, if $X = \{a, b, c\}$, members of X^* include:

$[\,]$
s_1
$[a, a, a, a, a, a]$

Because of the possibility of duplicates, the number of potential sequences is infinite even when the base set is finite. The type X^+ excludes the empty sequence but is otherwise the same as X^*.

Exercise 7.1.1 Which of the following expressions is true (in general)?

$$s_a \curvearrowright (s_b \curvearrowright s_c) = (s_a \curvearrowright s_b) \curvearrowright s_c$$
$$s_a \curvearrowright s_b = s_b \curvearrowright s_a$$
$$s_a \curvearrowright [\,] = s_a$$
$$s_a \curvearrowright s_a = s_a$$

Exercise 7.1.2 What is the value of each of the following?

$\mathbf{tl}\,[a, b]$
$\mathbf{len}\,[[a, b], [a, b]]$
$\mathbf{hd}\,[a]$
$\mathbf{tl}\,[a]$
$\mathbf{hd}\,[[a, b], [c]]$

$\mathbf{elems}\,[a, b, a]$
$\mathbf{elems}\,[\{a\}, a, [a], a]$
$[a] \curvearrowright [a]$
$[a] \curvearrowright [[b]]$

Exercise 7.1.3 In each of the following three cases, identify a possible value for a sequence which satisfies the properties:

$\mathbf{len}\, s_a \neq \mathbf{card}\, (\mathbf{elems}\, s_a)$
$\mathbf{hd}\, s_b = [b], \mathbf{hd}\,\mathbf{tl}\, s_b = \{1\}, \mathbf{tl}\,\mathbf{tl}\, s_b = [b]$
$\mathbf{tl}\, s_c = [\mathbf{hd}\, s_c]$

Exercise 7.1.4 Define a function which determines whether a sequence has only one occurrence of each of its elements. Specify a function which, given a set, lays it out as a sequence without duplicates – in a random order.

Exercise 7.1.5 It is often useful to be able to locate things within sequences (i.e. to determine indices where values are located). Specify a function which show all indices where a value can be found:

$$allocs: X^* \times X \to \mathbb{N}_1\textbf{-set}$$

Specify a function which gives the first index where a value can be found assuming that it does occur:

$$\mathit{firstocc}: X^* \times X \to \mathbb{N}_1$$

Specify a function which locates (the first contiguous occurrence of) one sequence within another:

$$locate: X^* \times X^* \to \mathbb{N}$$

such that:

$$locate([a,b],[a,a,b,a]) = 2$$
$$locate([b,b],[a,a,b,a]) = 0$$

Exercise 7.1.6 In the text of this chapter, operators like concatenation and tail are defined via the more basic operators length and application. Redefine all of the sequence operators directly in terms of the map model.

7.2 Reasoning about sequences

Sequence induction

The theory of finite sequences is strongly related to that of (finite) sets. As the reader should by now expect, the genesis of the theory is the generator functions – here they are the empty sequence ($[\,]$) and a constructor function (*cons*) whose signature is $X \times X^* \to X^*$. The function to insert an element into a sequence is called *cons* (rather than $\odot$) because the name is familiar from list-processing languages. Thus sequence values can be created by:

$$cons(e_1, \cdots (cons(e_n, [\,])) \cdots)$$

Whereas with both sets and maps, different terms built from the constructors correspond to the same value, the expressions built from sequence constructors stand in one-to-one correspondence with the values. For sets and maps, properties were given which showed that certain terms were equal; no such properties need be given for sequences. The distinction between the theory of sequences and that of sets is that any properties which rely on the commutativity and absorption of $\odot$ do not carry over to sequences.

Axiom 7.1 (*Seq-ind*) The induction rule for sequences is, apart from the changes of symbols, the same as the first one given for sets:

$$\boxed{\textit{Seq-ind}}\ \frac{\begin{array}{l} p([\,]); \\ e \in X,\ t \in X^*,\ p(t) \ \vdash\ p(cons(e,t)) \end{array}}{t \in X^* \ \vdash\ p(t)}$$

This induction axiom – as with those above – relies on the finiteness of individual sequence values.

Proofs about operators

The definition of concatenation (over the constructors) is essentially a translation of that for set union:

Rule 7.2 ($\frown$-b) Basis:

$$\boxed{\frown\text{-}b}\ \frac{s \in X^*}{[\,] \frown s = s}$$

Rule 7.3 ($\frown$-i) Induction:

$$\boxed{\frown\text{-}i}\ \frac{e \in X;\ s_1, s_2 \in X^*}{cons(e, s_1) \frown s_2 = cons(e, s_1 \frown s_2)}$$

It should therefore be obvious that the following two lemmas hold.

Lemma 7.4 Concatenation absorbs empty sequences on the right:

$$\boxed{\text{L7.4}}\ \frac{s \in X^*}{s \frown [\,] = [\,]}$$

Lemma 7.5 ($\frown$-ass) Concatenation is associative:

$$\boxed{\frown\text{-}ass}\ \frac{s_1, s_2, s_3 \in X^*}{(s_1 \frown s_2) \frown s_3 = s_1 \frown (s_2 \frown s_3)}$$

The proofs of these are simple transliterations of the corresponding ones for sets. The next properties which are developed for set union are commutativity and absorption. These proofs rely on the corresponding properties of the insertion operator and do not therefore carry over to concatenation. In general:

$$cons(a, cons(b, s)) \neq cons(b, cons(a, s))$$
$$cons(a, cons(a, s)) \neq cons(a, s)$$

A lemma which is used in later chapters is:

Lemma 7.6 The elements collected from the concatenation of two sequences are the union of the elements of the two sequences.

$$\boxed{\text{L7.6}}\ \frac{s_1, s_2 \in X^*}{\textbf{elems}\,(s_1 \frown s_2) = (\textbf{elems}\, s_1) \cup (\textbf{elems}\, s_2)}$$

The definitions of the other operators are left to the exercises.

Axiom 7.7 (*Seq-ind*2) Once these are defined, a restatement of the induction rule for sequences is possible. The two forms of the rule correspond to the option of defining induction over the natural numbers in terms of either *succ* or *pred*.

$$\boxed{\textit{Seq-ind2}}\ \frac{p([\,]);\quad t \in X^+,\ p(\mathbf{tl}\,t) \vdash p(t)}{t \in X^* \vdash p(t)}$$

Exercise 7.2.1 Write out the proofs for Lemmas 7.4 – 7.6.

Exercise 7.2.2 Only concatenation is defined in the text of this section. Define the operators **len**, application, **hd** and **tl** over the constructors. Prove some useful results like:

$$\forall s_1, s_2 \in X^* \cdot \mathbf{len}\,(s_1 \curvearrowright s_2) = \mathbf{len}\,s_1 + \mathbf{len}\,s_2$$
$$\forall s \in X^* \cdot s = [\,] \vee cons(\mathbf{hd}\,s, \mathbf{tl}\,s) = s$$

Reversing sequences

A definition of a function which reverses a sequence is:

$$rev : X^* \to X^*$$
$$rev(s) \quad \triangleq \quad \textbf{if } s = [\,] \textbf{ then } [\,] \textbf{ else } rev(\mathbf{tl}\,s) \curvearrowright [\mathbf{hd}\,s]$$

Its properties can be given by the two rules.

Rule 7.8 The basis:

$$\boxed{\textit{rev-b}}\ \frac{}{rev([\,]) = [\,]}$$

Rule 7.9 The inductive step:

$$\boxed{\textit{rev-i}}\ \frac{e \in X;\ s \in X^*}{rev(cons(e, s)) = rev(s) \curvearrowright [e]}$$

Rule 7.10 It is useful to define unit sequences as an abbreviation:

$$\boxed{\text{R7.10}}\ \frac{e \in X}{[e] = cons(e, [\,])}$$

Lemma 7.11 An obvious property of *rev* is that applying it twice to any sequence should yield the original sequence.

$$\boxed{\text{L7.11}}\ \frac{s \in X^*}{rev(rev(s)) = s}$$

A frontal attack on this result yields a messy proof. The identification of two preliminary lemmas (see page 174) gives rise to a more readable presentation (see page 175).

from $e \in X$		
1 $rev([e])$		
	$= rev(cons(e, [\,]))$	R7.10
2	$= rev([\,]) \curvearrowright [e]$	$rev\text{-}i$
3	$= [\,] \curvearrowright [e]$	$rev\text{-}b$
infer	$= [e]$	$\curvearrowright\text{-}b$

from $s_1, s_2 \in X^*$		
1 $rev([\,] \curvearrowright s_2)$		
	$= rev(s_2)$	$\curvearrowright\text{-}b$
2	$= rev(s_2) \curvearrowright [\,]$	L7.4
3	$= rev(s_2) \curvearrowright rev([\,])$	$rev\text{-}b$
4 **from** $e \in X,\ t \in X^*,\ rev(t \curvearrowright s_2) = rev(s_2) \curvearrowright rev(t)$		
4.1 $rev(cons(e, t) \curvearrowright s_2)$		
	$= rev(cons(e, t \curvearrowright s_2))$	$\curvearrowright\text{-}i$
4.2	$= rev(t \curvearrowright s_2) \curvearrowright [e]$	$rev\text{-}i$
4.3	$= (rev(s_2) \curvearrowright rev(t)) \curvearrowright [e]$	ih4
4.4	$= rev(s_2) \curvearrowright (rev(t) \curvearrowright [e])$	$\curvearrowright\text{-}ass$
infer	$= rev(s_2) \curvearrowright rev(cons(e, t))$	$rev\text{-}i$
infer $rev(s_1 \curvearrowright s_2) = rev(s_2) \curvearrowright rev(s_1)$		$Seq\text{-}ind$(3,4)

Lemmas on *rev*

Exercise 7.2.3 Specify the function *rev* by a post-condition using quantifiers and indexing. Sketch the argument that applying *rev* twice acts as an identity function on sequences.

A *palindrome* is a word (e.g. 'dad') which is the same when it is reversed. Define a palindrome by properties over the indices and prove that the result of applying *rev* to a palindrome p is equal to p.

Exercise 7.2.4 (*) Another alternative specification of *rev* could characterize the split point implicitly and not fix that only the head is moved on each recursive call. Experiment with the development of results about such a definition. This should show that being more abstract often results in a clearer exposition.

from $t \in X^*$		
1	$rev(rev([\,])) = rev([\,])$	*rev-b*
2	$rev(rev([\,])) = [\,]$	*rev-b*
3	**from** $e \in X,\ t \in X^*,\ rev(rev(t)) = t$	
3.1	$rev(rev(cons(e, t)))$	
	$= rev(rev(t) \curvearrowright [e])$	*rev-i*
3.2	$= rev([e]) \curvearrowright rev(rev(t))$	Lemma-b
3.3	$= [e] \curvearrowright t$	Lemma-a,ih3
	infer $= cons(e, t)$	$\curvearrowright$
infer $rev(rev(t)) = t$		*Seq-ind*(2,3)

Lemma 7.11: *rev* is its own inverse

7.3 Specifications

Specifying sorting

The task of sorting provides an obvious application for the sequence notation. Suppose records are to be sorted whose structure is:

$$\begin{aligned} Rec :: \ & k \ : \ Key \\ & d \ : \ Data \end{aligned}$$

The fact that a sequence of records is ordered in ascending key order can be defined:

$$is\text{-}orderedk : Rec^* \to \mathbb{B}$$
$$is\text{-}orderedk(t) \quad \triangleq \quad \forall i, j \in \mathbf{inds}\, t \cdot i < j \ \Rightarrow \ k(t(i)) \leq k(t(j))$$

For compactness, the ordering relation on keys is written $\leq$. Because the ordering relation is transitive, it is equivalent to write:

$$is\text{-}orderedk(t) \quad \triangleq \quad \forall i \in \{1, \ldots, \mathbf{len}\, t - 1\} \cdot k(t(i)) \leq k(t(i+1))$$

Notice how the rule about universal quantification over an empty set conveniently covers unit and empty sequences. Accepting, for the moment, some intuitive notion of permutation, the specification for the sorting task can be written:

SORT ()
ext wr *rs* : Rec^*

post $\mathit{is\text{-}orderedk}(rs) \wedge \mathit{is\text{-}permutation}(rs, \overleftarrow{rs})$

Defining the concept of one sequence being a permutation of another is an interesting exercise. Clearly, if the sequences can contain duplicates, it is not enough to check that their ranges (**elems**) are equal. Nor does it cover all cases to check both **len** and **elems**. One possibility is to write *is-permutation* as a recursive function which, in the recursive case, locates and removes the element at the head of one sequence from wherever it is in the other. Such a definition is rather mechanical for a specification and would not be easy to use in subsequent proofs. A direct model of the idea of counting occurrences can be given using bags. Thus:

$$\begin{array}{l} \mathit{bagof} : X^* \to \mathit{Bag} \\ \mathit{bagof}(t) \quad \triangleq \quad \{e \mapsto \mathbf{card}\, \{i \in \mathbf{inds}\, t \mid t(i) = e\} \mid e \in \mathbf{elems}\, t\} \end{array}$$

Then:

$$\begin{array}{l} \mathit{is\text{-}permutation} : X^* \times X^* \to \mathbb{B} \\ \mathit{is\text{-}permutation}(s_1, s_2) \quad \triangleq \quad \mathit{bagof}(s_1) = \mathit{bagof}(s_2) \end{array}$$

Another possibility is to think of a permutation as inducing a one-to-one map between the two sequences:

$$\begin{array}{l} \mathit{is\text{-}permutation}(s_1, s_2) \quad \triangleq \\ \quad \mathbf{len}\, s_1 = \mathbf{len}\, s_2 \wedge \\ \quad \exists m \in \mathbb{N}_1 \xleftrightarrow{m} \mathbb{N}_1 \cdot \\ \qquad \mathbf{dom}\, m = \mathbf{rng}\, m = \mathbf{inds}\, s_1 \wedge \forall i \in \mathbf{inds}\, s_1 \cdot s_1(i) = s_2(m(i)) \end{array}$$

It is not possible to argue convincingly that one of these is better than the other for all purposes. It is, however, likely that the last one would be of more use in developing a theory of sequences. For the sorting program itself, the only properties of *is-permutation* required for most internal sorts are reflexivity, transitivity and the fact that swapping two elements creates a permutation. It is clear that these properties follow easily from the latter definition of *is-permutation*.

The specification of *SORT* is non-deterministic in that the final placing of two different records with the same key is not determined. This reflects the fact that the sorting task is described as bringing the records into key order. There are applications where a stable sort is required in which records with the same key preserve their relative order from the starting state. The specification can be modified to cover this requirement by simply adding an extra conjunct to *post-SORT* whose definition is:

$$\begin{array}{l} \mathit{is\text{-}stable} : \mathit{Rec}^* \times \mathit{Rec}^* \to \mathbb{B} \\ \mathit{is\text{-}stable}(s_1, s_2) \quad \triangleq \quad \forall \mathit{key} \in \mathit{extractks}(s_1) \cdot \mathit{sift}(s_1, \mathit{key}) = \mathit{sift}(s_2, \mathit{key}) \end{array}$$

The keys required are defined by:

$$\begin{aligned}&extractks : Rec^* \to Key\text{-}\mathbf{set}\\ &extractks(s) \quad \triangleq \quad \{k(r) \mid r \in \mathbf{elems}\, s\}\end{aligned}$$

The sub-sequence of *Recs* with a given key can be defined:

$$\begin{aligned}&sift : Rec^* \times Key \to Rec^*\\ &sift(rs, key) \quad \triangleq \quad \mathbf{if}\ rs = [\,]\\ &\qquad\qquad\qquad\quad \mathbf{then}\ [\,]\\ &\qquad\qquad\qquad\quad \mathbf{else\ if}\ k(\mathbf{hd}\, rs) = key\\ &\qquad\qquad\qquad\qquad \mathbf{then}\ [\mathbf{hd}\, rs] \curvearrowright sift(\mathbf{tl}\, rs, key)\\ &\qquad\qquad\qquad\qquad \mathbf{else}\ sift(\mathbf{tl}\, rs, key)\end{aligned}$$

Priority queues

The introductory example in Section 7.1 specified a simple first-in-first-out queue. Another form of queue which is used in computing systems relies on a priority to govern which element is dequeued. This example provides a basis for a discussion of the choices to be made in constructing a model. Assume that there is some given set *Priority* which, for conciseness, is assumed to be ordered by $\leq$. Then items in the queue might be defined:

$$\begin{aligned}Qitem :: \ & p : Priority\\ & d : Data\end{aligned}$$

Perhaps the most obvious model for the queue type itself is:

$$Qtp = Qitem^*$$

where the data type invariant (say, *is-orderedp*) would require that the priority order holds in the sequence. This would permit the operation for adding elements to the queue to be specified:

$$\begin{aligned}&ENQ\ (it{:}\ Qitem)\\ &\mathbf{ext\ wr}\ q : Qtp\\ &\mathbf{post}\ \exists i \in \mathbf{inds}\, q \cdot del(q, i) = \overleftarrow{q} \wedge q(i) = it\end{aligned}$$

Recall that the invariant can be thought of as being conjoined to the pre- and post-conditions. It is then clear that the post-condition combines two of the techniques used to achieve concise specifications. The existentially quantified expression works back from the result to the starting state – thus providing a simple description of insertion. The (implied) conjunction of *is-orderedp* with that expression captures the required specification by stating two separate properties.

The specification as it stands does not constrain the placing of queue items with equal priority. Providing this matches the requirements, the next question to ask is whether the sequence model given is the most appropriate. Why are the queue items ordered in the state? Presumably because it makes the dequeuing operation easy to specify! But this is not really a convincing argument. In fact an alternative specification could be based on sets (or, if duplicate records have to be handled, bags). Thus:

$$Qtps = Qitem\text{-}\mathbf{set}$$

The *ENQ* operation simply adds its argument to the state and the *DEQ* operation locates one of the elements with lowest priority number. With the limited repertoire of operations, it is difficult to say which is the better model, but the set model is more abstract and might be preferred.

If, however, it is required to preserve the arrival order of queue items with the same priority, it is clear that the set model cannot support the intended semantics. On the other hand, it is easy to see how to extend the post-condition of *ENQ*, as defined on sequences, to ensure correct placement. The sequence model is, however, not the only one which would cover the ordering requirement. The *ENQ* operation is easier to specify if the queues for each priority are separated:

$$Qtpm = Priority \xrightarrow{m} Data^*$$

Some decisions have to be made in this model about whether each priority always has a (possibly empty) sequence associated with it. But, on balance, the map model is the best fit to the operations. The complete set of operations would have to be agreed before a final decision were made. (One could envisage operations which force consideration of the queue as a whole – for example, operations which manipulated the priorities.)

Exercise 7.3.1 Complete the operation specifications for enqueuing, dequeuing, and testing for empty for all three of the models discussed in the text for priority queues.

Exercise 7.3.2 A stack is a last-in-first-out storage structure.

- Specify an (unbounded) stack with operations for *PUSH*, *POP* and *ISEMPTY*; also show the initial stack object.
- As above, but assume a bound (say 256) on the contents of a stack; specify an additional operation *ISFULL*.
- As above but, instead of making *PUSH* partial, arrange that pushing an element onto a full stack loses the oldest element!

- Another form of stack which has attracted some interest is known as 'Veloso's Traversable Stack'. This stack – in addition to the normal operations – can be *READ* from a point indicated by a cursor; the cursor can be *RESET* to the top of the stack or moved *DOWN* one element; the normal *POP* and *PUSH* operations can only be performed with the cursor at the top of the stack but the operations preserve this property. Specify this form of stack.

Ciphering

Another example in which some thought must be applied to the choice of model is a specification for a cipher machine. Many children play games with coding messages by, for instance, changing letter a to b, b to c, etc. Such a cipher is called monoalphabetic and is very susceptible to cryptanalysis (code breaking) by measuring the frequency of letters. A more sophisticated polyalphabetic (or Vigenère) coding is somewhat more secure. The idea of substituting one letter by another is extended so that different letters of the original message (plain text) are coded under different translations. In order that the enciphered message can be deciphered, the appropriate transliterations must be known or be computable. One way to achieve this is to have a table of translation columns each headed by a letter; a keyword is then agreed and the ith letter of the keyword indicates the column under which the ith letter of the message is to be (or was) ciphered; the keyword can be replicated if it is shorter than the message. A table for a restricted alphabet could be:

	a	b	c
a	a	c	b
b	b	a	c
c	c	b	a

The plaintext $acab$ is coded under keyword abc to $abbb$:

plaintext	a	c	a	b
keyword	a	b	$c/$	a
ciphered text	a	b	b	b

A simple frequency analysis of letters will no longer disclose the coding table since, on the one hand, different letters are translated to the same letter and, on the other hand, the same plaintext letter can be translated to different letters.

How is this polyalphabetic cipher to be specified? The regular appearance of the table above might tempt one to describe the coding by index arithmetic on a sequence of twenty-six letters. There are two reasons to resist this particular temptation: the regular tables are only a subset of the possible tables, and anyway the index arithmetic becomes very confusing. The best model of an individual column appears to be a map from letters to letters. As is discussed below, it is necessary that such a map be one-to-one. Thus:

$$Mcode = Letter \xleftrightarrow{m} Letter$$
$$\textbf{inv } (m) \triangleq \textbf{dom } m = Letter$$

The invariant ensures that there is a translation for each letter.

The whole (polyalphabetic) table can be defined:

$$Pcode = Letter \xrightarrow{m} Mcode$$
$$\textbf{inv } (m) \triangleq \textbf{dom } m = Letter$$

In practice, it is obviously desirable that *Pcode* stores different *Mcodes* for each letter – this requirement is not, however, enshrined in the invariant. A function which defines the (polyalphabetic) translation of message *ml* under key *kl* is:

$$ptrans : Letter \times Letter \times Pcode \to Letter$$
$$ptrans(kl, ml, code) \quad \triangleq \quad (code(kl))(ml)$$

The remaining hurdle, before the specification can be written, is to choose a representation for the keyword. The obvious model is a sequence of letters. The problem of sufficient replications then becomes a manipulation of indices which is made slightly messy by the fact that the sequences here are indexed from one. (The alternative, to index all sequences from zero, turns out to be just as inconvenient in other cases.) Here, indexing from zero is simulated by:

$$Key = \mathbb{N} \xrightarrow{m} Letter$$
$$\textbf{inv } (m) \triangleq \exists n \in \mathbb{N} \cdot \textbf{dom } m = \{0, \ldots, n\}$$

This ensures that a *Key* is non-empty. It could be argued that the keyword should be replicated in the state but this is not done here since it appears to make the task of designing representations unnecessarily tiresome. The final specification is then:

$$
\begin{array}{l}
CODE \ (m{:}\, Letter^*) \ t{:}\, Letter^* \\
\textbf{ext rd } c \ : \ Pcode, \\
\qquad \textbf{rd } k \ : \ Key \\
\textbf{post len } t = \textbf{len } m \wedge \\
\qquad \textbf{let } l = maxs(\textbf{dom } k) + 1 \textbf{ in} \\
\qquad \forall i \in \textbf{inds } t \cdot t(i) = ptrans(k(i \textbf{ mod } l), m(i), c)
\end{array}
$$

The specification of *DECODE* is written as a mirror image of that for *CODE*.

$DECODE\ (t: Letter^*)\ m: Letter^*$
ext rd $c\ :\ Pcode$,
  rd $k\ :\ Key$
post len $m =$ **len** $t \wedge$
  let $l = maxs($**dom** $k) + 1$ **in**
  $\forall i \in$ **inds** $t \cdot t(i) = ptrans(k(i$ **mod** $l), m(i), c)$

This shows clearly that the task of *DECODE* is to recreate the input to *CODE*. It is possible, from this requirement, to deduce the need for the invariant on *Mcode*. The correct decipherment of messages can be stated (omitting all of the quantifiers):

$$post\text{-}CODE(m, \ldots, t) \wedge post\text{-}DECODE(t, \ldots, m') \;\Rightarrow\; m = m'$$

Inspecting the two post-conditions it is clear that the length of m and m' must be the same and thus the question is pushed back to whether *ptrans* is one-to-one. The function *ptrans* simply selects a (determined) *Mcode* in either case and thus it can be seen that *Mcode* must be a one-to-one map in order to prevent, for some i, two different letters $m(i)$ and $m(j)$ from giving the same translation $t(i)$.

Exercise 7.3.3 The German cipher machine which was known as 'Enigma' achieved polyalphabetic substitution but was constructed with a reflecting property (i.e. if a was coded as n then n was coded as a). This meant that the operator performed the same operation whether coding or decoding a text. What changes does this make to the specification given above?

Exercise 7.3.4 Specify an operation which has access to a set of file names (character strings). Given the prefix of a file name as input, the operation should yield the set of matching file names.

Exercise 7.3.5 Develop operators, predicates and a theory for sequences which are (not necessarily contiguous) sub-sequences of other sequences in the sense that the former can be within the latter (e.g. $[a, b, c]$ is a sub-sequence of $[a, c, a, d, b, c, a, b]$). Refine the notation by writing specifications of a number of tasks (e.g. a function which merges two sequences, a function which finds the 'longest ascending sub-sequence' of a sequence of natural numbers).

Exercise 7.3.6 (*) A formal model of the (English) children's game of snakes and ladders can be based on sequences. Design an appropriate state and specify some operations (e.g. *MOVE*).

Exercise 7.3.7 (*) Define an abstract syntax (cf. Section 5.1) for expressions of propositional logic (there are some interesting points to be decided upon). Write a function which determines, in classical two-valued logic, whether an expression is a tautology. Implication and equivalence operators can be expanded out using their definitions. In Disjunctive Normal Form (DNF) expressions are reduced to a form which is a disjunction of conjunctions of (possibly negated) literals (E_i). Define a function which converts arbitrary propositional expressions into DNF. In terms of this limited structure define an efficient algorithm for tautology checking.

Consider the changes required to handle the LPF (cf. Section 3.3) used in this book and define a function which checks LPF propositional sequents for validity. Design an abstract syntax for proofs in the propositional calculus. There is considerable scope for experiment here and it is worth considering the need for relations. Define an abstract syntax for formulae of the predicate calculus and functions to determine the free variables of a logical expression and to apply systematic substitution.

Exercise 7.3.8 (*) Develop the state for a relational database system. Unless the reader is an expert in this area, an actual system should be used as a reference point. Focus the work on building a model for the storage of, and the type information for, relations.

8

Data Reification

More than anything else mathematics is a method.
Morris Kline

It should be clear that the construction of a formal specification can yield greater understanding of a system than is normally possible before implementation is undertaken. On larger examples the process of constructing the formal specification can also prompt consideration of questions whose resolution results in a cleaner architecture. It is, therefore, possible that the work involved in producing a formal specification would be worthwhile even if the ensuing development were undertaken using informal methods. But the remaining chapters of this book present another exciting avenue which is opened up by formal specification: a formal specification provides a reference point against which a proof can be constructed. A proof can show that a program satisfies its specification for *all* valid inputs. Clearly, no real proof could be based on an informal description whose semantics are unclear. The idea that programs can be proved to satisfy formal specifications is now well-documented in scientific papers. More interestingly, it has been shown that a design process can be based on formal specifications. The essence of such a design process is to record a design step with a series of assumptions about subsequent development (i.e. specifications of sub-components) and then to show that the design step is correct under the given assumptions. Once this has been done, the assumptions (specifications of the sub-components) are tackled. It is a crucial property of the development method presented here that each subsidiary task in development is isolated by its specification. Without such a property of isolation, a development method is open to some of the worst risks of testing: errors are detected long after they are made and work based on such mistakes must be discarded when errors are detected. The isolation property is sometimes called *compositionality*.

There are a number of ways in which the above description is over-simplified. Firstly, a development hardly ever proceeds strictly top-down. But, even if one is forced to backtrack, the eventual design will be made clearer by documentation presented in a neat hierarchy. Sub-components can also be developed bottom-up; but such sub-components will be used safely only if they are accompanied by formal specifications. Another issue which could be taken with the over-simplified description is the level of formality to be used in the design process. Any design step generates a proof obligation. Such proof obligations can be discharged completely formally and some proofs are shown below in detail. Once one knows how to conduct such proofs, the level of formality can be relaxed for most steps of design. The formal structure provides a way of giving more detail when required. A knowledge of the formal structure will itself minimize the danger of mistakes. It is, however, clear that more confidence is justified in a machine-checked formal proof than an outline correctness argument.

The process of design can be seen as making commitments. The data representation chosen is a commitment which the designer makes based on an understanding of the required operations and their relative frequencies of use. The method outlined here is not intended to help make such choices. Design relies on invention. Such invention has been 'automated' only in very narrow areas. What is provided is a notation for recording designs and the proof obligations necessary to establish their correctness (rather than their optimality). Experience has shown that the formal structure does aid designers by clarifying their choices but the case for the rigorous approach should never be construed as claiming that the design process can be automated.

The style of formal specification proposed in the preceding chapters uses (abstract) models of data types and implicit specification by pre- and post-conditions. High-level design decisions normally involve choosing the representation of data: *data reification*[1] involves the transition from abstract to concrete data types and the justification of the transition. At the end of this process, the data types are those of the implementation language but the transformations are still defined implicitly (i.e. by pre- and post-conditions). Operation decomposition – described in Chapter 10 – is the process of choosing, and justifying, a sequence

[1]The term reification is preferred here to the more widely-used word 'refinement'. Michael Jackson pointed out to the author that the latter term is hardly appropriate for the step from a clean mathematical abstraction to a messy representation dictated by a particular machine architecture. The *Concise Oxford Dictionary* defines the verb 'reify' as 'convert (person, abstract concept) into thing, materialize'.

of transformations which can be expressed in the implementation language.

In choosing the data types for a specification, the aim is that they should be as abstract as possible. Although this notion is not made precise until Section 9.3, the reader should by now have a general feel for avoidance of unnecessary details in a state. The proof obligations given in Sections 8.1 and 8.2 relate to the special case where 'bias' increases at each step of reification. This is a very common special case: designers make commitments – commitments which reflect special properties of the application and of the implementation machine. These commitments give rise to redundancy, complexity (e.g. of invariants) and efficiency! Thus the data types which result from reification tend to require long descriptions and give rise to complex operation specifications. The examples in Chapters 4 to 7 include descriptions of data types which arise in design. For example, the choice of one form of binary tree is motivated by noting that it can provide a representation of a set. In general, a representation (of one data type) is just another data type – as such it can be described by the data structuring devices used above.

The key to relating an abstract data type and its representation is a 'retrieve' function – this concept, and the first of the proof obligations, is introduced in Section 8.1. The proof obligations which concern the operations are explained in the succeeding section. Section 8.3 discusses the problems of predefined interfaces and presents some larger examples.

8.1 Retrieve functions and adequacy

Establishing a link between states

Given a specification, a designer chooses a representation which reflects implementation considerations. The notion of *satisfaction* provides a criterion by which the correctness of the choice of representation can be judged. The proof obligations, which are explained here and in Section 8.2, are based on a satisfaction relation for which an implementation must exhibit an acceptable *behaviour*. (These proof obligations reflect an extremely common special case of data reification; Section 9.3 reviews some alternatives.) In these proof obligations, it is possible to separate some questions about the reification of the state itself from consideration of the operations which are associated with the states.

Suppose that some specification uses dates ($Date$) as in Exercise 5.1.1 on page 121. A representation might be chosen which packs the date into two bytes (5 bits for the day, 4 bits for the month, 7 bits for the year – this last allowing an increment from 0 to 127 to be added to some notional base date). One could fix the relation between elements of $Date$ and the bit representation by a relation.

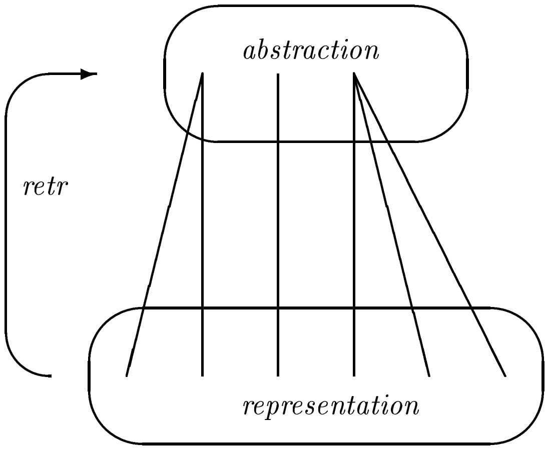

Figure 8.1 Retrieve function

The relation would be one-to-one, and this should suggest to the reader that a function could be used to record the relationship. In this simple example, there is no obvious reason to prefer one direction or the other for the function – more guidance comes from considering an example like the use of a binary tree to represent a set. The set might have been chosen in the specification because its properties were appropriate to the application being specified; a binary tree might be chosen as a representation so that a test operation could be performed efficiently for large volumes of data. In this example, each abstract set value has more than one possible representation as a tree. The relation between abstraction and representation values is one-to-many. Clearly, a one-to-many relation can be treated as a general relation. But there is also the possibility that it is treated as a function (from the 'many' to the 'one'). Is the reverse situation likely to arise? If different abstract values correspond to one concrete value, it is intuitively obvious that such values could have been merged in the abstraction. So, in the situation where the objects used in the specification were abstract enough, the many-to-one situation would not arise. Working with relations can lead to rather heavy notation. Here, the opportunity to avoid this heaviness is taken. The relationship between abstract values and their representations is expressed by a function from the latter to the former (e.g. from binary trees to sets). Because such functions can be thought of as regaining the abstraction from among the implementation details, they are called *retrieve functions.*

from	$d \in Dict$	
1	$[\,] \in Dicta$	$Dicta$
2	$retr\text{-}Dict([\,]) = \{\,\}$	$retr\text{-}Dict$, **elems**
3	$\exists da \in Dicta \cdot retr\text{-}Dict(da) = \{\,\}$	$\exists\text{-}I$(1,2)
4	**from** $d \in Word\text{-}\mathbf{set},\ w \notin d,$ $\exists da \in Dicta \cdot retr\text{-}Dict(da) = d$	
4.1	**from** $da \in Dicta,\ retr\text{-}Dict(da) = d$	
4.1.1	$\mathbf{elems}\ da = d$	h4.1,$retr\text{-}Dict$
4.1.2	$w \notin \mathbf{elems}\ da$	h4,4.1.1
4.1.3	$da \curvearrowright [w] \in Dicta$	$Dicta$, 4.1.2, $is\text{-}uniques$
4.1.4	$\mathbf{elems}\ (da \curvearrowright [w]) = \mathbf{elems}\ da \cup \{w\}$	L7.6
4.1.5	$retr\text{-}Dict(da \curvearrowright [w]) = d \cup \{w\}$	4.1.1,4.1.4,retr-Dict
	infer $\exists e_1 \in Dicta \cdot retr\text{-}Dict(e_1) = d \cup \{w\}$	$\exists\text{-}I$(4.1.3, 4.1.5)
	infer $\exists e_1 \in Dicta \cdot retr\text{-}Dict(e_1) = d \cup \{w\}$	$\exists\text{-}E$(h4,4.1)
infer	$\exists da \in Dicta \cdot retr\text{-}Dict(da) = d$	$Set\text{-}ind$(h,3,4)

Theorem 8.2: adequacy of *Dicta*

The spell-checking specification of Section 4.1 is based on:

$Dict = Word\text{-}\mathbf{set}$

Assuming that the dictionary is large, the designer is faced with the problem of choosing a representation which makes efficient searching possible. Many facets of efficiency must be considered: the choice must reflect not only algorithms but also storage usage – wasted store could cause excessive paging and subvert the performance of a representation which was chosen to support some particular algorithm. Here, some alternative representations are considered. The first is chosen for pedagogic, rather than implementation, reasons. Suppose the dictionary is represented by a sequence without duplicates:

$Dicta = Word^*$

$\mathbf{inv}\ (ws) \triangleq is\text{-}uniques(ws)$

The one-to-many situation mentioned above is shown clearly here – to each abstract set with n words, there correspond $n!$ different possible sequence representations. The relationship between the representation and abstraction is easily expressed:

$$retr\text{-}Dict : Dicta \to Dict$$
$$retr\text{-}Dict(ws) \quad \triangle \quad \mathbf{elems}\ ws$$

Here, *retr-Dict* can be said to be retrieving the abstract set from among the irrelevant ordering information of the sequence values.

Proof obligation 8.1 One straightforward property which is required of retrieve functions is that they be *total*. In this case there is no doubt about *retr-Dict* since the **elems** operator is total on sequences. In some cases, however, it is necessary to tighten an invariant on the representation in order to ensure that the retrieve function is defined for all values which can arise.

Adequacy

It is intuitively clear that there should be at least one representation for any abstract value. This property is embodied in the *adequacy* proof obligation which, for the case of *Dicta* is shown in the following lemma.

Theorem 8.2 There must exist at least one sequence (without duplicates) which can be retrieved onto any possible set value:

$$d \in Dict \vdash \exists da \in Dicta \cdot retr\text{-}Dict(da) = d$$

The result here is obvious and the proof on page 187 is given only for illustration.

In the majority of cases, the adequacy proof obligation can be discharged by an informal, constructive argument. For example:

> Given any finite set, its elements can be arranged into a sequence by taking them in an arbitrary order – choosing each element once ensures that the representation invariant is not violated.

Proof obligation 8.3 Figure 8.1 illustrates the idea behind the adequacy proof obligation; the general form (for $retr: Rep \to Abs$) is:

$$\forall a \in Abs \cdot \exists r \in Rep \cdot retr(r) = a$$

Strictly, a representation is adequate – or not – with respect to a retrieve function. When the retrieve function in question is clear, the qualification is omitted.

Intuitively, a retrieve function[2] can be seen as providing an interpretation of the representation. In the initial example, two bytes are interpreted as a date.

[2]Technically, the retrieve function is a homomorphism between the carrier of the representation and that of the abstraction. The retrieve function can also be seen to induce an equivalence relation on the representation: two elements are considered to be equivalent if they are retrieved onto the same abstract value. This is a key concept for the proofs of the operations: the proof obligations in Section 8.2 require that the induced equivalence relation is respected.

In the case of the sequence of words, the retrieve function interprets it as the unordered set of *Dict* – such a sequence could just as well have represented the current book (where the order of the words is believed to be important).

Understanding a proof obligation is often made easier by considering cases where it fails. The attempt to represent dates in two-bytes discussed earlier in this section is *not* adequate because the limitation of the representation was not matched by the abstraction which put no limit (e.g. 1900-2027) on the possible years. Clearly, proof obligations are likely to uncover genuine errors only on larger examples – such failures are discussed below. With *Dict* and *Dicta*, however, a simple illustration can be given: suppose that the specification had been based on the sequences (*Dicta*) and the implementation on sets (*Dict*). Even with this reversal of roles, a retrieve function could be given:

$$retr\text{-}Dicta: Dict \to Dicta$$

For example, the function could deliver a sequence sorted in alphabetical order. But the representation would not be adequate because there would be elements of the specification state (e.g. unordered sequences) for which there was no corresponding set. Although the example is, in some sense, just a restatement of the need to avoid 'bias' in a specification, it should give some feel for why adequacy is a useful check. In more realistic examples, there are two likely causes of inadequacy. The obvious one is that some combination of values has been overlooked. This is clearly what the proof obligation is intended to uncover, and the situation must be remedied by redesigning the representation. The other way in which adequacy might fail is if the invariant on the abstraction is too loose: values might satisfy it which never arise as a result of a sequence of operations. If such values cannot be represented in the chosen design, the adequacy failure is only a technical issue. The invariant in the specification can be tightened (satisfiability must be rechecked) and the design can then be pursued.

More dictionary representations

The notions of retrieve functions and adequacy can now be applied to a more realistic design for the spell-checking specification. One way to provide for efficient searching is to split the dictionary into sections by word length; each such section is then stored in alphabetical order. As words are scanned from the text to be checked, their length is computed. The relevant dictionary section can then be located via a table and the word to be tested sought in the selected section. The search can use a technique known as 'binary search' (cf. Section 10.3), which is efficient because it relies on the order.

A series of distinct design decisions are embodied in this description. A record of the first design decision can be given in terms of the following objects:

$$Dictb = Section^*$$
$$\mathbf{inv}\ (sl) \triangleq \forall i \in \mathbf{inds}\ sl \cdot \forall w \in sl(i) \cdot \mathbf{len}\ w = i$$

$$Section = Word\text{-}\mathbf{set}$$

$$Word = Letter^+$$

Notice that, in order to describe the invariant, it has been necessary to say more about *Words* than in the specification. The retrieve function required here is:

$$retr\text{-}Dict : Dictb \to Dict$$
$$retr\text{-}Dict(sl) \quad \triangleq \quad \bigcup \mathbf{elems}\ sl$$

Here again there is no difficulty with totality, since both distributed union and **elems** are total; adequacy can be established by a simple constructive argument:

> the empty set can be represented by an empty sequence of sections; the way of representing a new *Word* depends on whether *Words* of the same length already occur in the *Dictb* value; if so, the new word is placed in the set; if not, the *Section* sequence is extended (if necessary) with empty *Sections* and the new *Word* is placed in a unit *Section* at the appropriate place in the *Section* sequence.

The next step of development might again be a reification of each *Section* onto a sequence. The final steps would concern the decomposition of operations specified by post-conditions onto the envisaged binary search algorithms.

The choice of representation is the crucial decision made by a designer to achieve efficiency: no amount of clever coding can restore performance squandered on ill-conceived data structures. Equally, correctness is vital. Representation decisions are normally made early in design. Errors made at this stage can be eradicated only by repeating the work based on the mistaken decision. It is, then, very important to make careful checks at this stage. The documentation of a retrieve function requires little effort and experience shows that this effort often uncovers errors. Similarly, outlining an adequacy argument for the representation of a state is not onerous and may uncover serious errors. Here the state alone is being considered; the proof obligations in Section 8.2 must be undertaken for each operation of the data type. It is therefore harder to justify the work of completely formal proofs for the operation proof obligations. It is, then, fortunate that experience has shown that these proof obligations are less likely (than adequacy) to uncover important errors.

The preceding representation required that a whole word be scanned before any searching could be done. A student project on this example proposed a way of using each letter as it is scanned. The initial proposal was to use Pascal arrays indexed by letters; the values stored in such arrays were to be pointers to other arrays; all of the arrays were allocated on the heap; **nil** pointers were to be used to mark where words ended. Using map notation, it is possible to represent this by nesting maps as follows:

$$Dicte = Letter \xrightarrow{m} Dicte$$

The word set:

$$\{[a, n, d], [a, n, t]\}$$

can then be represented by:

$$\{a \mapsto \{n \mapsto \{d \mapsto \{\,\}, t \mapsto \{\,\}\}\}\}$$

Notice how the lack, for example, of any word beginning with b is shown by the absence of this letter from the domain of the outer map.

But one must also notice that this representation is not adequate (with respect to any retrieve function)! There is, for example, no way of adding a word in *Dicte* which is a prefix of an existing word (consider $[a, n]$). On realizing this, the students had to add an indicator to each array (in Pascal, a record is used with a Boolean value and the array of pointers as its fields) – here:

$$\begin{aligned} Dictc :: \; & eow \; : \mathbb{B} \\ & map \; : Letter \xrightarrow{m} Dictc \end{aligned}$$

The retrieve function required is defined by recursion:

$$\begin{aligned} & retr\text{-}Dict : Dictc \to Dict \\ & retr\text{-}Dict(mk\text{-}Dictc(eow, m)) \quad \underline{\triangle} \\ & \qquad \bigcup\{\{[l] \curvearrowright w \mid w \in retr\text{-}Dict(m(l))\} \mid l \in \mathbf{dom}\, m\} \cup \\ & \qquad \mathbf{if}\ eow\ \mathbf{then}\ \{[\,]\}\ \mathbf{else}\ \{\,\} \end{aligned}$$

The reader should experiment with this retrieve function in order to understand the distinction in the second case of the set union. From this understanding it is possible to provide an invariant for *Dictc*.

Exercise 8.1.1 This exercise continues the spell-checking problem:

- In terms of some particular programming language, discuss the efficiency – especially storage requirements – of *Dictb* and *Dictc*.

- Define a representation in which all words with the same first letter are collected into a set, each such set is the range element of a map from the first letter. Write a retrieve function and argue the adequacy of the representation.

Exercise 8.1.2 Document the relationship between the state given in Exercise 4.4.4 on page 108 and that given in the text of Section 4.4 by writing retrieve functions in *both* directions.

Exercise 8.1.3 Consider the set of objects *Llist* described in Section 5.1 and the sequences of Chapter 7. In which directions can retrieve functions be written?

Exercise 8.1.4 Explain the binary trees (*Setrep*) of Section 5.2 as representations of sets by using retrieve functions, and present an adequacy proof.

Exercise 8.1.5 Many encodings are used for integers. A binary numeral can be thought of as a sequence of symbols – show how a (natural number) value can be associated with such a symbol sequence by providing a retrieve function. The sign-and-magnitude representation of integers used in some computers reserves one bit for the sign and the remaining bits in a word are used as above – again, explain this relation with a retrieve function. The ones-complement representation essentially stores, for negative numbers, the bit-wise complement of the positive number – here again, explain the relation by a retrieve function (remember that all zeros or all ones represent the number zero).

Exercise 8.1.6 Consider the abstract state:

$$\begin{array}{lll} State :: & as & : X\textbf{-set} \\ & bs & : X\textbf{-set} \end{array}$$

$$X = \{1, ..., n\}$$

(for some n) and a representation:

$$Arep = \mathbb{B}^*$$

in which it is intended to use one 'bit' for each number. Write a retrieve function and either prove *Arep* to be adequate or show how it fails to be and suggest an invariant on *State* which ensures that the representation is adequate.

8.2 Operation modelling proofs

Section 8.1 gives some examples of the way in which design steps of data reification give rise to complex data objects. This complexity reflects the move from data objects which are chosen to match the task being specified to representations which can be efficiently implemented. Efficiency might require redundancy (e.g. doubly-linked lists or extra copies) and this results in lengthier invariants. Turning now to the operations: in general, representation detail forces operation specifications to be more algorithmic; for example, neat post-conditions on the abstraction might give way to recursive functions on the representation. As the examples below illustrate, post-conditions are more concise than code – but the closer the representation is to the data types of programming languages, the more complex will be the specifications. This is, of course, precisely the reason that overall functional specifications should be written in terms of abstract data types. But the time has come to look at the proof obligations associated with the modelling of operations.

Modelling in the function case

An abstract specification consists of a set of states, the initial states, and operations. The preceding section has shown how the states themselves are reified. The next design task is to respecify the operations on the chosen state representation. The format of such operation specifications is standard. Thus the *CHECKWORD* operation of Section 4.1 would be respecified on *Dicta* of the preceding section by:

$$
\begin{array}{l}
\mathit{CHECKWORDa}\ (w\colon \mathit{Word})\ b\colon \mathbb{B} \\
\textbf{ext rd}\ \mathit{dict} : \mathit{Dicta} \\
\textbf{post}\ b \ \Leftrightarrow\ \exists i \in \textbf{inds}\, \mathit{dict} \cdot \mathit{dict}(i) = w
\end{array}
$$

This operation on *Dicta* has to be shown to exhibit the same behaviour as *CHECKWORD* on *Dict*. It is easier, however, to picture the proof obligations which arise in the case of functions than operations. Figure 8.2 shows an (abstract) function f_a over elements of A; an alternative way of performing such a mapping is to find a corresponding element of R (assume, for now, that *retr* is one-to-one so that its inverse *rep* is a function), apply a function f_r on R, and then map this back to A by applying *retr*. The function f_r models f_a if the alternative mapping is the same as f_a for all values in the domain of f_a. This could be written:

$$\forall a \in A \cdot \mathit{retr}(f_r(\mathit{rep}(a))) = f_a(a)$$

The essence of this is to require that f_r's behaviour is the same as that of f_a.

A neater way of presenting this (which avoids the need for the inverse: *rep*) is to require:

$$\forall r \in R \cdot \mathit{retr}((f_r(r)) = f_a(\mathit{retr}(r))$$

$$\begin{tikzcd} a_i \arrow[r, "f_a"] \arrow[d, "rep"'] & a_j \\ r_i \arrow[r, "f_r"'] & r_j \arrow[u, "retr"'] \end{tikzcd}$$

Figure 8.2 Function modelling

But this only works in the case that the representation is adequate: the universally quantified statement would be true for an empty R. This transition then shows why adequacy is important and prepares the way for the rules required for operations.

Proof obligations for operation modelling

The proof obligations needed for operations have the same motivation as those for the functional case but have to cope with two complications. Firstly, operations themselves are partial (cf. pre-condition) and non-deterministic (cf. post-condition); secondly, retrieve functions are normally many-to-one and thus their inverses are not functions. The basic proof obligations for *operation modelling* follow.[3]

Proof obligation 8.4 The domain rule is:

$$\forall r \in R \cdot \textit{pre-A}(\textit{retr}(r)) \;\Rightarrow\; \textit{pre-R}(r)$$

Proof obligation 8.5 The result rule is:

[3] The validity of the proof rules given here relies on the adequacy of the representation. The concept of viewing under the retrieve function can be formalized by requiring that representation operations respect the equivalence relation induced on the representation states by the retrieve function.

$$\forall \overleftarrow{r}, r \in R \cdot$$
$$\quad pre\text{-}A(retr(\overleftarrow{r})) \wedge post\text{-}R(\overleftarrow{r}, r) \;\Rightarrow\; post\text{-}A(retr(\overleftarrow{r}), retr(r))$$

These rules can be extended in an obvious way to cope with inputs and results of operations since these do not get changed by reification: it is the behaviour – as seen via the inputs/outputs – which is preserved by reification. One way of comprehending the resulting proof rules is to think of *viewing the behaviour* of the operations on the representation via the retrieve function. The second of these proof obligations is known as the *result rule*. This can be seen as requiring that any pair of states in the *post-R* relation must – when viewed under the retrieve function – satisfy the *post-A* relation. An implementation should not be rejected for an unnecessarily wide pre-condition, nor should it be forced to perform any particular (e.g. *post-R*) computation outside the required domain. Thus the first conjunct of the antecedent of the implication limits the proof obligation to those states which – when viewed under the retrieve function – satisfy the abstract pre-condition. The result rule requires that, although defined on different states, the operations R and A model the same behaviour.

The explanation of the result rule argues against requiring too much of the operations on the representation. It must, however, be remembered that the specification of the operations on the representation consist of two parts. The result rule ensures that the post-condition is not too wide; the *domain rule* (first above) requires that the pre-condition of the operation on the representation is not too narrow. So, if the pre-condition of the abstract operation is true of a retrieved state, the representation state must satisfy the pre-condition of the representation operation.

Theorem 8.6 For the first example in the preceding section, the sequent form of the domain obligation 8.4:

$$ws \in Dicta,\; w \in Word \;\vdash$$
$$\quad pre\text{-}CHECKWORD(w, retr\text{-}Dict(ws))$$
$$\quad \Rightarrow\; pre\text{-}CHECKWORDa(w, ws)$$

is vacuously true because the operation on the representation is total.

Theorem 8.7 Noting that the pre-condition of the abstract operation is also **true**, the sequent form of the result obligation (8.5) becomes:

$$ws \in Dicta,\; w \in Word,\; b \in \mathbb{B} \;\vdash$$
$$\quad post\text{-}CHECKWORDa(w, ws, b) \;\Rightarrow$$
$$\quad\quad post\text{-}CHECKWORD(w, retr\text{-}Dict(ws), b)$$

which follows from:

$$ws \in Word^*,\ w \in Word,\ b \in \mathbb{B} \vdash$$
$$(b \Leftrightarrow \exists i \in \mathbf{inds}\, ws \cdot ws(i) = w) \Rightarrow (b \Leftrightarrow w \in \mathbf{elems}\, ws)$$

Thus, *CHECKWORDa* can be said to model *CHECKWORD*. Strictly, this statement is with respect to *retr-Dict* but, here again, the qualification can normally be omitted without confusion.

The *ADDWORD* operation changes the state and can be modelled by:

$ADDWORDa\ (w{:}\ Word)$
ext wr $dict\ :\ Dicta$
pre $\neg\, \exists i \in \mathbf{inds}\, dict \cdot dict(i) = w$
post $dict = \overleftarrow{dict} \curvearrowright [w]$

Theorem 8.8 Its domain rule becomes:

$$ws \in Dicta,\ w \in Word \vdash$$
$$pre\text{-}ADDWORD(w, retr\text{-}Dict(ws)) \Rightarrow pre\text{-}ADDWORDa(w, ws)$$

This is proved on page 197.

Theorem 8.9 It is often convenient to expand out the definitions. The result rule becomes:

$$\overleftarrow{ws}, ws \in Dicta,\ w \in Word \vdash$$
$$w \notin \mathbf{elems}\, \overleftarrow{ws} \wedge ws = \overleftarrow{ws} \curvearrowright [w] \Rightarrow \mathbf{elems}\, ws = \mathbf{elems}\, \overleftarrow{ws} \cup \{w\}$$

Which is, again, straightforward (cf. page 197).

Thus *ADDWORDa* models *ADDWORD*. If these are the only operations, the reification has been justified and attention can be turned to the next step of development.

If defined, it is also necessary to show that the initial states correspond – with respect to the retrieve function. The proof is straightforward in this case and is shown explicitly only on examples where the initial states are less obvious.

In large applications of the rigorous approach, there are likely to be several stages of data reification: when the data objects have been refined to the level of the machine or language constructs, operation decomposition is carried out. In either case, the compositionality property of the development method requires that the next step of development relies only on the result (e.g. *Dicta*, etc.) of this stage of development and not on the original specification.

$$
\begin{array}{lll}
\textbf{from } ws \in Word^*,\ w \in Word & & \\
1 \quad \textbf{from } w \notin \textbf{elems } ws & & \\
\quad\quad \textbf{infer } \neg\, \exists i \in \textbf{inds } ws \cdot w = ws(i) & & \textbf{elems} \\
2 \quad \delta(w \notin \textbf{elems } ws) & & \in\text{,h} \\
3 \quad w \notin \textbf{elems } ws \ \Rightarrow\ \neg\, \exists i \in \textbf{inds } ws \cdot w = ws(i) & & \Rightarrow\text{-}I(2,1) \\
\textbf{infer } \textit{pre-ADDWORD}(w, \textit{retr-Dict}(ws)) \ \Rightarrow\ \textit{pre-ADDWORDa}(w, ws) & &
\end{array}
$$

Theorem 8.8: domain rule for $ADDWORDa$

$$
\begin{array}{lll}
\textbf{from } \overleftarrow{ws}, ws \in Word^*,\ w \in Word & & \\
1 \quad \textbf{from } ws = \overleftarrow{ws} \curvearrowright [w] & & \\
1.1 \quad\quad \textbf{elems } ws = \textbf{elems } \overleftarrow{ws} \cup \textbf{elems } [w] & & \text{L7.6(h1)} \\
\quad\quad \textbf{infer } \quad = \textbf{elems } \overleftarrow{ws} \cup \{w\} & & \textbf{elems} \\
2 \quad \delta(ws = \overleftarrow{ws} \curvearrowright [w]) & & \curvearrowright\text{, h} \\
\textbf{infer } ws = \overleftarrow{ws} \curvearrowright [w] \ \Rightarrow\ \textbf{elems } ws = \textbf{elems } \overleftarrow{ws} \cup \{w\} & & \Rightarrow\text{-}I(2,1)
\end{array}
$$

Theorem 8.9: result rule for $ADDWORDa$

Modelling proofs for the other dictionary representation

The operations on the second dictionary representation are addressed in Exercise 8.2.1 below. The third dictionary representation given above is more interesting. In this case, the initial state is worth special consideration.

Theorem 8.10 The proof obligation for initial states is (with $\textit{retr-Dict}: \textit{Dictc} \to \textit{Dict}$):

$$\textit{dictc}_0 \in \textit{Dictc} \ \vdash\ \textit{retr-Dict}(\textit{dictc}_0) = \textit{dict}_0$$

This can be satisfied with:

$$\textit{dictc}_0 = \textit{mk-Dictc}(\textbf{false}, \{\,\})$$

The specification of *CHECKWORDc* must be written in terms of *Dictc*. A specification which used the retrieve function would make little real progress in design. To avoid such insipid steps of development, one could use a function:

$$\begin{array}{l}
\mathit{is\text{-}inc} : \mathit{Word} \times \mathit{Dictc} \to \mathbb{B} \\
\mathit{is\text{-}inc}(w, \mathit{mk\text{-}Dictc}(\mathit{eow}, m)) \quad \triangleq \\
\qquad w = [\,] \wedge \mathit{eow} \vee \\
\qquad w \neq [\,] \wedge \mathbf{hd}\, w \in \mathbf{dom}\, m \wedge \mathit{is\text{-}inc}(\mathbf{tl}\, w, m(\mathbf{hd}\, w))
\end{array}$$

Theorem 8.11 The modelling proof relies on the lemma:

$$w \in \mathit{Word},\ d \in \mathit{Dictc} \vdash \mathit{is\text{-}inc}(w, d) \ \Leftrightarrow\ w \in \mathit{retr\text{-}Dict}(d)$$

This can be proved by structural induction.

In fact, a theory of *Dictc* can be developed. A function which inserts words is:

$$\begin{array}{l}
\mathit{insc} : \mathit{Word} \times \mathit{Dictc} \to \mathit{Dictc} \\
\mathit{insc}(w, \mathit{mk\text{-}Dictc}(e, m)) \quad \triangleq \\
\qquad \mathbf{if}\ w = [\,] \\
\qquad \mathbf{then}\ \mathit{mk\text{-}Dictc}(\mathbf{true}, m) \\
\qquad \mathbf{else\ if}\ \mathbf{hd}\, w \in \mathbf{dom}\, m \\
\qquad\qquad \mathbf{then}\ \mathit{mk\text{-}Dictc}(e, m \dagger [\mathbf{hd}\, w \mapsto \mathit{insc}(\mathbf{tl}\, w, m(\mathbf{hd}\, w))]) \\
\qquad\qquad \mathbf{else}\ \mathit{mk\text{-}Dictc}(e, m \cup [\mathbf{hd}\, w \mapsto \mathit{insc}(\mathbf{tl}\, w, \mathit{mk\text{-}Dictc}(\mathbf{false}, \{\,\}))])
\end{array}$$

Lemma 8.12 The relevant lemma here is:

$$\boxed{\text{L8.12}}\ \frac{w \in \mathit{Word};\ d \in \mathit{Dictc}}{\mathit{retr\text{-}Dict}(\mathit{insc}(w, d)) = \mathit{retr\text{-}Dict}(d) \cup \{w\}}$$

Buffer pools and non-determinism

In the spell-checking example, all of the operations are deterministic. The buffer pool example of Section 4.4 exhibits non-determinism. The abstract buffer pool is shown as:

$$\mathit{Bid}\text{-}\mathbf{set}$$

Suppose this is modelled by:

$$\begin{array}{l}
\mathit{Bufl} = \mathit{Bid}^* \\
\mathbf{inv}\ (l) \triangleq \mathit{is\text{-}uniques}(l)
\end{array}$$

Clearly this is an adequate representation with respect to the retrieve function:

$$retr\text{-}BUF : Bufl \to Bid\textbf{-set}$$
$$retr\text{-}BUF(bidl) \quad \triangleq \quad \textbf{elems}\ bidl$$

The model of $OBTAIN$ can be specified:

$OBTAIN1$ () res: Bid
ext wr us : $Bufl$
pre $us \neq [\,]$
post $\overleftarrow{us} = [res] \frown us$

The domain proof obligation is straightforward.

Theorem 8.13 That for the result becomes:

$$\overleftarrow{us}, us \in Bufl;\ res \in \mathbb{B} \vdash$$
$$retr\text{-}Buf(\overleftarrow{us}) \neq \{\,\} \wedge \overleftarrow{us} = [res] \frown us \;\Rightarrow$$
$$res \in retr\text{-}Buf(\overleftarrow{us}) \wedge retr\text{-}Buf(us) = retr\text{-}Buf(\overleftarrow{us}) - \{res\}$$

Notice that a proof of this result relies on the invariant of *Bufl*.
Thus $OBTAIN1$ resolves the non-determinism in $OBTAIN$ and exhibits an acceptable behaviour.

Exercise 8.2.1 The spell-checker application can be used to show that the proof obligation given in this section caters for non-determinism in representation operations. Respecify $ADDWORDa$ to insert the new word anywhere in the sequence and show that the revised operation specification models $ADDWORD$. Specify operations to work on *Dictb* of the preceding section and show that they model those of the specification in Section 4.1.

Other applications

Two models are in some sense equivalent if retrieve functions can be written in both directions. There is, in fact, a one-to-one correspondence[4] between elements of both models. It can be useful to build a specification around two or more equivalent models. For example, one model may require a minimal invariant, while another may offer a state with many sub-components, thus shortening the specifications of operations which affect only some of the sub-components. In such a case, two models should be used and the appropriate retrieve functions given. This is an alternative to the creation of extra functions which define alternative views of one basic model.

[4] They are isomorphic.

Exercise 8.2.2 Exercise 4.4.2 on page 108 introduces a security tracking application. Define a representation in terms of sequences; provide retrieve functions and adequacy proofs; specify operations on the sequences; and prove that they model those on the abstract state.

Exercise 8.2.3 Exercise 4.4.3 on page 108 and Exercise 6.3.4 on page 161 use two different states for the same family of operations. Show that the specification based on a pair of sets can be thought of as a reification of that based on a map (expand and check the proof obligations for the state and the operations).

Exercise 8.2.4 It is easy to specify operations which allocate elements onto two distinct sequences. If there is not a reasonable upper size bound for at least one of the sequences, the representation in a normal linearly addressed store presents problems. Such a situation arises with the stack and heap in some programming languages. One well-known technique is to reserve a large contiguous area for both sequences and to allocate their space from opposite ends of the space. Describe the abstract problem and its solution using two models and show that one is a reification of the other (consider the initial state).

Exercise 8.2.5 Section 6.3 includes a discussion of virtual store showing abstract and implemented models. Justify the correctness of the development in terms of the proof obligations of this chapter.

Exercise 8.2.6 Complete the development begun in Exercise 8.1.4 on page 192 by specifying and justifying the operations on *Setrep* (consider the initial state).

8.3 Exercises in reification

This section presents a larger exercise in data reification and, as well as this development from abstraction to representation of the sort discussed above, it indicates the way in which the same techniques can be used to handle interfaces which are predefined in a project. Although many people argue that systems should be developed top-down – thus developing interfaces as part of the design process – many large systems are in fact split by setting some concrete interface decisions very early. The problem of working to a predefined interface is also often faced by developers who design an addition to an existing system.

Binary tree representations

Section 5.2 shows that a form of binary tree (*Setrep*) can be used to store representations of sets. The advantage of the binary tree representation is that it facilitates efficient search and update operations: the number of search steps is proportional to the logarithm – base 2 – of the number of elements, provided the tree is balanced. A great many computer applications rely in some way on large associations between keys and data. An extended form of binary tree can be used to provide similar performance advantages for representations of such maps. In contrast to those used for set representations (cf. *Setrep*), these trees have nodes which contain a *Key*/*Data* pair.

The top-level specification of a map from *Keys* to *Data* is made trivial by the availability of suitable base objects. Thus:

$$Kdm = Key \xrightarrow{m} Data$$

The initial object in *Kdm* is: $m_0 = \{\,\}$. Operations can be defined:

$FIND\ (k{:}\,Key)\ d{:}\,Data$
$\textbf{ext rd}\ m : Kdm$
$\textbf{pre}\ k \in \textbf{dom}\, m$
$\textbf{post}\ d = m(k)$

$INSERT\ (k{:}\,Key, d{:}\,Data)$
$\textbf{ext wr}\ m : Kdm$
$\textbf{pre}\ k \notin \textbf{dom}\, m$
$\textbf{post}\ m = \overleftarrow{m} \cup \{k \mapsto d\}$

$DELETE\ (k{:}\,Key)$
$\textbf{ext wr}\ m : Kdm$
$\textbf{pre}\ k \in \textbf{dom}\, m$
$\textbf{post}\ m = \{k\} \mathbin{\lhd\!\!\!-} \overleftarrow{m}$

The maps (*Kdm*) can be represented by:

$$Mrep = [Mnode]$$

$$\begin{array}{lll} Mnode :: & lt & : Mrep \\ & mk & : Key \\ & md & : Data \\ & rt & : Mrep \end{array}$$

$\textbf{inv}\ (mk\text{-}Mnode(lt, mk, md, rt)) \triangleq$
$(\forall lk \in collkeys(lt) \cdot lk < mk) \wedge (\forall rk \in collkeys(rt) \cdot mk < rk)$

Where:

$$
\begin{array}{l}
\mathit{collkeys} : \mathit{Mrep} \to \mathit{Key}\text{-}\mathbf{set}\\
\mathit{collkeys}(t) \;\;\triangleq\\
\quad \mathbf{cases}\ t\ \mathbf{of}\\
\quad \mathbf{nil} \qquad\qquad\qquad\qquad\quad \to \{\,\},\\
\quad \mathit{mk\text{-}Mnode}(lt, mk, md, rt) \to \mathit{collkeys}(lt) \cup \{mk\} \cup \mathit{collkeys}(rt)\\
\quad \mathbf{end}
\end{array}
$$

A small theory of the *Mrep* type can be developed. Some lemmas which are stated here without proof are:

Lemma 8.14 The function *collkeys* is total:

$$
\boxed{\text{L8.14}}\ \frac{t \in \mathit{Mrep}}{\mathit{collkeys}(t) \in \mathit{Key}\text{-}\mathbf{set}}
$$

Lemma 8.15 Left and right sub-trees contain, because of the invariant, disjoint sets of keys:

$$
\boxed{\text{L8.15}}\ \frac{\mathit{mk\text{-}Mnode}(lt, mk, md, rt) \in \mathit{Mnode}}{\mathit{is\text{-}prdisj}(\mathit{collkeys}(lt), \{mk\}, \mathit{collkeys}(rt))}
$$

where:

$$
\mathit{is\text{-}prdisj} : X\text{-}\mathbf{set} \times X\text{-}\mathbf{set} \times X\text{-}\mathbf{set} \to \mathbb{B}
$$

Lemma 8.16 The value *mk* shows which sub-tree to search:

$$
\boxed{\text{L8.16}}\ \frac{\mathit{mk\text{-}Mnode}(lt, mk, md, rt) \in \mathit{Mnode};\quad k \in \mathit{collkeys}(\mathit{mk\text{-}Mnode}(lt, mk, md, rt))}{(k < mk \;\Rightarrow\; k \in \mathit{collkeys}(lt)) \wedge (mk < k \;\Rightarrow\; k \in \mathit{collkeys}(rt))}
$$

The retrieve function is:

$$
\begin{array}{ll}
\mathit{retr\text{-}Kdm} : \mathit{Mrep} \to \mathit{Kdm} & \\
\mathit{retr\text{-}Kdm}(t) \;\triangleq & \mathbf{cases}\ t\ \mathbf{of}\\
& \mathbf{nil} \qquad\qquad\qquad\quad \to \{\,\},\\
& \mathit{mk\text{-}Mnode}(l, k, d, r) \to \mathit{retr\text{-}Kdm}(l) \cup \{k \mapsto d\} \cup \mathit{retr\text{-}Kdm}(r)\\
& \mathbf{end}
\end{array}
$$

The totality of this retrieve function relies on Lemma 8.15. The adequacy of *Mrep* can be argued in a way similar to the proof for *Setrep* in Section 5.2.

Lemma 8.17 Another useful lemma is:

$$
\boxed{\text{L8.17}}\ \frac{t \in \mathit{Mrep}}{\mathbf{dom}\ \mathit{retr\text{-}Kdm}(t) = \mathit{collkeys}(t)}
$$

This example is developed in Section 10.4 where the induction rule (A10.1) is given.

Exercise 8.3.1 (*) B-trees are generalizations of binary trees. The order (say N) of a B-tree limits the branching at each node and can be chosen to achieve efficient transfers from backing store. Any (non-root) node of a B-tree has between N and $2N$ elements; in leaf nodes, these elements are pairs of key and data; for intermediate nodes, the elements are pointers to other nodes – as with binary trees, keys which guide the search are also stored at intermediate nodes. (A full description can be found in *The Ubiquitous B-tree* by D. Comer in ACM Computing Surveys, Vol.11, No.2, pp121-137.) Write descriptions of B-trees on several levels of abstraction.

Exercise 8.3.2 (*) Hashing provides an alternative way of storing information for rapid searching. A hash function maps $Keys$ to a subset of natural numbers. If the hash function were one-to-one, this would simply be a computed address where the information ($Key/Data$) is stored. Hash functions are, in fact, many-to-one and the interesting problems concern the handling of collisions where two or more $Keys$ are mapped to the same hash address. (Much of the subtlety in developing hashing techniques for particular applications concerns the minimization of collision – these aspects are not of concern here.) Describe on two levels of abstraction the general idea of hashing where records with colliding keys are placed in the 'next gap'.

Exercise 8.3.3 (*) A graph consists of a set of nodes and arcs. An abstract representation considers an arc as an ordered pair (of node identifiers) and the whole graph as a set of arcs. Document this abstract description and define simple operations to test if an arc is present and to add an arc.

Two possible representations are:

- a two-dimensional array (where each dimension is indexed by the node identifiers) which records whether or not the relevant pairs are linked by an arc;
- a one-dimensional array of pointers (indexed by the node identifiers) to linked lists of records; the non-link information in each record is the identifier of nodes to which arcs are present.

Document and justify these two representations at sensible levels of abstraction.

Handling fixed interfaces

The method of developing from an abstract type to a more concrete representation should be clear. There are, however, situations in software development where a concrete interface definition is one of the reference points in a development. There is nothing essentially wrong with this situation, and the remainder of this section shows how the data reification ideas can still be applied. The problems which *can* occur with interface descriptions are legion. Firstly, interfaces are often recorded with far too much syntactic detail. Information on physical control blocks is sometimes described at the bit and byte level. This militates against the modern programming ideas of abstract data types. Use of such detail can lead to efficiency but almost certainly results in systems which are not maintainable. Many very large systems have made the mistake of fixing bit/byte details and an enormous penalty has resulted. In spite of the fact that this mistake is so serious, it is not the purpose of the current book to preach ideas which have long been standard practice in better organized development groups. Here, it is necessary to show only how the data reification ideas can help avoid the problem. An even more common failing is the lack of semantics in interface descriptions. In contrast to the excessive syntactic detail, the actual meaning or effect of fields in an interface is often suggested by no more than the field names. The programming language Ada is in danger of perpetuating this problem by using the term 'interface' to describe something which only has the power to define syntactic (procedure) interface questions.

Faced with a fixed concrete interface in a development, there is a series of steps which can be used to clarify an interface and to record the understanding. These steps are:

1. write an (abstract) data type with only the essential information content;

2. record the semantics with respect to this abstract interface;

3. relate the (given) concrete details to the abstraction with a retrieve function.

These steps cannot, in large applications, be performed strictly sequentially: there is a constant interplay between them.

A major application in which the author was involved concerned the development of a compiler for the PL/I language. The interest was in the back-end (object time organization and object code) issues and it was decided to take over the front-end (parser and dictionary building) of an existing compiler. The text interface had a fairly obvious linearized version of the parse tree (see Exercise 8.3.5 on page 207). Variable references in the text were represented (among other things) by pointers into the dictionary. The dictionary had been designed for compactness and was a mass of special cases. The documentation was quite good on byte values but the main part of the semantics had to be deduced from examples. The proposal to follow the plan set out above was met with some scepticism as to whether the time was available. Only the impossibility of getting the interface under intellectual control in any other way convinced the group. (Some of the material is available as a technical report – see [Wei75].) The effect was certainly worthwhile from this point of view alone.

Here, a simpler – but equally representative problem – is considered. A paper by Henderson and Snowdon – see [HS72] – includes the following introduction of a problem:

> A program is required to process a stream of telegrams. This stream is available as a sequence of letters, digits and blanks on some device and can be transferred in sections of predetermined size into a buffer area where it is to be processed. The words in the telegrams are separated by sequences of blanks and each telegram is delimited by the word 'ZZZZ'. The stream is terminated by the occurrence of the empty telegram, that is a telegram with no words. Each telegram is to be processed to determine the number of chargeable words and to check for occurrences of overlength words. The words 'ZZZZ' and 'STOP' are not chargeable and words of more than twelve letters are considered overlength. The result of the processing is to be a neat listing of the telegrams, each accompanied by the word count and a message indicating the occurrence of an overlength word.

A number of questions are unresolved by this text and some computing scientists have used this as a criticism of the text as a specification. Although this is not an excuse, far worse documents are used as the basis of far larger systems. The debate is, however, sterile, and here the text is treated as an indication of requirements. One important by-product of the proposed method for addressing interfaces is that it is likely to expose many of the lacunae. Questions on the text fall into two broad areas. Questions about the effect (semantics) of the operations include:

- Are overlength words to be truncated in the output?
- How are overlength words to be charged?
- What output is to be printed for overlength words?
- Does the count for overlength really not count digits?
- Is a report required for the empty telegram?
- What error handling is to be provided?

Some of the questions about how the information is represented are:

- What is the meaning of 'delimit'?
- What is the meaning of 'sequence' (e.g. zero occurrences)?
- What determines the buffer size?
- Can words span blocks?
- What is an 'empty' telegram?
- What is a 'neat listing'?
- Are leading spaces allowed in records?

It is not difficult – in this case – to find a suitable abstract description of both the input and output:

$$Input = Telegram^*$$

$$Telegram = Word^*$$

$$Word = Character^*$$

$$Character = Letter \cup Digit$$

$$Output = Report^*$$

Where *Telegram* and *Word* are non-empty sequences; but $[Z, Z, Z, Z] \notin Word$; and:

$$\begin{array}{lll} Report :: & tgm & : Telegram \\ & count & : \mathbb{N} \\ & ovlen & : \mathbb{B} \end{array}$$

This abstraction ignores the details of the blanks which delimit words or the special words used to terminate telegrams. The required meaning is given by:

$$\begin{array}{l} ANALYZE\ (in{:}\ Telegram^*)\ out{:}\ Report^* \\ \textbf{post len } out = \textbf{len } in\ \wedge \\ \qquad \forall i \in \textbf{inds } in \cdot out(i) = analyze\text{-}telegram(in(i)) \end{array}$$

$$\begin{array}{l} analyze\text{-}telegram(wordl) \quad \triangleq \\ \qquad mk\text{-}Report(wordl, charge\text{-}words(wordl), check\text{-}words(wordl)) \end{array}$$

$$charge\text{-}words(wordl) \quad \triangleq \quad \mathbf{card}\ \{j \in \mathbf{inds}\ wordl \mid wordl(j) \neq [\mathbf{S,T,O,P}]\}$$

$$check\text{-}words(wordl) \quad \triangleq \quad \exists w \in \mathbf{elems}\ wordl \cdot \mathbf{len}\ w > 12$$

This has shown how the process of recording such a description can be used to document the interpretation of open semantic questions. For this author, it was also the way of generating the list of questions about the requirements. The next step is obviously to face the other part of the problem, which is the representation details. The representation can be viewed as:

$$Inputr = Block^*$$

$$Block = Symbol^*$$

$$Symbol = Character \cup \{\textsc{Blank}\}$$

The specification is completed by documenting the relationship of *Inputr* to *Input* via a retrieve function. Here, this is left as an exercise. The important message of this approach to interfaces is both the value for uncovering imprecision and the ability to record precisely the chosen understanding. The documentation can also be an aid in implementation: separate data types can be readily identified.

One of the reasons that the Henderson and Snowdon paper has evoked so much interest is their description of how one error got into their design. Not only was this error avoided by the development based on the abstract specification given here, but also other errors were uncovered in the program given in the original paper.

Exercise 8.3.4 Write a retrieve function for the input to the telegram analysis problem. Fix a representation for output and document its relationship to the abstraction.

Exercise 8.3.5 (*) Choose a simple linear form (e.g. reverse Polish) for an expression language and document the relationship to a tree form for expressions.

Exercise 8.3.6 (*) A syntax-directed editor permits a user to enter, at a terminal, a program by placing information into an abstract syntax tree. The current content of a program (and the identification of holes) is displayed by an 'unparsing scheme' which relates concrete to abstract syntax. Such syntax-directed editors are table driven in that the abstract syntax and projection schemes are stored as data. Describe the general idea of syntax-directed editors.

9

More on Data Types

> Everything should be made as simple as possible,
> but not more so.
> A. Einstein

The material in this book is, in spite of using VDM, not specific to that notation. A reader interested in the ideas – but working with another notation – should have had no difficulty in following the concepts presented. When one wishes to employ tools to handle specifications, it is necessary to become pedantic about syntax. Whereas formulae can be linked by text in a textbook, a framework of keywords is needed by a mechanical parser. The first two sections of this chapter discuss further aspects of VDM notation as defined by the British Standards Institution (BSI) committee (BSI IST 5/50). Section 9.1 develops the material on abstract data types by showing how modules can be used as a unit of encapsulation. Section 9.2 describes notation for specifying exceptional conditions. Since the module notation is one of the parts of the BSI work which is marked as tentative (see [BSI89] – the essential parts of which are reproduced in Appendix F), only an outline is provided. There should be no difficulty in filling in the details when the standard is finally frozen in this area.

A number of more subtle points about data types are also considered in this chapter. The approach described above is to define the data types in terms of models. This model-oriented approach presents some danger of overspecification. In particular, models can be biased towards certain implementations. A test for bias is given in Section 9.3 together with some general proof rules for data reification. Section 9.4 presents an alternative way of specifying data types: the property-oriented approach is shown to be well-suited to basic types; the applicability of this approach to the sort of data types required in larger applications is also reviewed; a comparison with the model-oriented approach is included along with an attempt to define appropriate roles for the two approaches.

9.1 Modules as data types

This section outlines notation which binds a model and a collection of operations together into a *module*. Such modules (with import/export lists governing what can be used across the interfaces) make it possible for one module to rely on another in a controlled way. It is important to understand that the concept of modules in a specification language is intended to help form a specification; it is not intended to provide a guide to the implementations. There are several reasons for this caveat. The most obvious one is that the implementation might need to adopt a different structuring in order to achieve acceptable performance.

Module notation

The basic module notation is very simple. Keywords are used around a module as follows (the example of bags from Section 6.3 is used to introduce the notation):

module BAG

$\vdots$

end BAG

Thought of as a collection of a state description – possibly some auxiliary functions – and a collection of operations, one can see the use of the following keywords:

```
module BAG
   ⋮
definitions
types
   ⋮
state
   ⋮
end ;
functions
   ⋮
operations
   ⋮
end BAG
```

The need to define the interface of a module leads to keywords for (in this case) export lists:

```
module BAG
exports
types
  Bag
operations
   ⋮
definitions
   ⋮
end BAG
```

Putting this structure around the material in Section 6.3 yields:

module BAG
parameters types X
exports
types
Bag
operations
$COUNT\colon X \xrightarrow{o} \mathbb{N}$,
$ADD\colon X \xrightarrow{o}$
definitions
types
$Bag = X \xrightarrow{m} \mathbb{N}_1$;
state
$State$ **of** $b\colon Bag$
init $(mk\text{-}State(b_0)) \triangleq b_0 = \{\,\}$
end ;

functions

$mpc : X \times Bag \to \mathbb{N}$
$mpc(e, m) \quad \triangleq \quad \textbf{if } e \in \textbf{dom}\, m \textbf{ then } m(e) \textbf{ else } 0$

operations

$COUNT\ (e\colon X)\ c\colon \mathbb{N}$
ext rd $b : Bag$
post $c = mpc(e, b)$

$ADD\ (e\colon X)$
ext wr $b : Bag$
post $b = \overleftarrow{b} \dagger \{e \mapsto mpc(e, \overleftarrow{b}) + 1\}$

end BAG

One important property of data types is the possibility which they offer to 'close off' one piece of work so that it can be used in another. One manifestation of the separation property – in the case of state-based data types – is the ability to change the internal details of one data type without needing to change any data type which uses it. This can be achieved only if the behaviour of the operations within the used type remains the same: insulation is given only against changes to internal details. But, providing the change is to an equivalent specification in the sense that sequences of its operations yield the same results in the externally visible types, it is true that an operation using it will itself preserve its original behaviour. Thus, given the definition of the module BAG, objects of type Bag can now be used in other data types. Such objects can, however, only be manipulated by the exported operations and the internal representation of Bag could be changed without affecting the data types which use BAG. The next subsection explains how the exported operations can be used.

The insulation provided by data types is a valuable property, but it certainly does not justify making every component of every state into a separate data type. Taste in the selection of data types comes both from consideration of their likely reuse and of their representing coherent concepts. This latter consideration has to be judged on whether the operators present a clear theory.

Operation quotation

In a programming language, the exported operations of one data type would be invoked in another data type as procedures. VDM's operations are like procedures in that they change a state. Although this fact has been seen to be very useful in constructing specifications of systems, it does present a difficulty when operations in a new data type are to be defined by predicates in terms of an existing data type. It would be meaningless just to 'call' such operations from within, say, a post-condition. The use of operations in one (state-based) data type from another is facilitated by *quoting* their pre- and post-conditions. That is, their 'effect' is shown by using their post-conditions with appropriate arguments (and their applicability is shown by using their pre-conditions in the same way). Looking at the definition of BAG, the explanation of the signatures of pre- and post-conditions given in Section 3.4 shows that the following are intended:

$$pre\text{-}COUNT\colon\ X \times Bag \to \mathbb{B}$$
$$post\text{-}COUNT\colon\ X \times Bag \times \mathbb{N} \times Bag \to \mathbb{B}$$
$$pre\text{-}ADD\colon\ X \times Bag \to \mathbb{B}$$
$$post\text{-}ADD\colon\ X \times Bag \times Bag \to \mathbb{B}$$

Using these truth-valued functions, and:

$$init\text{-}Bag\colon\ \to Bag$$

it is easy to define a module for a data type which uses BAG to build a collection of bags:

module *MBAG*
imports
⋮
exports
⋮
definitions
types
$Mbag = D \xrightarrow{m} Bag;$
-- The bags can be stored in a map indexed by elements of D
state
State **of** m: *Mbag*
init $(mk\text{-}State(m_0)) \triangleq m_0 = \{\,\}$
end ;
operations

-- An operation which enlarges the collection of bags could be specified:

MNEW $(w: D)$
ext wr m : *Mbag*
pre $w \notin \mathbf{dom}\, m$
post $m = \overleftarrow{m} \cup \{w \mapsto init\text{-}Bag()\}$

-- Counting within a bag can be shown by quotation:

MCOUNT $(w: D, e: X)$ $c: \mathbb{N}$
ext rd m : *Mbag*
post $post\text{-}COUNT(e, m(w), c)$

-- An operation which adds an element to a stated bag is:

MADD $(w: D, e: X)$
ext wr m : *Mbag*
post $\exists b \in Bag \cdot post\text{-}ADD(e, \overleftarrow{m}(w), b) \wedge m = \overleftarrow{m} \dagger \{w \mapsto b\}$

end *MBAG*

Remember that the specification of these three operations is insulated from any reformulation of the specification of *Bag* itself. This ability to adapt with such changes is the essential feature of a module notation.

Annotating specifications

It is obvious that formal specifications for large systems are likely to be long. The formal description of the PL/I language contains about 120 pages of formulae. Care and preparedness to rewrite parts of such a specification can make the model itself far easier to understand. The tasteful use of natural language annotations can also make it much easier for a reader to begin to understand a large formal specification. There are several possible styles of annotation:

- in-line comments – as in programming languages;
- numbered formulae lines with annotations which relate to line numbers placed after each block of formulae;
- careful decomposition into abstract data types with text introducing each such separate concept.

The first of these options (marked by --) is used in the example above. For this textbook, the third option has been used in preference to a somewhat heavy alternative with:

annotation
"..."
end annotation

which is possible in the BSI syntax. It is likely that the development of appropriate machine support will make a form of the second approach much more attractive.

Compiler dictionary example

The idea of quoting the post-condition of one operation in the specification of another is most often used for applying operations to parts of a state. The following specification is a case where this is used in a way which neatly separates (state-based) data types. A compiler dictionary can be used to record attribute information about identifiers. Many texts on compiler writing refer to such a dictionary as a 'symbol table'. Information is added to a local dictionary when the declarations of a block are processed, and this information can be looked up when code is to be generated for the statements in the block. In a block-structured language like ALGOL, the declaration information for different blocks must be kept separately. The attributes of a non-local identifier must be found by looking in the local dictionaries for outer blocks, but the appropriate declaration is always the one in the closest surrounding block. Here, it is assumed that the compiler is one-pass and that entering (and leaving blocks) causes the creation of empty (and the destruction of) local dictionaries. The reader should have no difficulty in specifying such a system as one module. Here, the specification is presented by first defining a module with the operations on the local dictionaries.

module $LDICT$
parameters types $Id, Attribs$
exports operations
  $STOREL$: $Id \times Attribs \xrightarrow{o}$,
  $ISINL$: $Id \xrightarrow{o} \mathbb{B}$,
  $LOOKUPL$: $Id \xrightarrow{o} Attribs$
definitions
types
  $Ldict = Id \xrightarrow{m} Attribs$;
state
  $State$ **of** ld: $Ldict$
  init $(mk\text{-}State(ld_0)) \triangleq ld_0 = \{\,\}$
end ;
operations

$STOREL\ (i{:}\ Id, a{:}\ Attribs)$
ext wr $ld\ :\ Ldict$
pre $i \notin$ **dom** ld
post $ld = \overleftarrow{ld} \cup \{i \mapsto a\}$

$ISINL\ (i{:}\ Id)\ r{:}\ \mathbb{B}$
ext rd $ld\ :\ Ldict$
post $r \ \Leftrightarrow\ i \in$ **dom** ld

$LOOKUPL\ (i{:}\ Id)\ r{:}\ Attribs$
ext rd $ld\ :\ Ldict$
pre $i \in$ **dom** ld
post $r = ld(i)$

end $LDICT$

The definition *Ldict* can be regarded as a state-based data type whose module can be used in the definition of the main operations.

$$
\begin{array}{l}
\textbf{module } CDICT \\
\textbf{parameters types } Id, Attribs \\
\textbf{exports operations} \\
\quad ENTER\colon () \xrightarrow{o}, \\
\quad LEAVE\colon () \xrightarrow{o}, \\
\quad STORE\colon Id \times Attribs \xrightarrow{o}, \\
\quad ISLOC\colon Id \xrightarrow{o} \mathbb{B}, \\
\quad LOOKUPC\colon Id \xrightarrow{o} Attribs \\
\textbf{definitions} \\
\textbf{types} \\
\quad Cdict = LDICT.State^*; \\
\textbf{state} \\
\quad State \textbf{ of } cd\colon Cdict \\
\quad \textbf{init } (mk\text{-}State(cd_0)) \triangleq cd_0 = [\,] \\
\textbf{end };
\end{array}
$$

$$
\begin{array}{l}
STORE\ (i\colon Id, a\colon Attribs) \\
\textbf{ext wr } cd\ :\ Cdict \\
\textbf{pre } cd \neq [\,] \wedge pre\text{-}STOREL(i, a, \textbf{hd } cd) \\
\textbf{post } \exists ld \in Ldict \cdot post\text{-}STOREL(i, a, \textbf{hd } \overleftarrow{cd}, ld) \wedge cd = [ld] \curvearrowright \textbf{tl } \overleftarrow{cd}
\end{array}
$$

$$
\begin{array}{l}
ISLOC\ (i\colon Id)\ r\colon \mathbb{B} \\
\textbf{ext rd } cd\ :\ Cdict \\
\textbf{pre } cd \neq [\,] \\
\textbf{post } post\text{-}ISINL(i, \textbf{hd } cd, r)
\end{array}
$$

$$
\begin{array}{l}
LOOKUPC\ (i\colon Id)\ r\colon Attribs \\
\textbf{ext rd } cd\ :\ Cdict \\
\textbf{pre } \exists j \in \textbf{inds } cd \cdot pre\text{-}LOOKUPL(i, cd(j)) \\
\textbf{post let } k = mins\{j \in \mathbb{N} \mid pre\text{-}LOOKUPL(i, cd(j))\} \textbf{ in} \\
\qquad post\text{-}LOOKUPL(i, cd(k), r)
\end{array}
$$

$$
\begin{array}{l}
ENTER\ () \\
\textbf{ext wr } cd\ :\ Cdict \\
\textbf{post } cd = [init\text{-}Ldict()] \curvearrowright \overleftarrow{cd}
\end{array}
$$

$LEAVE$ ()
ext wr cd : $Cdict$
pre $cd \neq [\,]$
post $cd = \mathbf{tl}\ \overleftarrow{cd}$

end $CDICT$

As explained above, quoting the pre- and post-conditions makes it possible to change the internal detail of *Ldict* without having to change *CDICT*.

File stores

The next example, as well as making use of operation quotation, also illustrates the extent to which the state model of a specification can be used to investigate the possibilities of an architecture. It is possible to discern the architecture of a system without reading the whole description. With experience, the underlying state-like objects of a definition can be understood to define the overall architecture. In the (120 page) PL/I description, the so-called 'semantic objects' occupy about five pages. A clear understanding of this material ties down many facets of the language without having to read all of the fine detail. Here, the importance of the state is shown by the development of a series of vignettes of file systems. Suppose – for all of the definitions – the internal structure of a file is of no interest. (A *File* might be an unstructured sequence of bytes or it might have a richer structure. In the latter case, it could be treated as a separate data type.) If files are to be accessed, they must be named. Thus the state of the most trivial file system is:

$$Trivfs = Name \xrightarrow{m} File$$

It would be possible to define a range of operations on this state (e.g. *CREATE*, *DELETE*, *COPY*); but it is more interesting to observe what cannot be done. It is obvious from the properties of maps that no two different files can have the same name. If two users wish to have separate name spaces, the state of this file system is not rich enough. This observation can be made without an exhaustive search of operation specifications or – as here – even before the effort is expended to write such definitions.

It is not difficult to extend the state in a way which permits nested directories. For example:

$$Nestedfs = Dir$$

$$Dir = Name \xrightarrow{m} Node$$

$$Node = Dir \cup File$$

This allows separate users to utilize the same name in the way that *Unix*$^{(tm)}$ directories support. Here again, operations could be specified on this state; but one can also see what still cannot be done in any system based on this state. In particular, it is not possible to share the same file via two different name paths. Here sharing is taken to imply that if a user changes the file by one path, the change will appear when the file is accessed via another path. There is a standard way of establishing such sharing patterns in specifications and that is to introduce some intermediate link, or surrogate, (here a file identifier – *Fid*). Thus:

$$\begin{array}{lll} Sharedfs :: & root & : Dir \\ & filem & : Fid \xrightarrow{m} File \end{array}$$

$$Dir = Name \xrightarrow{m} Node$$

$$Node = Dir \cup Fid$$

It is now clear, from the state above, that files can be shared in the sense that different paths can lead to the same file identifier.

Having developed a plausible state, some operations are given. An operation to show the contents of a directory is:

$$\begin{array}{l} SHOW\ ()\ m{:}\,Dirstatus \\ \textbf{ext rd}\ d\ :\ Dir \\ \textbf{post}\ m = \{nm \mapsto (\textbf{if}\ d(nm) \in Dir\ \textbf{then}\ \ \textbf{DIR}\ \textbf{else}\ \ \textbf{FILE}) \mid \\ \qquad\qquad\qquad\qquad nm \in \textbf{dom}\ d\} \end{array}$$

$$Dirstatus = Name \xrightarrow{m} \{\text{FILE}, \text{DIR}\}$$

An operation to create a new directory is:

$$\begin{array}{l} MKDIR\ (n{:}\,Name) \\ \textbf{ext wr}\ d\ :\ Dir \\ \textbf{pre}\ n \notin \textbf{dom}\ d \\ \textbf{post}\ d = \overleftarrow{d} \cup \{n \mapsto \{\,\}\} \end{array}$$

Once more, a somewhat optimistic pre-condition is given. A way to avoid this and indicate exceptions is described in the next section.

It is then possible to quote these operations in order to form other, more global, operations such as:

$$Path = Name^*$$

$SHOWP\ (p{:}\ Path)\ m{:}\ Dirstatus$
ext rd $d : Node$
pre $d \in Dir \wedge$
$\quad (p = [\,] \vee$
$\quad p \neq [\,] \wedge \mathbf{hd}\, p \in \mathbf{dom}\, d \wedge \textit{pre-SHOWP}(\mathbf{tl}\, p, d(\mathbf{hd}\, p)))$
post $p = [\,] \wedge \textit{post-SHOW}(d, m) \vee$
$\quad p \neq [\,] \wedge \textit{post-SHOWP}(\mathbf{tl}\, p, d(\mathbf{hd}\, p), m)$

The claim being made here is that the state can convey a great deal of useful information about a system. This is, of course, only true where the state is well chosen. An alternative state for the system above is:

$$\begin{aligned} Sharedfs :: \ & access : Path \xrightarrow{m} Fid \\ & filem \ \ : \ldots \end{aligned}$$

A state which has basically this form is chosen by Carroll Morgan and Bernard Sufrin in their contribution to [Hay87]. It is instructive to compare this with the earlier state. The most obvious comment is that this would complicate the definition of *SHOW*. It is also clear that there would have to be a complicated invariant on this state. This having been said, it is possible to define all of the operations on such a state. What is left is the observation that this second state conveys a much less clear picture of the intended system than the first state shows.

In specifying even moderately sized systems, one must be prepared to discard possible states as it becomes clear that some operations or invariants become inconvenient. In this way the state comes to be the essence of the specification, and can then provide much insight.

The point about the knowledge derivable from a well-chosen state can also be made by counter-example. The ECMA/ANSI specification of PL/I is based on a formal model. As with the the Vienna definition ([BBH$^+$74]), the state is given formally and is rather short. However, it contains many sequences but no sets since there was some feeling that sets might be too abstract for the standards organization! On checking, one finds that no use is made of the order of some of these sequences. To know which sequences do convey essential order one has to inspect the remaining 300 or more pages of the definition. Thus information which could have been made clear in the state is dispersed over the whole definition.

The reference to standards activities presents an appropriate point to contrast the terms 'specification' and 'description'. Although the former term has been used in this book, it should be noted that it really relates to an official status; the term 'description' is often the more appropriate one. It is, of course, the hope of the author that it will become ever more frequent for standards committees to adopt formal specifications.

Exercise 9.1.1 A very simple diary reminder system can be specified around:

$$Diary = Date \xrightarrow{m} Task^*$$

Specify an operation which adds a *Task* for a given *Date* (do not assume that the *Date* is already in the *Diary*). This operation should then be quoted in the specification of an operation for a given user in a state:

$$Diarysys = Uid \xrightarrow{m} Diary$$

Exercise 9.1.2 (*) Exercise 9.1.1 introduces a trivial diary system. Write down a reasonable list of requirements and then develop (using separate data types and combining them) a specification of a realistic computer-based diary manager.

9.2 Exceptions

Some of the operation specifications given in this book have overly restrictive pre-conditions. It is pointed out, in earlier chapters, that this might well be realistic for operations which are used within a system: essentially, the environment of the operations ensures that the pre-condition is fulfilled. There are, however, operations which might be invoked in a way which makes such restrictive pre-conditions unrealistic. This section introduces some notational extensions which can be used to record *exceptions*.

It is worth introducing the extended notation by considering the effect of trying to avoid it. Suppose it were wished to make the *DEQUEUE* operation of Section 7.1 total in the sense that it did not have the pre-condition given there. It would be possible to write:

$DEQUEUE$ () e: [Qel]
ext wr q : $Queue$
pre true
post $\overleftarrow{q} \neq [\,] \wedge \overleftarrow{q} = [e] \curvearrowright q \vee$
$\quad\overleftarrow{q} = [\,] \wedge \overleftarrow{q} = q \wedge e = \textbf{nil}$

Here, the return of the **nil** value is taken to indicate an error.

It would also be possible to base the specification on the signature:

$DEQUEUE$() e: [Qel] err: [QUEUEEMPTY]

There are several observations which can be made about this approach. Perhaps the most obvious problem is that the specification of the normal case can become submerged in detail. But, this may not be the worst problem. This style of specification forces decisions about how errors are to be shown. In some programming languages (e.g. Pascal) it might be necessary to return an extra result, or a distinguished value, in order to indicate an exception; but there are languages (e.g. PL/I, Ada, ML) which contain explicit exception mechanisms. As far as possible, it is worth postponing commitments to implementation languages. It should certainly not be necessary to choose an implementation language in order to record a specification.

The requirements for exception specifications thus include the ability to separate exceptional cases from the normal and an avoidance of commitment as to how exceptions are to be signalled.

BSI-VDM has adopted one possible notation which adds error clauses to operation specifications. In general, the format becomes (where the r_i are logical expressions):

$$
\begin{array}{l}
OP\ (i{:}\ Ti)\ r{:}\ Tr \\
\textbf{ext wr}\ v\ :\ Tv \\
\textbf{pre}\ p \\
\textbf{post}\ r_0 \\
\textbf{errs}\ COND_1{:}\ c_1 \rightarrow r_1 \\
\qquad\ \ COND_2{:}\ c_2 \rightarrow r_2
\end{array}
$$

The condition names ($COND_i$) can be taken to be the name of the exception: how this is returned is a matter for the implementation. Leaving aside the name, the specification can be explained by its translation to:

$$
\begin{array}{l}
OP\ (i{:}\ Ti)\ r{:}\ Tr \\
\textbf{ext wr}\ v\ :\ Tv \\
\textbf{pre}\ p \vee c_1 \vee c_2 \\
\textbf{post}\ p \wedge r_0 \vee c_1 \wedge r_1 \vee c_2 \wedge r_2
\end{array}
$$

Some consequences of this translation should be noted. Firstly, the pre-condition is effectively widened by the conditions on the error clauses. Secondly, the form of the given post-condition is, in general, non-deterministic: if both c_1 and c_2 are true, either exception can be signalled and the corresponding state transformation can occur. Even if both the normal case and an exception can arise, this translation does not fix the effect. In practice, it is wise to make the normal and exception conditions mutually disjoint, but there are advantages in not determining which of several exceptions should occur since it leaves an implementation some freedom to choose the order in which tests are made. If it is important which exception is signalled, the conditions can again be made mutually exclusive.

The above example could now be written:

$DEQUEUE\ ()\ e{:}\,[Qel]$
ext wr $q\ :\ Queue$
pre $q \neq [\,]$
post $\overleftarrow{q} = [e] \curvearrowright q$
errs QUEUEEMPTY: $q = [\,] \rightarrow q = \overleftarrow{q} \wedge e = \mathbf{nil}$

A very common special case is where the exceptions do not cause a state change. This is, in fact, a very desirable property of a system. It is possible to further economize on notation by recognizing this special case.

Exercise 9.2.1 Rewrite the specification of Exercise 4.4.2 on page 108 using the exception notation.

Exercise 9.2.2 Write the exception specifications (where appropriate) for the stack examples given in Exercise 7.3.2 on page 178.

Exercise 9.2.3 (*) Extend the specification of the file system given in Section 9.1 so that operations *MKDIR* and *SHOWP* handle exceptions. Now that the exception notation is understood, it is reasonable to define other operations on *Sharedfs*. Consider new features (e.g. security/authority, stored path names) and show how these affect the state. In all operation definitions, attempt to use operation quotation to separate the data types.

9.3 Implementation bias in models

The remainer of this chapter addresses special issues about the concept of data types.

Biased model of queues

The concept of *implementation bias* is most simply introduced by example. Section 7.1 begins by introducing a specification of a queue based on objects *Queue*. A specification which defines the identical behaviour is:

$$\begin{aligned} Queueb\ ::\ & s\ :\ Qel^* \\ & i\ :\ \mathbb{N} \end{aligned}$$
inv $(mk\text{-}Queueb(s, i)) \triangleq i \leq \mathbf{len}\, s$
init $q_0 = mk\text{-}Queueb([\,], 0)$

$ENQUEUE\ (e{:}\,Qel)$
ext wr $s\ :\ Qel^*$
post $s = \overleftarrow{s} \curvearrowright [e]$

$DEQUEUE$ () e: Qel
ext rd s : Qel^*,
wr i : $\mathbb{N}$
pre $i <$ **len** s
post $i = \overleftharpoon{i} + 1 \land e = \overleftharpoon{s}(i)$

$ISEMPTY$ () r: $\mathbb{B}$
ext rd s : Qel^*,
rd i : $\mathbb{N}$
post $r \Leftrightarrow (i = \mathbf{len}\ s)$

The model in this specification keeps unnecessary history of the queue and this is intuitively wrong. This intuitive concern can be made more formal by considering retrieve functions. A retrieve function can easily be constructed in one direction:

$$retr\text{-}Queue : Queueb \to Queue$$
$$retr\text{-}Queue(mk\text{-}Queueb(s, i)) \quad \triangleq \quad s(i+1, \ldots, \mathbf{len}\ s)$$

Thus:

$$retr\text{-}Queue(mk\text{-}Queueb([a, b, c, d], 1) = [b, c, d]$$

But a retrieve function *cannot* be constructed in the other direction because the unnecessary history information cannot be found in $Queue$. This discloses why the problem is referred to as 'implementation bias'. Using the reification proof obligations given in Sections 8.1 and 8.2, the $Queueb$ model is biased towards (proving correct) implementations which retain at least as much information. An implementation which keeps even more history (e.g. the exact order of $ENQUEUE/DEQUEUE$ operations) can be proved correct: a retrieve function can be constructed to $Queueb$.

It is important to realize that the behaviour of the operations on $Queueb$ is the same as that on $Queue$. Thus it is possible to show that the operations on the former model those on the latter. It is only the acceptability of the $Queueb$ model as a specification which is being challenged. As an implementation, its behaviour is as required.

The bias of the $Queueb$ specification is a criticism of a specific model. Is it also an indication of a weakness of the model-oriented approach to specification? There are certainly some computer scientists who have argued in this direction. The proof rules shown below permit even a biased model to be used as a starting point for development. More importantly, it is normally possible to avoid bias. Moreover, it is possible to prove that bias is absent.

A test for bias

The problem which is to be avoided is that an implementation is invented such that a retrieve function from its states to those of the specification cannot be constructed. This itself cannot serve as a test of a specification since it requires consideration of possible implementations. The problem with the storage of unnecessary history information in *Queueb* can, however, be described in another way: the information is unnecessary precisely because it cannot be detected by any of the available operations. The following definition is therefore given:

> A model-oriented specification is based on an underlying set of states. The model is biased (with respect to a given set of operations) if there exist different elements of the set of states which cannot be distinguished by any sequence of the operations.

In terms of the example above, there is no way of distinguishing between:

$$mk\text{-}Queueb([a, b, c], 1) \qquad \text{and} \qquad mk\text{-}Queueb([b, c], 0)$$

The precision of this test makes it possible to use it as a proof obligation. A model is *sufficiently abstract* (to be used as a specification) providing it can be shown to be free of bias.

It is important to realize that the bias test is relative to a particular set of operations. The *Queue* model of Section 7.1 is unbiased for the collection of operations given there. However, for a different set of operations, *Queue* is a biased model. For example, if the *DEQUEUE* operation were replaced by one which only removed, but did not show, the removed value:

$REMOVE$ ()
ext wr q : $Queue$
pre $q \neq [\,]$
post $q = \mathbf{tl}\ \overleftarrow{q}$

there is no operation which could distinguish between the queues:

$$[a, b] \qquad [b, a] \qquad [c, d]$$

An unbiased model[1] for this collection of operations is a natural number which records the number of elements in the queue. Furthermore, if the *REMOVE* operation is entirely discarded, the only distinction which can be detected is between empty and non-empty queues. A sufficiently abstract model for this restricted set of operations is a single Boolean value.

[1]Another term which is used in connection with bias is 'full abstraction'. A specification can be said to be fully abstract (with respect to a given set of operations) if it is not biased.

The test for bias was discovered after many model-oriented specifications had been written. Since then, it has been applied to a number of specifications which were written without its guidance. The experience is that very few specifications were found to have been biased. Even those which were revolve around rather subtle problems. It is therefore not envisaged that this proof obligation need normally be discharged in a formal way. The concept of sufficient abstractness is more likely to be useful in general discussions about alternative models. One cause of failure is where an invariant on the specification state has been overlooked. It must be understood that there is not a unique sufficiently abstract model for any particular application. Different models can pass the bias test. With such a class of models, it will be possible to construct retrieve functions in either direction between any pair.[2]

Among the class of unbiased models, some are more complex than others. Consider, for example, a problem in which a set can be used to define the needed operations. A model based on a list is likely to be biased – state values might, for instance, store a history of the order of operations which cannot be detected. It is, however, possible to reduce the equivalent states to single values by adding an invariant. If the elements of a list are required to be in a particular order (e.g. numeric order), there is then a one-to-one correspondence between the lists (with invariants) and sets. The restricted lists are not biased – but the model is certainly more complicated.

This appears to suggest another criterion for the choice of models: in general, it is better to choose a state which minimizes the need for invariants. There are, however, exceptions to this guideline, and the reader is reminded of the discussion in Section 8.3 about the use of more than one isomorphic model. One such model may have a minimum invariant while another might be more complicated; if the more complicated model makes some operations easier to define, it can pay its way.

All of the above comments about bias relate to the choice of models for specifications. Reification certainly brings in bias. In fact, the commitments which are made by the designer are intended to introduce implementation bias. At each successive step of data reification, the range of models (which can be justified using retrieve functions) is intentionally reduced. The designer's goal is to arrive at a final, single implementation.

[2]Technically, the unbiased models form an isomorphism class – they partition the possible behaviour histories into equal sets.

More general proof rules

The remainder of this section is concerned with proof rules for handling development from biased specifications. There are two reasons for what may appear to be a *volte-face*. Firstly, bias may occur by accident. Although the point is made above that the investment of rewriting specifications (even several times) is likely to pay off in clarity, not all industrial environments are prepared to accept this austere advice. It is shown below that there are ways of handling development from biased specifications. Some users of formal methods may choose to employ the more general reification rules.

The other reason for presenting ways of handling the more general situation is that there are places where a specification which is technically biased should be used! The most common situation where (technical) bias is justified is when the full extent of the set of operations is unknown. Michael Jackson presents examples in his books (see, for example, [Jac83]) in which attempts to tailor the state too closely to a particular collection of operations makes subsequent extension all but impossible. It is argued in Section 8.3 that the state represents the essence of the operations. When the operations are not a fixed collection, the state must be chosen to be the essence of the application itself. The extent to which this rather vague goal is achieved, will govern the difficulty of subsequent modifications.

There are some cases where a biased state can lead to a clearer specification than an unbiased one. Such cases are rare. An example is forming the average and standard deviation of a collection of values. An obvious specification first stores all of the numbers; to avoid bias, a specification has to rely on subtle properties of the definitions.

There is one more case where the state of a specification has more information than that of correct implementations. This is the most technical of the cases. It is sometimes necessary for the state of the specification to contain information which defines the range of non-determinacy. An implementation which resolves the non-determinism in a particular way may need less information in the state. A representative example of this situation can be built around a symbol table. A specification can use the state:

$$Symtab = Sym \xrightarrow{m} Addr$$

$$Addr = \mathbb{N}$$

A non-deterministic operation to allocate addresses is:

$ALLOC\ (s\colon Sym)\ a\colon Addr$
ext wr $t\ :\ Symtab$

$$\begin{array}{l}
\textbf{pre } s \notin \textbf{dom } t \\
\textbf{post } a \notin \textbf{rng } \overleftarrow{t} \wedge t = \overleftarrow{t} \cup \{s \mapsto a\}
\end{array}$$

An implementation of this specification can use:

$$\begin{array}{l}
Symtabrep = Sym^* \\
\textbf{inv } (t) \triangleq is\text{-}uniques(t)
\end{array}$$

with:

$$\begin{array}{l}
ALLOCr\ (s{:}\,Sym)\ a{:}\,Addr \\
\textbf{ext wr } t : Symtabrep \\
\textbf{pre } s \notin \textbf{elems } t \\
\textbf{post } t = \overleftarrow{t} \frown [s] \wedge a = \textbf{len } t
\end{array}$$

An attempt to use the reification rules of Chapter 8 may lead to the retrieve function:

$$\begin{array}{l}
retr\text{-}Symtab : Symtabrep \to Symtab \\
retr\text{-}Symtab(t) \quad \triangleq \quad \{t(i) \mapsto i \mid i \in \textbf{inds } t\}
\end{array}$$

But this clearly shows that *Symtabrep* is not adequate: any value of *Symtab* with gaps in the allocated addresses cannot be represented. The need to provide a general model in the specification was to express the potential non-determinacy; the decision to yield particular addresses in the implementation renders this information redundant.

One way of handling this situation is to generate a special proof obligation for steps of development which reduce non-determinacy in this way. Although straightforward, this avenue is not pursued here since the more general proof rule covers this somewhat rare case.

It has been made clear that the behaviour of a data type is what is to be specified and verified. But there are steps of reification which cannot be proved correct by the rules of Chapter 8 even though the putative implementation manifests the same behaviour as the specification. Thus, it is clear that the given rules are too weak in the sense that they are sufficient but not necessary. Although they cover a very large percentage of the development steps which one is likely to meet, it is useful to know the more general rule.

The key to the more general rule is to realize that the retrieve function can revert to a relation. The proof rules of Chapter 8 capitalized on the one-to-many situation brought about by the lack of bias. If this restriction no longer applies, the many-to-many situation can be represented by:

$$rel{:}\,Abs \times Rep \to \mathbb{B}$$

Suppose the biased *Queueb* from the beginning of this section were to have been used in a specification; the relation to *Queue* (now taken as an implementation!) could be recorded by:

$$\begin{array}{l} \mathit{rel\text{-}Queue} : \mathit{Queueb} \times \mathit{Queue} \to \mathbb{B} \\ \mathit{rel\text{-}Queue}(\mathit{mk\text{-}Queueb}(l, i), s) \quad \triangleq \quad l(i+1, \ldots, \mathbf{len}\, l) = s \end{array}$$

With the more general rules, there is no adequacy proof obligation. The domain rule is similar to that of Chapter 8:

$$\mathit{rel\text{-}Queue}(qb, q) \wedge \mathit{pre\text{-}OPA}(qb) \;\Rightarrow\; \mathit{pre\text{-}OPR}(q)$$

Notice that *OPA* works on *Queueb* and *OPR* on *Queue*. The result rule is:

$$\begin{array}{l} \mathit{rel\text{-}Queue}(\overleftarrow{qb}, \overleftarrow{q}) \wedge \mathit{pre\text{-}OPA}(\overleftarrow{qb}) \wedge \mathit{post\text{-}OPR}(\overleftarrow{q}, q) \;\Rightarrow \\ \qquad\qquad \exists qb \in \mathit{Queueb} \cdot \mathit{post\text{-}OPA}(\overleftarrow{qb}, qb) \wedge \mathit{rel\text{-}Queue}(qb, q) \end{array}$$

Proofs using these results are left as exercises. In general, they become more difficult than proofs using the rules of Chapter 8, if for no other reason than the appearance of the existential quantifier.[3] It is also necessary to handle initial states – the rule should be obvious from the corresponding rule in Chapter 8.

There are other ways of handling situations where bias occurs in the specification. In early work on formal development of compilers, Peter Lucas (see [Luc68]) showed how *ghost variables* can be erected in the implementation state. These variables initially retain any redundant information but can be disposed of once there are no essential references to them.

Exercise 9.3.1 Justify *Queueb* as an implementation with respect to the *Queue* specification given in Section 7.1.

Exercise 9.3.2 Design an implementation of the queue operations which retains the full history of the queue. Since this is even more information than is contained in *Queueb*, it is possible to use the (biased) *Queueb* operations as a specification. Sketch a justification which illustrates this fact.

Exercise 9.3.3 Justify *Queue* as an implementation of the specification *Queueb* – since this latter is biased, the more general proof rule of this section must be used.

Exercise 9.3.4 Write a biased specification of a stack (cf. Exercise 7.3.2 on page 178).

[3] It is, however, the existential quantifier in the result rule which ensures that this more general rule covers the sort of non-deterministic situation which arose in the symbol table example.

Exercise 9.3.5 The first conjunct in *invp* (Section 4.2) bars an empty set from a partition. One reason for needing this is the equivalence relation specification mentioned in Section 4.4. Discuss the problem in terms of bias.

Exercise 9.3.6 Outline the proof of the operation *ALLOC* for the *Symtabrep* representation of *Symtab*. The proof obligation will have to use the more general rule.

Exercise 9.3.7 (*) It is standard practice to define the rational numbers as a pair of integers. Set up such a model and define some functions (e.g. addition of rationals). Discuss the problem of bias in this, functional, context.

9.4 Property-oriented specifications of data types

The preceding section should have allayed any fears about being forced into over-specification in the model-oriented approach. But the concern has been fruitful in that it is one of the stimuli which have led computer scientists to develop a way of specifying data types without using a model at all. The idea goes back to the concept of a data type being a pattern of behaviour. The *property-oriented* approach[4] to specifying data types defines properties of these behaviours by a series of equations.

This section does not aim to provide a course on the property-oriented approach: it only explores the presentations, given in Chapters 4 to 7 above, for the basic data types, and discusses the role of property-oriented specifications in data types required in applications.

Properties of collections

It has already been seen that the generators for sequences, etc. present a convenient basis for proofs. In Chapter 7 the generating operators are given as $[\,]$ and *cons* $(X \times X^* \to X^*)$. There, these generators are closed off by an induction rule; in a property-oriented specification, the induction rule is subsumed by the *interpretation* which is ascribed to the equations.

The properties of concatenation can be given by the equations:

$$[\,] \curvearrowright t = t$$
$$cons(e, t_1) \curvearrowright t_2 = cons(e, t_1 \curvearrowright t_2)$$

[4]What is referred to here as the 'property-oriented approach' is known in the literature under a variety of different names: '(equational) presentations of algebras'; 'the axiomatic approach' (viewing the equations as axioms); or even '*the* algebraic approach'.

Viewed innocently, the equalities in these equations indicate that terms of one form can be rewritten into another form. In the *initial interpretation*, the objects denoted by terms are equal exactly when the terms can be proven to be equal from the equations. This appears to be a very plausible position but it is not the only one possible. In fact, the consequence that inequalities can be established only by showing that a term cannot be deduced is extremely aggravating.

The reader should remember that Chapter 4 introduced the set constructors $(\{\,\}, \odot)$ by equations which, apart from the symbols, are identical to those given for sequences. But clearly the sets denoted by the terms:

$$e_1 \odot (e_2 \odot \{\,\}) \qquad \text{and} \qquad e_2 \odot (e_1 \odot (e_1 \odot \{\,\}))$$

should be equal. In the initial interpretation, it is necessary to add extra equations in order to ensure that term equality defines object equality. The need for these equations can be avoided in the alternative final interpretation of such equations. In the *final interpretation*, objects are assumed to be equal unless the terms which they denote can be proved unequal. The normal way to show that terms are unequal is by using some external (already understood) type. In the final interpretation for sets, there would be no need to add the absorptive and commutative equations for $\odot$. It would, however, be necessary to add some operators in order to prevent the complete collapse of the value space. In this case the membership operator could be used (see below).

To make these points clear, the specification of three data types (sequences, bags and sets) are considered under the two interpretations. A useful concept in this discussion is a term algebra. Given some set of operators, the *term algebra* is the set of all terms which can be generated such that each application respects the types. (This set of terms could be formalized using an abstract syntax.)

With the types:

$$\begin{aligned}&\mathbf{null}: \mathit{Colln}\\&\odot: X \times \mathit{Colln} \to \mathit{Colln}\end{aligned}$$

the initial model of *Colln* is exactly the sequence values. In fact, the term algebra of these generators can be thought of as providing a model on which other operators (e.g. concatenation) can be defined. The initial interpretation of these equations is a natural match for sequences. The operator $+$ (of type $\mathit{Colln} \times \mathit{Colln} \to \mathit{Colln}$) which satisfies:

$$\begin{aligned}&\mathbf{null} + c = c\\&(e \odot c_1) + c_2 = e \odot (c_1 + c_2)\end{aligned}$$

automatically becomes sequence concatenation.

The same generators can be used for the bags, but here the term algebra for the operators above needs breaking into equivalence classes. Since bags do not have the concept of the order of their elements, any terms which differ only by position denote the same objects. This fact can be captured by the single equation:

$$e_1 \odot (e_2 \odot b) = e_2 \odot (e_1 \odot b)$$

This equation can be used (cf. Section 4.2) to show the commutative properties of bag operators defined over these generators (e.g. $+$ on bags becomes union). In some sense, the initial interpretation is not such a good match for bags. The values now correspond to sets of terms. One possibility is to think of choosing a representative member of each equivalence class (e.g. relying on some ordering over the elements).

For sets, the equivalence classes have to be made yet coarser. The necessary effect can again be achieved by adding one more equation:

$$e \odot (e \odot s) = e \odot s$$

One can picture what has been done by considering the set of all possible terms formed from **null**/$\odot$ and partitioning this set into equivalence classes as indicated by the equations defining the commutativity and absorption of $\odot$. To each such (infinite) set of terms, there corresponds one set value which is denoted by each of the terms.

In the final interpretation, the equivalence classes of terms are as coarse as possible. Thus, the final interpretation comes closest to matching sets. However, there is nothing about the generating operators which prevents even the terms:

$$e_1 \odot \mathbf{null} \qquad\qquad \mathbf{null}$$

from being treated as equal. The danger is that all terms are in one equivalence class. This is avoided by adding an operator which yields values in another type. For sets, an appropriate operator is membership. Equations for $\in$ (of type $X \times \mathit{Colln} \to \mathbb{B}$) which show:

$$\neg\,(e \in \mathbf{null}), \qquad e \in (e \odot s), \qquad e \in s \;\Rightarrow\; e \in (e' \odot s)$$

result in the appropriate algebra.

For bags (cf. Section 6.3), the equivalence relation on terms must be made finer. This can be done by replacing the membership operator with *count* (of type $X \times \mathit{Colln} \to \mathbb{N}$), where:

$$\begin{array}{ll} \mathit{count}(e, \mathbf{null}) = 0 & \\ \mathit{count}(e_1, (e_2 \odot b)) = \mathit{count}(e_1, b) & e_1 \neq e_2 \\ \mathit{count}(e_1, (e_1 \odot b)) = \mathit{count}(e_1, b) + 1 & \end{array}$$

The equivalence class so defined still has:

$$e_1 \odot (e_2 \odot t) \qquad e_2 \odot (e_1 \odot t)$$

in the same partition since they cannot be proved unequal. To make *Colln* behave, in the final interpretation, like sequences, one could add **hd** $(Colln \to X)$ with:

$$\mathbf{hd}\,(e \odot c) = e$$

There are then at least two interpretations of a set of equations. Clearly, if specifications of data types are to be given by properties, the interpretation must be defined.

Implementation proofs

The choice of interpretation is closely related to the question of how one shows that an implementation is correct with respect to a property-oriented specification. The obvious approach to such proofs might be to check that all terms which are in the same equivalence classes denote the same value in the implementation. Chapter 8 shows that, in an implementation, there may be several representations for the same abstract object. The equality of terms cannot, therefore, be used as the criterion for the correctness of implementations.

The (equivalence classes of) terms are, however, the basis for such implementation proofs. Where, as for sequences, the equivalence classes contain exactly one term, it is possible to use a style of implementation proof similar to that of Chapter 8 (i.e. based on retrieve functions). In the case that the equivalence classes contain more than one element, another technique is required. The basis of this technique is to define a homomorphism from the set of terms to the implementation. This is like a retrieve function in reverse. It can always be constructed (at least in the deterministic case) since the term algebra is the finest possible partition. The proof obligation is, then, to show that the equivalence classes represented by the equations are respected.

Scope of alternative methods

The remainder of this section considers the extent to which the property-oriented approach can be applied to specifications of applications. Property-oriented specifications are given by a signature part and a set of equations. The *signature* defines the syntactic information about the functions. The *equations* fix the semantics of the functions. (For the sake of definiteness, the initial interpretation is assumed.)

Just as the factorial program is a standard example for program proof methods, the stack is the standard example for data type specifications. The signature part of a property-oriented specification is:

$init\colon \to Stack$
$push\colon \mathbb{N} \times Stack \to Stack$
$top\colon Stack \to (\mathbb{N} \cup \textsc{Error})$
$remove\colon Stack \to Stack$
$isempty\colon Stack \to \mathbb{B}$

Several comments are in order. The standard texts on algebra consider functions rather than (what are called in this book) operations. It is possible to generalize functions to return more than one result and then operations can be viewed as functions which receive and deliver an extra (state) value. Here, the operation *POP* (cf. Section 7.3) has been split into two functions (i.e. *top*, *remove*). Another restriction is that functions are deterministic. Thus, the post-condition idea does not have an immediate counterpart here. Nor, at first sight, do the pre-conditions and their role in defining partial functions. There is a considerable literature on the algebraic treatment of errors in algebraic presentations of data types. In this section, special error values are used.

The semantics of the stack functions are fixed by the equations:

$top(init()) = \textsc{Error}$
$top(push(i, s)) = i$
$remove(init()) = init()$
$remove(push(i, s)) = s$
$isempty(init()) = \mathbf{true}$
$isempty(push(i, s)) = \mathbf{false}$

Only the first and third of these equations should require comment. The third is somewhat artificial in that it extends the domain of *remove* to avoid introducing an error value for stacks. The first shows when it is not possible to generate a natural-number result from *top*.

When the restrictions implied by the comments above are acceptable, one might prefer a property-oriented to a model-oriented specification because a definition without a model would appear to avoid problems like implementation bias. As is shown below, however, it is not always straightforward to find a property-oriented specification.

The reader would have no difficulty in providing a model-oriented specification of the above stacks. Nor would there be any difficulty in showing the changes required to define a queue. The signature of the property-oriented specification is also easy to change:

$init: \to Queue$
$enq: \mathbb{N} \times Queue \to Queue$
$first: Queue \to \mathbb{N}$
$deq: Queue \to Queue$
$isempty: Queue \to \mathbb{B}$

The changes to the equations are, however, less obvious. Clearly:

$$first(enq(e, init())) = e$$

but this covers only half of the corresponding stack equation (the second above). The remaining case must be specified:

$$first(enq(e_1, enq(e_2, q))) = first(enq(e_2, q))$$

A similar split is required for:

$$deq(enq(e, init())) = init()$$
$$deq(enq(e_1, enq(e_2, q))) = enq(e_1, deq(enq(e_2, q)))$$

This second equation is particularly disappointing since it has the feeling of recreating the queue in a very operational way, whereas a state automatically defines an equivalence over the histories. In fact property-oriented specifications can be thought of as being built on models. The model is the term algebra of the generating functions. This, in some sense, has more mathematical economy than introducing a separate model. But predetermining the model in this way has the disadvantage that it is sometimes less convenient than others. For stacks the model is convenient; for queues it is less so.

It is also possible that the generating functions do not provide an unbiased model. An example can be constructed for the integers with 0 and *succ* (as for the natural numbers) and a general *minus* operator: there are then many terms corresponding to each negative number.

The generators can be taken as guidance to the equations which are needed. The specific choice of equations is, however, a task requiring some mathematical sophistication. For example, sets could be introduced via the union operator and its properties. Another example is apparent if one considers the wide range of axiomatizations of propositional calculus.

There are also some technical points which must be considered. A set of equations (axioms) must be shown to be consistent and complete.[5] There are also data types which cannot be characterized by a finite set of equations (Veloso's stack – cf. Exercise 7.3.2 on page178 – is an interesting example).

Rather than criticize the property-oriented approach, the intention here is

[5] Or to define a non-trivial class of models.

to determine the correct roles for property-oriented and model-oriented specifications. It would be useful if all of the data types which were to be used in other specifications were given property-oriented specifications. This, basically, has been done in Chapters 4 to 7. The advantages of this approach include its firm mathematical framework, which is particularly needed to define type parameterization. Such specifications should, however, be constructed with great care and – at least – checked by a mathematician. The model-oriented approach can, in contrast, be used relatively safely for specifications of applications which are to be implemented (e.g. a database system). The state model itself can provide considerable insight into a system and makes it possible to consider operations separately. Given an understanding of the concept of implementation bias, it should be possible to provide model-oriented specifications which are sufficiently abstract.

A number of examples above have shown how properties can be deduced from a model-oriented specification. Such properties can be used as a check against the intuitive requirements for a system. This section shows that sets of properties can be completed in a way which elevates them to a property-oriented specification. This book adopts the position that the effort required to do this is rarely justified for applications. (The respective roles suggested here correspond closely to those for denotational and axiomatic semantics of programming languages.)

Exercise 9.4.1 Present a property-oriented specification of maps.

Exercise 9.4.2 The first person to introduce the idea of abstract syntax was John McCarthy. Make-functions (as they are called here) and selectors were presented by their properties. Experiment with this idea on some abstract syntax.

Exercise 9.4.3 It is possible to characterize the equivalence-relation specification by a property-oriented specification. Write an appropriate signature and set of equations.

10

Operation Decomposition

> I feel that controversies can never be finished ...
> unless we give up complicated reasonings in favour
> of simple calculations, words of vague and
> uncertain meaning in favour of fixed symbols ...
> every argument is nothing but an error of
> calculation. [With symbols] when controversies
> arise, there will be no more necessity for
> disputation between two philosophers than between
> two accountants. Nothing will be needed but that
> they should take pen and paper, sit down with their
> calculators, and say 'Let us calculate'.
> Gottfried Wilhelm Leibniz

In spite of the discussion of alternative approaches in Section 9.4, the main approach in this book uses specifications which are built around abstract states with a collection of operations each specified by pre- and post-conditions. Chapter 8 describes techniques by which abstract objects (particularly states) are reified onto data types which are available in the implementation language. After such reification the related operations are, however, still only specified: their pre- and post-conditions say what should be done but not how to do it. Post-conditions are not, in general, executable. The process of *operation decomposition* develops implementations (for operations) in terms of the primitives available in the language and support software.

The control constructs (e.g. **while**) which are used to link the primitive instructions can be thought of as combinators. The specific combinators available vary from one programming language to another. Here fairly general forms of the main combinators for structured coding are employed. It is interesting to note that this is the first place in this book that there is a clear commitment to procedural programming languages. Although operations are introduced in Section 3.4, all of the ideas of using abstract objects could be employed in the specification of functional programs and the data reification techniques could be applied to the arguments and results of functions.

The placing of this material on operation decomposition reflects the fact that it applies to the later stages of the design process. Other textbooks treat this material at far greater length – normally at the expense of adequate discussion of data abstraction and reification.

As the reader should by now expect, the process of operation decomposition gives rise to proof obligations. Section 10.1 introduces the proof obligations and Section 10.2 exhibits a style in which programs can be annotated with their correctness arguments. There are similarities between such texts and the natural deduction style of proof used in the preceding chapters. Ways in which these ideas can be used in the development of programs are discussed in Section 10.3 and this approach is further developed in Section 10.4 where an alternative rule for loops is given.

10.1 Decomposition rules

A specified operation might be decomposed into a **while** loop. The body of the loop might, in a simple case, contain a few assignment statements; in larger problems the body can be an operation whose specification is recorded for subsequent development. Thus operation decomposition is normally an iterative design process. The decomposition rules show the conditions under which combinations of proposed code and specifications of sub-components provide correct decompositions of a given specification: the rules facilitate showing that a design step is correct.

When a design is presented as a specific combination of (specified) sub-problems it becomes important to identify the precise nature of the claims that can be made at this stage of development. The need is for development methods which have the property that implementations which satisfy specifications of sub-components can be composed so as to satisfy the specification of a system without further proof. A *compositional* development method permits the verification of a design in terms of the specifications of its sub-programs. Thus, one step of development is independent of subsequent steps in the sense that any implementation of a sub-program can be used to form the implementation of the specification which gave rise to the sub-specification. In a non-compositional development method, the correctness of one step of development might depend not only on the fulfilment of the specifications of the sub-components but also on their subsequent development.

Sequential decomposition

Consider the following specification (in order to introduce the new concepts simply, the initial examples in this chapter use only arithmetic variables; later sections pick up some of the non-numeric applications from earlier chapters):

$MULT$
ext wr $m, n, r\ :\ \mathbb{Z}$
pre true
post $r = \overleftarrow{m} * \overleftarrow{n}$

A designer might decide that the overall task would be easier if one of the variables were definitely positive so that a loop could be designed which counted up to that value. It might also be a design decision to copy – possibly negated versions of – the variables m and n into new variables (the method for introducing new variables is not discussed in this first step). The design step could be recorded as the sequential composition of two new operations:

$MULT$: $COPYPOS$; $POSMULT$

The two operations are specified:

$COPYPOS$
ext rd $m, n\ :\ \mathbb{Z}$
  wr $mp, nn\ :\ \mathbb{Z}$
pre true
post $0 \leq mp \wedge mp * nn = \overleftarrow{m} * \overleftarrow{n}$

$POSMULT$
ext rd $mp, nn\ :\ \mathbb{Z}$
  wr $r\ :\ \mathbb{Z}$
pre $0 \leq mp$
post $r = \overleftarrow{mp} * \overleftarrow{nn}$

Hopefully, a few minutes inspection of these specifications should give the reader a feeling that the design step is correct. This concept is made completely formal below. But, before looking at the proof rules in detail, it is worth making explicit what is being claimed in such a design step. The given task is to produce a program which satisfies the specification $MULT$ (i.e. for all variables of the appropriate type which satisfy *pre-MULT*, the program must terminate and the initial/final states must satisfy *post-MULT*). If the whole development were done in one step, the designer would claim that the presented program had this behaviour. A proof of such a big step might be difficult but could theoretically be written (providing the program is indeed correct!). Here, it is assumed that the designer is more circumspect: in fact, this 'designer' obligingly makes a step of development for each inference rule which has to be covered. The decision to implement *MULT* by a composition of *COPYPOS* and *POSMULT* is equivalent to the claim that, given any code which satisfies their specifications, the combination of such code must satisfy the specification of *MULT*.[1] The reader's earlier, intuitive, check of the decomposition should have observed:

- the first operation can be applied in (at least) the states in which *MULT* is expected to work: compare *pre-COPYPOS* with *pre-MULT*;
- the second operation can safely be applied in the states which result from executing the first operation: compare *pre-POSMULT* with *post-COPYPOS* (in fact, *pre-POSMULT* records the *interface* between *COPYPOS* and *POSMULT*);
- the composition of the effects of the two operations achieves the required effect of *MULT*: compare *post-COPYPOS*/*post-POSMULT* with (recognizing which states are referred to) *post-MULT*.

This could be recorded in a proof rule which looked like:

$$\boxed{;\text{-}I}\ \frac{S_1 \ \mathbf{sat}\ (pre_1, post_1);\ S_2 \ \mathbf{sat}\ \cdots;\\ \vdots}{(S_1; S_2)\ \mathbf{sat}\ \cdots}$$

But these rules are made much easier to read by writing the assertion that S satisfies a particular *pre*/*post* as:

$$\{pre\}\ S\ \{post\}$$

This useful shorthand has no other meaning than that S is claimed, for all states which satisfy *pre*, to bring about a state transition which satisfies *post*.[2] (Notice

[1] The notion of satisfaction used, a denotational semantics, and proofs that the decomposition rules are consistent with the denotational semantics are all discussed in the *Teacher's Notes*. Furthermore, each of the programming constructs has been shown to be monotone in the satisfaction ordering which justifies the claim to compositionality.

[2] This is closely linked to the so-called 'Hoare-triples' introduced in [Hoa69]. Notice, however, that 'total correctness' is required here (i.e. termination for all states satisfying *pre*) and that

that the use of braces here has nothing to do with set notation: they are employed as comment delimiters.) Using these *triples* the inference rule for sequential composition can be stated:

$$\boxed{;\text{-}I}\ \frac{\{pre_1\}\ S_1\ \{pre_2 \wedge post_1\};\ \{pre_2\}\ S_2\ \{post_2\}}{\{pre_1\}\ (S_1; S_2)\ \{post_1 \mid post_2\}}$$

Where the composition of two post-conditions is defined:

$$post_1 \mid post_2 \triangleq \exists \sigma_i \in \Sigma \cdot post_1(\overleftarrow{\sigma}, \sigma_i) \wedge post_2(\sigma_i, \sigma)$$

(The generalization to longer sequences is straightforward.) For the example above:

$$\{pre\text{-}MULT\}\ (COPYPOS; POSTMULT)\ \{post\text{-}MULT\}$$

follows because:

$pre\text{-}MULT \;\Leftrightarrow\; pre\text{-}COPYPOS$
$pre\text{-}POSMULT$ is a conjunct of $post\text{-}COPYPOS$
$post\text{-}COPYPOS \mid post\text{-}POSMULT$
$\quad\Leftrightarrow\; \exists mp_i, nn_i \cdot mp_i * nn_i = \overleftarrow{m} * \overleftarrow{n} \wedge r = mp_i * nn_i$
$\quad\Rightarrow\; post\text{-}MULT$

Section 10.2 shows that it is not normally necessary to write such proofs in as great a level of detail as has been done for this initial example. But the reader should be aware of the advantages of such formal rules: the decomposition rules are like the rules for the logical operators in that they provide a completely sound basis, whose proofs can be mechanically checked, for the claim that particular design steps are correct.

Decomposition into conditionals

Having brought out most of the general points about decomposition inference rules in the discussion of ;-I, the other rules can be more easily covered. To illustrate the introduction of conditional statements, it is assumed that the next step of design is to decompose $COPYPOS$ as follows:

$COPYPOS$: **if** $0 \leq m$ **then** TH **else** EL

where:

---

the post-condition here is a predicate of two states. It is for this reason that VDM cannot use the Hoare rules and – more subtly – that VDM's post-conditions hook initial, rather than prime final, values.

TH
ext rd $m, n \quad : \mathbb{Z}$
  wr $mp, nn \; : \mathbb{Z}$
pre $0 \leq m$
post $0 \leq mp \wedge mp * nn = \overleftarrow{m} * \overleftarrow{n}$

EL
ext rd $m, n \quad : \mathbb{Z}$
  wr $mp, nn \; : \mathbb{Z}$
pre $m < 0$
post $0 \leq mp \wedge mp * nn = \overleftarrow{m} * \overleftarrow{n}$

There is, however, a danger here which results from the generous interpretation of logical expressions given in LPF. The logical expressions in the pre-conditions are now to be used in code; this is only valid if they are defined (δ_l) in the programming language – this is the third hypothesis of the decomposition rule:

$$\boxed{\textbf{if-}I}\;\frac{\{pre \wedge test\}\ TH\ \{post\};\ \{pre \wedge \neg\, test\}\ EL\ \{post\};\ pre \ \Rightarrow\ \delta_l(test)}{\{pre\}\ (\textbf{if}\ test\ \textbf{then}\ TH\ \textbf{else}\ EL)\ \{post\}}$$

It is not difficult to see that:

$\{0 \leq m\}\ (mp := m; nn := n)\ \{post\text{-}TH\}$
$\{m < 0\}\ (mp := -\, m; nn := -\, n)\ \{post\text{-}EL\}$

The actual rules for assignment are given in Section 10.2.

Weakening triples

This 'decomposition' of *COPYPOS* must appear rather strange: even by the standard of this pedagogic example, the step is rather insipid and the actual code would be more clearly foreshadowed if the designer specified the putative sub-components:

TH
ext rd $m, n \quad : \mathbb{Z}$
  wr $mp, nn \; : \mathbb{Z}$
pre true
post $mp = \overleftarrow{m} \wedge nn = \overleftarrow{n}$

EL
ext rd $m, n \quad : \mathbb{Z}$
  wr $mp, nn \; : \mathbb{Z}$

pre true
post $mp = -\overleftarrow{m} \wedge nn = -\overleftarrow{n}$

Although this does not then directly fit the **if**-I rule, it ought to be possible to prove it to be a valid design. (Remember that the claimed decomposition of $COPYPOS$ has to satisfy the former specification for any code which satisfies the specifications of TH and EL only in the context of **if** $0 \leq m$ **then** TH **else** EL.) This situation is handled by a rule which claims that anything which satisfies a specification necessarily satisfies a weaker one:

$$\boxed{weaken}\ \frac{pre_s \ \Rightarrow \ pre;\ \{pre\}\ S\ \{post\};\ post \ \Rightarrow \ post_w}{\{pre_s\}\ S\ \{post_w\}}$$

Notice that a 'weaker' specification is one with a narrower pre-condition or a wider post-condition. In either case, the implication could be just an equivalence thus changing only the other part of the specification. The reader should check that, for both TH and EL, the second specification given above is the stronger and the 'insipid' one can be inferred by *weaken* providing information about the state prior to an operation is available in the post-condition. This inheritance of information can be formalized with the rule:

$$\boxed{pre}\ \frac{\{pre\}\ S\ \{post\}}{\{pre\}\ S\ \{\overleftarrow{pre} \wedge post\}}$$

where $\overleftarrow{pre}$ is like pre except that all of its free variables have been hooked.

Introducing blocks

Clearly, the real work of the initial decomposition of $MULT$ remains to be done in designing $POSMULT$. Its development will introduce a loop and, at this first attempt, a local variable is first defined to control the loop. Thus:

$POSMULT$:
  begin **var** $t := 0;\ r := 0;\ LOOP$ **end**

Where:

$LOOP$
ext rd $mp, nn\ :\ \mathbb{Z}$
  wr $t, r\ \ \ \ :\ \mathbb{Z}$
pre $r = t * nn \wedge t \leq mp$
post $r = \overleftarrow{mp} * \overleftarrow{nn} \wedge t = \overleftarrow{mp}$

To see that this decomposition is correct it is necessary to use *weaken* to get $(t = 0 \wedge r = 0 \wedge 0 \leq mp \ \Rightarrow \ pre\text{-}LOOP)$:

$$\{t = 0 \wedge r = 0 \wedge 0 \leq mp\}\ LOOP\ \{post\text{-}LOOP\}$$

and ;-I to obtain:[3]

$$\{t = 0 \wedge 0 \leq mp\}\ r := 0; LOOP\ \{post\text{-}LOOP\}$$

The introduction of the block is justified by:

$$\boxed{\textbf{block-}I}\ \frac{\{pre \wedge v = e\}\ S\ \{post\}}{\{pre\}\ \textbf{begin}\ \ \textbf{var}\ v := e; S\ \textbf{end}\ \{\exists v \cdot post\}}$$

Which gives:

$$\{pre\text{-}POSMULT\}\ POSMULT\ \{post\text{-}POSMULT\}$$

Decomposing into loops

The actual introduction of the carefully prepared loop construct:

$LOOP$:
  while $t \neq mp$ **do**
    $(t := t + 1;\ r := r + nn)$

is now somewhat of an anti-climax! The intuitive process of convincing oneself that this satisfies *pre-LOOP*/*post-LOOP* should cover the following points:

- the body of the loop keeps the assertion $r = t * nn$ true;
- the negation of the test condition $(t \neq mp)$ conjoined with $r = t * nn$, and the knowledge that mp and nn are read-only, justifies *post-LOOP*;
- the loop terminates: this follows from the fact that $t \leq mp$ initially and, because of the test, remains true after any number of iterations coupled with the fact that increasing t and holding mp constant must eventually result in the test evaluating to false.

The actual rule (**while**-I) which is given below can be seen as a consequence of an unfolding of a while loop into a conditional. Thus if:

WH = **while** *test* **do** S

then (with **skip** as a null statement which changes nothing):

WH = **if** *test* **then** $(S; WH)$ **else skip**

[3]The meaning of the assignment should be obvious; a formal rule is given is the next section.

If *inv* is the condition which remains true at each iteration, *sofar* is a post-condition for S, and *iden* for **skip**, then an overall post-condition for the loop could be proved by the conditional rule as follows:

$$\begin{array}{l} \{inv \wedge \neg\, test\}\ \mathbf{skip}\ \{inv \wedge \neg\, test \wedge iden\} \\ \{inv \wedge test\}\ (S;\, WH)\ \{inv \wedge \neg\, test \wedge sofar\} \\ inv \ \Rightarrow\ \delta_l(test) \vdash \\ \quad \{inv\}\ \mathbf{if}\ test\ \mathbf{then}\ (S;\, WH)\ \mathbf{else\ skip}\ \{inv \wedge \neg\, test \wedge (sofar \vee iden)\} \end{array}$$

The first hypothesis follows from the meaning of **skip**. The second hypothesis is true providing:

$$\{inv \wedge test\}\ S\ \{inv \wedge sofar\}$$

and the relation *sofar* is transitive (i.e. $sofar \mid sofar \ \Rightarrow\ sofar$) and well-founded: this follows by induction on the well-founded ordering *sofar*.

This unfolding idea is provided only to introduce the rule, formally, the **while**-I rule requires that a loop invariant ($inv\colon \Sigma \to \mathbb{B}$) is identified which limits the states which can arise in the computation and that a relation ($sofar\colon \Sigma \times \Sigma \to \mathbb{B}$) is given which holds over one or more iterations of the loop; technically the requirement that ($sofar \mid sofar \ \Rightarrow\ sofar$) is stated by saying that *sofar* must be *transitive*. It is also necessary to ensure termination and this can be done by ensuring that the *sofar* is *well-founded* (cf. the discussion in Section 6.3) over the set defined by *inv*. The rule then is:

$$\boxed{\mathbf{while}\text{-}I}\ \frac{\{inv \wedge test\}\ S\ \{inv \wedge sofar\};\ inv \ \Rightarrow\ \delta_l(test)}{\{inv\}\ \mathbf{while}\ test\ \mathbf{do}\ S\ \mathbf{end}\ \{inv \wedge \neg\, test \wedge (sofar \vee iden)\}}\ sofar \text{ is twf}$$

The decomposition of *LOOP* given above can be seen to be an instance of this rule with:

$$\begin{array}{l} inv \ \Leftrightarrow\ r = t * nn \wedge t \leq mp \\ test \ \Leftrightarrow\ t \neq mp \\ sofar \ \Leftrightarrow\ \overleftarrow{t} < t \\ (sofar \vee iden) \ \Leftrightarrow\ \overleftarrow{t} \leq t \end{array}$$

In this, as in most cases, well-foundedness is easy to exhibit by identifying some expression $(mp - t)$ which decreases at each iteration and is bounded below. The body satisfies:

$$\{r = t{*}nn \wedge t \leq mp \wedge t \neq mp\}\ t := t{+}1; r := r{+}nn\ \{r = t{*}nn \wedge t \leq mp \wedge \overleftarrow{t} < t\}$$

Finally:

$$inv \wedge (sofar \vee iden) \wedge \neg\, test \ \Rightarrow\ r = \overleftarrow{mp} * \overleftarrow{nn} \wedge t = \overleftarrow{mp}$$

Notice the role played by the external clause of $LOOP$: the fact that mp and nn are read-only enables many assertions to be simplified. Without them, $sofar$ would also need to record $mp = \overleftarrow{mp} \land nn = \overleftarrow{nn}$.

An alternative development

One of the advantages claimed for VDM's post-conditions which are truth-valued functions of two states is that they facilitate the specification of operations which modify their input values. This advantage carries over to the inference rules presented above (and, more particularly, to that of Section 10.4). A demonstration of this is obtained by an alternative development of $MULT$ which provides practice with the formal use of the inference rules.

Since m and n can be overwritten (cf. the externals clause of the given specification of $MULT$), an implementation which is adumbrated by:

$MULT$: $MAKEPOS$; $POSMUL$

is possible, with:

$MAKEPOS$
ext wr $m, n \ : \mathbb{Z}$
pre true
post $0 \leq m \land m * n = \overleftarrow{m} * \overleftarrow{n}$

$POSMUL$
ext wr $m, n, r \ : \mathbb{Z}$
pre $0 \leq m$
post $r = \overleftarrow{m} * \overleftarrow{n}$

Notice that, although somewhat similar to $POSMULT$ above, the new operation can change the values of m on n. The actual details of these justifications are left as exercises (see Exercises 10.1.1 and 10.1.2 on page 247).

The development of $POSMULT$ needed a local variable. The need for a temporary variable is avoided by overwriting the value in m. $POSMUL$ is developed directly into the loop:

$POSMUL$:
  $r := 0$;
  while $m \neq 0$ **do**
    $(m := m - 1;\ r := r + n)$

The termination argument for this loop is even simpler than that above. The loop invariant shows that m is never made negative $inv \ \Leftrightarrow \ 0 \leq m$ and the relation is well-founded by showing that the value of m decreases at each iteration ($m < \overleftarrow{m}$). The freedom has been left (and is exploited in Exercise 10.1.4 to give a more efficient algorithm) to change the value of n. Since this is not actually used in this first algorithm, a constraint ($n = \overleftarrow{n}$) is added to the relation. It is, observe, no longer possible to capture the function of the loop by some invariant clause like $r = t * nn$. The essence of the loop must now be captured in the relation by noting that the value of the expression $r + m * n$ is unchanged by executing the body of the loop: what gets added to r gets removed from the product. Thus:

$$sofar \;\Leftrightarrow\; r + m * n = \overleftarrow{r} + \overleftarrow{m} * \overleftarrow{n} \wedge n = \overleftarrow{n} \wedge m < \overleftarrow{m}$$

Notice that *sofar* is transitive. Here again, the detailed justification is deferred to an exercise on page 247. The result in Exercise 10.1.3 is not exactly what is required for *post-POSMUL* but remember that *POSMUL*'s implementation begins with the initialization of r to zero. Intuitively the reader should be able to see that this provides the key result: the required rule is *pre*. Thus:

$$\begin{array}{l}\{0 \leq m \wedge r = 0\}\; LOOP\; \{r = \overleftarrow{r} + \overleftarrow{m} * \overleftarrow{n}\} \vdash \\ \qquad \{0 \leq m \wedge r = 0\}\; LOOP\; \{\overleftarrow{r} = 0 \wedge r = \overleftarrow{r} + \overleftarrow{m} * \overleftarrow{n}\}\end{array}$$

$$\{0 \leq m\}\; r := 0\; \{0 \leq m \wedge r = 0 \wedge n = \overleftarrow{n} \wedge m = \overleftarrow{m}\}$$

$$\begin{array}{l}\{0 \leq m \wedge r = 0\}\; LOOP\; \{r = \overleftarrow{m} * \overleftarrow{n}\} \vdash \\ \qquad \{0 \leq m\}\; r := 0; LOOP\; \{r = \overleftarrow{m} * \overleftarrow{n}\}\end{array}$$

conclude the development.

The use of the inference rules in this section has been rather pedantic so as to make clear how they can be used formally. The next section indicates how annotations of (evolving) designs can rely on the inference rules; this opens a less formal route to documenting justifications which is akin to the level of rigour which has been sought in the natural deduction proofs in this book. Finally, Sections 10.3 and 10.4 show how the inference rules can actually help with choosing specifications of sub-components during the design process.

Exercise 10.1.1 Justify the first step of the alternative development of *MULT* using the ;-I rule.

Exercise 10.1.2 The development of *MAKEPOS* to a conditional is straightforward except that one of the branches is an identity: in the mould set by the discussion of *COPYPOS*, present two developments of *MAKEPOS*.

Exercise 10.1.3 Use **while**-I to prove:

$$\{0 \leq m\} \text{ while } m \neq 0 \text{ do}(m := m - 1; r := r + n) \; \{r = \overleftarrow{r} + \overleftarrow{m} * \overleftarrow{n}\}$$

Exercise 10.1.4 Both the initial and the alternative algorithms for $MULT$ take time proportional to m to compute multiplication. Clearly this is inefficient. The specification of $POSMUL$ has been written so that it is easy to develop an algorithm that takes time proportional to $log_2 m$. For the code:

$r := 0$;
while $m \neq 0$ **do**
  (**while** $is\text{-}even(m)$ **do**
    $(m := m/2;\; n := n * 2)$
  ;
  $r := r + n;\; m := m - 1)$

Prove that the outer loop provides a correct step of development (Hint: compare with the version in the text). Then with *inv* as $1 \leq m$ and *sofar* as $m * n = \overleftarrow{m} * \overleftarrow{n} \wedge m < \overleftarrow{m}$ prove that the inner loop is a valid step.

Exercise 10.1.5 Develop an algorithm for integer division according to the following specification:

IDIV
ext wr $m, n, q \;:\; \mathbb{N}$
pre $n \neq 0$
post $\overleftarrow{n} * q + m = \overleftarrow{m} \wedge m < \overleftarrow{n}$

Use the proof rules of this section to justify each step of decomposition.

Exercise 10.1.6 Just as with the material on logic, it is possible to develop derived rules for programming constructs. Loops are often easier to understand if they are viewed together with their initialization rather than viewing the initialization and the loop as being composed by the rule ;-I. Develop a derived rule for initialized loops (such rules – in a clumsier notation than used in this book – were given in [Jon80].)

10.2 Assertions as annotations

The preceding section introduced and exemplified the decomposition proof rules at a very detailed level. This can be compared with the presentation of the inference rules for logic in Section 1.3; subsequent use of these rules in the ensuing chapters has become more relaxed. The rules are the final recourse while most proofs are at the level of sketches whose detail is provided only in case of doubt.

$MULT$:
wr $m, n, r: \mathbb{Z}$
pre true
  pre true
    if $0 < m$ **then** $(m := -m; n := -n)$
  post $0 \le m \wedge m * n = \overleftarrow{m} * \overleftarrow{n}$
  ;
  pre $0 \le m$
    $r := 0$;
    pre $0 \le m$
      while $m \ne 0$ **do**
      inv $0 \le m$
        $(m := m - 1; r := r + n)$
      sofar $r + m * n = \overleftarrow{r} + \overleftarrow{m} * \overleftarrow{n} \wedge m < \overleftarrow{m}$
    post $r = \overleftarrow{r} + \overleftarrow{m} * \overleftarrow{n}$
  post $r = \overleftarrow{m} * \overleftarrow{n}$
post $r = \overleftarrow{m} * \overleftarrow{n}$

Figure 10.1 Annotated program for multiplication

This section shows that annotating programs with assertions can provide the same sort of sketch. Section 10.3, however, shows that the proof rules for operation decomposition can significantly aid the design process and might be used quite formally for this reason.

Figure 10.1 displays the final program for the second version of $MULT$ as developed in the preceding section. It should be clear how this relates to the detailed inferences of the earlier presentation. Such annotated programs are far easier to review in walkthroughs or inspections than uncommented code. Not only do the assertions record the programmer's intentions; they also provide precisely stylized comments which can be checked against the code by using the inference rules.

The reader might well feel a strong link between annotated programs of this sort and the **from/infer** presentations of natural deduction proofs. It is certainly fair to think of both **from** and *pre* as hypotheses and of **infer** and *post* as goals; what is between them is – in both cases – a form of deduction. Furthermore, the link could be made more obvious if the inference rules used in the steps of Figure 10.1 were shown; that this is not necessary results from there being only one rule per construct. But, in the case of annotated programs, some extra care is required in the handling of variable names. Notice, for example, that the post-

$$
\begin{array}{rrrrr}
 & 1 & 3 & 7 & 4 \\
 & 5 & 9 & 0 & 6 \\
 & 6 & 7 & 1 & 9 \\
 & 4 & 3 & 3 & 7 \\
 & 7 & 7 & 6 & 8 \\
\hline
2 & 6 & 1 & 0 & 4 \\
\hline
 & {\scriptstyle 3} & {\scriptstyle 2} & {\scriptstyle 3} &
\end{array}
$$

Figure 10.2 Turing's addition example

condition $r = \overleftarrow{m} * \overleftarrow{n}$ of the code developed from the specification of *POSMUL* refers to the values of m and n before $r := 0$ is executed, while the same formula as the overall post-condition refers to the values of the variables when execution of *MULT* begins. This is emphasized by the indentation. Furthermore, the **inv**/**sofar** assertions – written to annotate the **while** construct – play a threefold part in **while**-I: discharging the hypothesis ($\{inv \wedge test\}\ S\ \{inv \wedge sofar\}$), the step from $inv \wedge \neg\, test \wedge (sofar \vee iden)$ to $r = \overleftarrow{r} + \overleftarrow{m} * \overleftarrow{n}$, and the check that the pre-condition of the whole loop justifies inv.

As experience with this style of annotation increases, the amount which actually needs to be written diminishes. In the extreme, the absolute minimum is to record the pre- and post-conditions of each procedure. These provide the essential documentation of its specification. But, apart from the oft-repeated argument that – because of the formal framework – more detail can be provided if it is needed to convince readers, the author should be prepared to record enough to help future readers (which might well include the author after many months of separation from the text). The British mathematician Alan Turing made this point very graphically[4] with a comparison to the simple addition in Figure 10.2: if the carry digits are recorded, the task of checking can be separated into four disjoint tasks whereas, without the carries, the whole sum must be checked.

Figure 10.1 represents the final code but the annotation idea can be used to record intermediate stages of development. On such a simple example, this is less convincing but Figure 10.3 gives an indication of what can be done. The ideal is to have a computer-based support system which could work at a level of design like that in Figure 10.3 and facilitate (generate proof obligations, etc.) proof of that level of design; it could then separate the specifications of the sub-operations showing only their specifications to the programmer developing the

[4]This was in a paper published, incredibly, in 1949 – see [MJ84] for a discussion of his proof method.

$MULT$:
wr $m, n, r: \mathbb{Z}$
pre true
  $MAKEPOS$
  wr $m, n: \mathbb{Z}$
  pre true
  post $0 \leq m \wedge m * n = \overleftarrow{m} * \overleftarrow{n}$
  ;
  $POSMUL$
  pre $0 < m$
  post $r = \overleftarrow{m} * \overleftarrow{n}$
post $r = \overleftarrow{m} * \overleftarrow{n}$

Figure 10.3 Annotated design

respective code. Furthermore, when all is complete, the system could gather the code for compilation (and present any level of annotation selected by a subsequent reader). References to such systems can be found in [Lin88, JL88].

Assignment statements

At the level of detail suggested here, it is not normally necessary to reason very formally about the basic building blocks of procedural programming. The obvious rule for assignment statements is:

$$\boxed{:=\text{-}I}\;\frac{}{\{\mathbf{true}\}\ x := e\ \{x = \overleftarrow{e}\}}$$

The so-called 'frame problem' has been referred to above. To state that variables other than that on the left-hand-side of the assignment do *not* change either requires some extended notation to describe state identity over a set of variables, or – as here – can be defined:

$$\boxed{:=\text{-}pres}\;\frac{}{\{E\}\ x := e\ \{E\}}\ x \text{ does not occur free in } E$$

Notice that this relies on the assumption that the programming language does not allow different references to refer to the same variable. This property has been ensured by stating that all parameters are assumed to be passed by value.

Exercise 10.2.1 Present the design of $POSMUL$ from Exercise 10.1.4 on page 248 as an annotated program.

Exercise 10.2.2 Present the design of the program from Exercise 10.1.5 as an annotated program.

10.3 Decomposition in design

The preceding section introduces the decomposition proof rules by showing their use on given programs. This section shows how the proof obligations can be used to stimulate program design steps. An obvious example of the way in which a proof rule can help a designer's thinking about decomposition is given by the rule for sequence – the assertion pre_2 fixes an interface between the two sub-operations.

It is, however, important that the reader is not led to expect too much from this idea. Design requires intuition and cannot, in general, be automated. What is offered is a framework into which the designer's commitments can be placed. If done with care, the verification then represents little extra burden. Even so, false steps of design cannot be avoided in the sense that even a verified decision can lead to a blind alley (e.g. a decomposition which has unacceptable performance implications). If this happens, there is no choice but to reconsider the design decision which led to the problem. Once again, what is being offered is a framework into which a final design explanation can be fitted. This section aims only to show that the need for verification can also help the design process.

Searching

The outline annotations of the preceding section can be used, together with the associated proof rules, as an aid to the design process. As a first example, consider the task of searching for some value e in a vector v; if the value is found a flag is set and i is to contain a (not necessarily unique) index to v such that $v(i) = e$; if the value is absent, the flag *found* is to be set to **false**. The specification can be written:

$$
\begin{array}{l}
\mathit{SEARCH}\\
\textbf{ext rd } v \quad : \mathit{El}^*\\
\qquad \textbf{rd } e \quad : \mathit{El}\\
\qquad \textbf{wr } i \quad : \mathbb{N}\\
\qquad \textbf{wr } \mathit{found} : \mathbb{B}\\
\textbf{pre true}\\
\textbf{post } \mathit{checked}(v, e, i, \mathit{found})
\end{array}
$$

where:

$$checked : (El^*) \times El \times \mathbb{N} \times \mathbb{B} \to \mathbb{B}$$
$$checked(v, e, i, f) \quad \triangleq \quad f \wedge v(i) = e \vee \neg f \wedge e \notin \textbf{elems}\ v$$

An obvious approach to the design is to iterate over the indices of v with the variable i and exit if and when a suitable index is found. This suggests a loop invariant which as well as constraining i, asserts that *checked* is true for the initial $(v(1, \ldots, i))$ part of v:

$$\textbf{inv}\ \ i \leq \textbf{len}\ v \wedge checked(v(1, \ldots, i), e, i, found)$$

Since the major variables are read-only, the invariant expresses most of what is going on in the loop. The loop relation need only provide evidence of termination: well-foundedness of $\textbf{len}\ v - i$ is established with:

$$\textbf{sofar}\ \overleftarrow{i} < i$$

which is obviously transitive. The loop test needs to be such that the conjunction of its negation with the loop invariant yields *post-SEARCH*:

$$\neg\, test \wedge i \leq \textbf{len}\ v \wedge checked(v(1, \ldots, i), e, i, found) \vdash checked(v, e, i, found)$$

With *test* as $\neg found \wedge i < \textbf{len}\ v$ this follows because $\neg\, test \wedge i \leq \textbf{len}\ v$ gives:

$$found \vee i = \textbf{len}\ v$$

which, when distributed over the disjunction in $checked(v(1, \ldots, i), e, i, found)$, gives:

$$found \wedge v(i) = e \vee \neg found \wedge e \notin \textbf{elems}\ v$$

The last step needed in the design is to establish the invariant: this is simply done by setting i to 0 and *found* to **false**. Thus the summary of the design step can be written as in Figure 10.4. Code which achieves the preservation of the invariant and which also respects the loop relation is:

$$i := i + 1;\ \ \textbf{if}\ v(i) = e\ \ \textbf{then}\ found := \textbf{true};$$

Binary search

The *SEARCH* problem as specified has poor performance for large vectors but a small change makes a much faster algorithm possible. If frequent searches of this sort were necessary it would be worth trying to ensure that v is kept in order:

$$Ordv = El^*$$
$$inv(v) \triangleq is\text{-}ord(v)$$

pre true
  pre true
    $found := \mathbf{false}$;
    $i := 0$;
  post $i \leq \mathbf{len}\ v \wedge checked([\,], e, i, found)$
  while $\neg found \wedge i < \mathbf{len}\ v$ **do**
  inv $i \leq \mathbf{len}\ v \wedge checked(v(1, \ldots, i), e, i, found)$
    BODY
  sofar $\overleftarrow{i} < i$
post $checked(v, e, i, found)$

Figure 10.4 Summary of first design step

Then the specification becomes:

BSEARCH
ext rd v : $Ordv$
  rd e : El
  wr ind : $\mathbb{N}$
  wr $found$: $\mathbb{B}$
pre true
post $checked(v, e, ind, found)$

This could – if efficiency were ignored – be realized by the development above, but could also be implemented by a *binary search*. The first steps of this design again show the advantage of thinking about loop construction via **inv/sofar** pairs. The basic concept is to move two indices m and n so that they delimit the yet-to-be-checked portion of v. The loop invariant is, in spirit, very like that for *SEARCH*; it is only longer because of the need to constrain both indices and to define the checked area:

inv $1 \leq m \wedge n \leq \mathbf{len}\ v \wedge checked(v(1, \ldots, m-1) \frown v(n+1, \ldots \mathbf{len}\ v), e, ind, found)$

Here again, the (transitive) loop relation only has to ensure termination:

sofar $(n - m) < (\overleftarrow{n} - \overleftarrow{m}) \vee found$

The loop test needed to ensure *post-BSEARCH* is – by very similar reasoning to that used above:

$\neg found \wedge m \leq n$

$$
\begin{array}{l}
BODY \\
\textbf{pre } 1 \le m \le n \le \textbf{len } v \wedge e \notin \textbf{elems } v(1,\ldots,m-1) \wedge e \notin \textbf{elems } v(n+1,\ldots,\textbf{len } v) \\
\quad PICKIND \\
\quad \textbf{ext rd } m, n : \mathbb{N} \\
\qquad \textbf{wr } ind : \mathbb{N} \\
\quad \textbf{pre } m \le n \\
\quad \textbf{post } m \le ind \le n \\
\quad ; \\
\quad \textbf{if } v(ind) = e \textbf{ then } found := \textbf{true} \\
\quad \textbf{else } \textbf{ if } v(ind) < e \textbf{ then } m := ind + 1 \\
\quad \textbf{else } n := ind - 1 \\
\textbf{post } 1 \le m \wedge n \le \textbf{len } v \wedge \\
\quad checked(v(1,\ldots,m-1) \curvearrowright v(n+1,\ldots,\textbf{len } v), e, ind, found) \wedge \\
\quad (n - m < \overleftarrow{n} - \overleftarrow{m} \vee found)
\end{array}
$$

Figure 10.5 *BODY* for *BSEARCH*

So, not surprisingly, the first design step is very like that in Figure 10.4 (the initialization sets m to 1 and n to $\textbf{len}\, v$). The interest is in *BODY*. Figure 10.5 shows the next stage of design. The overall pre- and post-conditions are formed from the loop invariant and relation in an obvious way. The next level of design is also shown. The process of picking an index (roughly midway between m and n) is left as an under-determined specification. The task of adjusting the search area (after possibly setting *found*) is written as nested conditional statements. If this design were the subject of an inspection, the author might be called on to justify that – for example – the big steps to m retained $checked(1,\ldots,m-1)$ in *post-BODY*: it is precisely here that the invariant on v would have to be mentioned in addition to $v(ind) < e$.

Sorting

The obvious territory to explore after searching – with the development method at hand – is that vast area of knowledge about sorting algorithms. Partly because this is covered so thoroughly elsewhere (see [Dro87] for a recent paper with useful references), but also because most algorithms fail to illustrate what is important about VDM's post-conditions of two states, this foray is limited. The task of sorting is discussed in Section 6.3; the main points are collected as:

$$
\begin{array}{l}
SORT \\
\textbf{ext wr } v : Rec^*
\end{array}
$$

pre **len** $v \geq 1$
post $is\text{-}ord(v) \wedge is\text{-}perm(v, \overleftarrow{v})$

It is exactly here that, were a longer discourse planned, a theory of ordered sequences and permutations might be undertaken. This work is left to a (starred) exercise but, rather than expand out the definitions, properties of *is-ord* and *is-perm* are identified below as needed.

The simplest approach to internal sorting appears to be to have an increasing group of ordered elements at one end of the vector. Using an index i to mark the end of this area suggests a loop invariant:

inv $1 \leq i \leq$ **len** $v \wedge is\text{-}ord(v(1, \ldots, i))$

Unlike the searching task above, it is of the essence of internal sorting that the major data structure changes. The loop relation then is used both to ensure that a permutation of the original values is retained and to establish termination:

sofar $is\text{-}perm(v, \overleftarrow{v}) \wedge \overleftarrow{i} < i$

This **sofar** is transitive but – since the fact is less obvious – the reader should check the fact. The invariant is easily established by setting i to 1 (notice the sequence is non-empty) since it is a property of *is-ord* that it is true for any unit sequence (i.e. $v(1, \ldots, 1)$). A loop test condition which, combined with both the invariant and the (reflexive closure of the) relation, gives *post-SORT* is $i \neq$ **len** v (or $i <$ **len** v). The comments thus far give the outer structure shown in Figure 10.6 where n has been written as a constant for **len** v. The body of the loop clearly has to preserve the loop invariant and respect the loop relation. Given the test, it is safe to increase i by 1 and still respect the first clause of the invariant. The second conjunct is clearly more interesting. The obvious element to absorb into $\overleftarrow{v}(1, \ldots, \overleftarrow{i})$ (to form $v(1, \ldots \overleftarrow{i} + 1)$ or $v(1, \ldots, i)$) is that located at $\overleftarrow{v}(\overleftarrow{i} + 1)$ but the invariant is only satisfied if it is correctly placed. It seems reasonable to postpone the issue of how this is achieved to the next step of development. The post-condition of $SBODY1$ therefore defines the movement of $\overleftarrow{v}(i)$ to some position (j) in v and fixes the constancy (or limited movement) of the rest of v:

$SBODY1$
ext wr v : Rec^*
  rd i : $\mathbb{N}$
pre $is\text{-}ord(v(1, \ldots, i)) \wedge 1 \leq i < n$

$SORT$
ext wr $v: Rec^*$
pre len $v \geq 1$
  var $i: \mathbb{N}$
  $i := 1;$
  while $i \neq n$ **do**
  inv $1 \leq i \leq n \wedge is\text{-}ord(v(1, \ldots, i))$
    $SBODY1$
  sofar $is\text{-}perm(v, \overleftarrow{v}) \wedge \overleftarrow{i} < i$
post $is\text{-}ord(v) \wedge is\text{-}perm(v, \overleftarrow{v})$

Figure 10.6 Development of insertion sort

$$\textbf{post } i = \overleftarrow{i} + 1 \wedge is\text{-}ord(v(1, \ldots, i)) \wedge \\ \exists j \in \{1, \ldots, i\} \cdot \\ del(\overleftarrow{v}, i) = del(v, j) \wedge \overleftarrow{v}(i) = v(j)$$

Two significant points can be drawn from this material. Firstly, notice how the *is-ord* and *is-perm* naturally slotted into the loop invariant and relation respectively. Secondly, the use of a specification for $SBODY1$ has made it easy to fix one design decision (which element to absorb) and postpone another – the algorithm by which it is to be placed in its correct position. (In fact, one should really say 'an acceptable position' since – in the presence of duplicates – the algorithm is under-determined.) This algorithm could now be developed into a 'straight insertion' or, if there are more elements, the binary search idea presented above can be used to achieve a 'binary insertion' with slightly better performance.

As mentioned above, it is not the intention in this chapter to reproduce the wealth of material published on sorting as illustrations of the use of the decomposition rules in design. However, in order to prompt interested readers in this direction, one further class of sorting strategies can be mentioned. Algorithms which find the correct final placing of an element need an additional clause in the loop invariant which records the fact that the sequence (v) is *split* around a point:

$$split : Rec^* \times \mathbb{N} \to \mathbb{B}$$
$$split(v, i) \quad \triangleq \quad \forall j \in \{1, \ldots, i\} \cdot \forall k \in \{i+1, \ldots, n\} \cdot v(j) \leq v(k)$$

Exercise 10.3.1 Complete the development of $SBODY$ to, at least, a simple insertion routine. Continue the development of an algorithm with the property that it places elements in their final position.

Exercise 10.3.2 (*) Pursue the development of some non-trivial sorting algorithms using the method described in this section. In particular, use **inv/sofar** pairs in the design of loops and attempt to make only one design decision per step and, if possible, develop different algorithms from the same intermediate step to show their family likeness.

Integer division

Part of the interest in the developments from the specification of $SORT$ is the fact that the programs have to overwrite v. A development from the specification given in Exercise 10.1.5 on page 248 wallows in this sort of overwrite and offers a challenge for clear exposition. The intuition behind the algorithm is the way in which mechanical calculators performed division. For the specification on page 248, n is shifted (i.e. multiplied by 10^i) until it is larger than m; after shifting one place back, subtraction is performed until the next step would cause the evolving remainder in m to go negative; this is repeated in each of the remaining $i-1$ positions.

So, in the first step of development, the interface between left-shifting (LS) and right-shifting (RS) is mediated by:

$$10^i \textbf{ divides } n \wedge m < n$$

The variable q also has to be initialized to 0 and this task can also be given to LS. Generalizing *post-RS* in a way which should by now be familiar, the reader should easily be able to verify the first step of development as shown in Figure 10.7. The exact form of *pre-RS* is not contained in *post-LS* but the first conjunct of the former is a consequence of the latter. The remaining information in $n = \overleftarrow{n} * 10^i$ is used in *post-LS* | *post-RS* to show (using ;-I) that the value of n reverts over the composition of the two operations to its value before their execution. Notice that $m = \overleftarrow{m}$ over LS because it only has read access.

The development of LS is straightforward (cf. Figure 10.8) but it is interesting to note that the design of the loop is controlled entirely by the loop relation with the invariant offering no constraint. (Notice that the argument about $q = 0$ is not fully formalized.)

Surprisingly, the first step of development of RS is simple (see Figure 10.9): the loop relation is derived by generalizing the first conjunct of *post-RS*, conjoining this with an unchanged second conjunct of *post-RS* and finally a term to ensure termination. The loop invariant is exactly *pre-RS*. It is not difficult to see how to describe one step of right shifting $i := i - 1$ and the attendant changes ($n := n/10; q := q * 10$) to re-establish the loop relation. The key problem is

IDIV
ext wr $m, n, q, i \colon \mathbb{N}$
pre $n \neq 0$
  LS
  ext rd $m \colon \mathbb{N}$
    wr $n, q, i \colon \mathbb{N}$
  pre $n \neq 0$
  post $n = \overleftarrow{n} * 10^i \wedge m < n \wedge q = 0$
  ;
  RS
  ext wr $m, n, q, i \colon \mathbb{N}$
  pre 10^i **divides** $n \wedge m < n$
  post $n = \overleftarrow{n} / 10^{\overleftarrow{i}} \wedge n * q + m = \overleftarrow{n} * \overleftarrow{q} + \overleftarrow{m} \wedge m < n$
post $\overleftarrow{n} * q + m = \overleftarrow{m} \wedge m < \overleftarrow{n}$

Figure 10.7 First step of integer division

LS
ext rd $m \colon \mathbb{N}$
  wr $n, q, i \colon \mathbb{N}$
pre $n \neq 0$
  $q := 0; i := 0;$
  while $n \leq m$ **do**
  inv true
    $(n := n * 10; i := i + 1)$
  sofar $n * 10^{\overleftarrow{i}} = \overleftarrow{n} * 10^i \wedge \overleftarrow{n} < n$
post $n = \overleftarrow{n} * 10^i \wedge m < n \wedge q = 0$

Figure 10.8 *LS* development for integer division

RS
ext wr $m, n, q, i : \mathbb{N}$
pre 10^i **divides** $n \wedge m < n$
  while $i \neq 0$ **do**
  inv 10^i **divides** $n \wedge m < n$
    $i := i - 1; n := n/10; q := q * 10$
    ;
    INNER
    ext rd $n : \mathbb{N}$
      wr $m, q : \mathbb{N}$
    pre $n \neq 0$
    post $n * q + m = \overleftarrow{n} * \overleftarrow{q} + \overleftarrow{m} \wedge m < n$
  sofar $n/10^i = \overleftarrow{n}/10^{\overleftarrow{i}} \wedge n * q + m = \overleftarrow{n} * \overleftarrow{q} + \overleftarrow{m} \wedge i < \overleftarrow{i}$
post $n + \overleftarrow{n}/10^{\overleftarrow{i}} \wedge n * q + m = \overleftarrow{n} * \overleftarrow{q} + \overleftarrow{m} \wedge m < n$

Figure 10.9 *RS* development for integer division

INNER
ext rd $n : \mathbb{N}$
  wr $m, q : \mathbb{N}$
pre $n \neq 0$
  while $n \leq m$ **do**
  inv true
    $m := m - n; q := q + 1$
  sofar $n * q + m = \overleftarrow{n} * \overleftarrow{q} + \overleftarrow{m} \wedge m < \overleftarrow{m}$
post $n * q + m = \overleftarrow{n} * \overleftarrow{q} + \overleftarrow{m} \wedge m < n$

Figure 10.10 Design of *INNER* for *RS*

how to re-establish the second clause of the loop invariant. The task of so doing is pushed on to the yet-to-be-developed *INNER*.

Here again, the design step is not difficult. The loop relation comes naturally from *post-INNER* and the loop invariant is **true** (see Figure 10.10). Gathering the final code from Figures 10.7–10.10 yields a short program. It is, however, one which a reader is unlikely to make any sense of without the aid of assertions. It is particularly interesting how the tendency to overwrite variables appears to force more reliance on loop relations at the expense of loop invariants.

Exercise 10.3.3 Provide annotated code for both versions of the factorial program in Section 3.4 (one in the body of the section, the other in Exercise 3.4.4 on page 86).

10.4 An alternative loop rule

Strictly the **while**-I rule is powerful enough to prove any result needed about while loops.[5] There are, however, pragmatic grounds for presenting the alternative rule given below. Recall that one of the objectives of the operation decomposition rules given in this book is to cope with post-conditions of two states. This they do; they have even been shown above to deal naturally with programs which overwrite the initial values of some variables. But there is something unnatural in the way that *sofar* has to be defined in some examples. Consider again Figure 10.1 on page 249 – *sofar* contains $r + m * n = \overleftarrow{r} + \overleftarrow{m} * \overleftarrow{n}$. The fact that the essential operation – multiplication in this case – appears on both sides of the equality is disturbing. It is needed because **while**-I essentially relies on relating the state after n loop iterations back to the initial state. If, instead, the relational predicate relates states after some arbitrary number of loop iterations to the final state of the loop, it is possible to write:

$$r = \overleftarrow{r} + \overleftarrow{m} * \overleftarrow{n}$$

This neatly expresses the intended function of the whole loop if one considers the situation after zero iterations.

It might appear to be excessively pernickety to introduce another decomposition rule for while statements just to avoid a repeated multiplication sign but this is a difficulty which can become more serious with larger examples. As is shown below, the alternative rule also functions very well when used in the design process.

If the analysis via conditional statements which was done in Section 10.1 is followed, the reader should obtain a good grasp of the alternative rule. Here again, assume:

$$WH = \textbf{while } test \textbf{ do } S$$

In order to show that:

$$\{inv\}\ WH\ \{toend\}$$

is true, the analysis of the conditional unfolding of WH gives:

[5] Peter Aczel (Manchester University) has provided a completeness proof in an unpublished note.

$$
\begin{array}{l}
\textbf{pre}\ 0 \leq m \\
\quad \textbf{while}\ m \neq 0\ \textbf{do} \\
\quad \textbf{inv}\ 0 \leq m \\
\qquad (m := m - 1;\ r := r + n) \\
\quad \textbf{toend}\ r = \overleftarrow{r} + \overleftarrow{m} * \overleftarrow{n} \\
\textbf{post}\ r = \overleftarrow{r} + \overleftarrow{m} * \overleftarrow{n}
\end{array}
$$

Figure 10.11 Alternative rule for *POSMUL* annotation

$$
\begin{array}{l}
\{inv \wedge \neg\, test\}\ \textbf{skip}\ \{toend\} \\
\{inv \wedge test\}\ (S;\ WH)\ \{toend\} \vdash \\
\quad \{inv\}\ \textbf{if}\ test\ \textbf{then}\ (S;\ WH)\ \textbf{else skip}\ \{toend\}
\end{array}
$$

The first of these requirements is adopted as a hypothesis of **while**-$I2$. The second requirement must again rely on induction. If S conserves inv, it is sufficient to prove that $\{inv \wedge test\}\ (S;\ WH)\ \{toend\}$ holds under the assumption that $\{inv\}\ WH\ \{toend\}$ is true. The termination of the loop is assured providing S reduces some value which is bounded in inv. The final rule is then:

$$
\boxed{\textbf{while-}I2}\ \frac{\begin{array}{c}\{inv \wedge \neg\, test\}\ \textbf{skip}\ \{toend\}; \\ \{inv \wedge test\}\ S\ \{inv\}; \\ \{inv\}\ WH\ \{toend\} \vdash \{inv \wedge test\}\ (S;\ WH)\ \{toend\}\end{array}}{\{inv\}\ \textbf{while}\ test\ \textbf{do}\ S\ \{toend\}}
$$

The hypotheses of this rule are more complicated than for **while**-I, but it is the simplicity of the conclusion which is the key to its usefulness in design. It naturally prompts the designer to think of the loop for say *POSMUL* as computing $r := r + m * n$ and then to convert this to a predicate of two states and to compute the other predicates needed.

The annotated code for *POSMUL* is shown in Figure 10.11. As before, it is important to see that each of the steps in **while**-$I2$ is established. Thus:

$$
\begin{array}{l}
\{0 \leq m \wedge \neg\,(m \neq 0)\}\ \textbf{skip}\ \{r = \overleftarrow{r} + \overleftarrow{m} * \overleftarrow{n}\} \\
\{0 \leq m \wedge m \neq 0\}\ (m := m - 1; r := r + n)\ \{0 \leq m\}
\end{array}
$$

and:

$$
\begin{array}{l}
\{0 \leq m\}\ WH\ \{r = \overleftarrow{r} + \overleftarrow{m} * \overleftarrow{n}\} \vdash \\
\qquad \{0 \leq m \wedge m \neq 0\}\ (m := m - 1; r := r + n;\ WH)\ \{r = \overleftarrow{r} + \overleftarrow{m} * \overleftarrow{n}\}
\end{array}
$$

must all be true as must the fact that the meaning of the body of the loop is well-founded over the states defined by inv.

INNER
pre $n \neq 0$
  $q := 0$;
  pre $n \neq 0$
    while $n \leq m$ **do**
    inv **true**
      $m := m - n; q := q + 1$
    toend $q = \overleftarrow{q} + \overleftarrow{m} \div \overleftarrow{n} \wedge m = \overleftarrow{m}$ **mod** $\overleftarrow{n}$
  post $q = \overleftarrow{q} + \overleftarrow{m} \div \overleftarrow{n} \wedge m = \overleftarrow{m}$ **mod** $\overleftarrow{n}$
post $q = \overleftarrow{m} \div \overleftarrow{n} \wedge m = \overleftarrow{m}$ **mod** $\overleftarrow{n}$

Figure 10.12 Simple algorithm for integer division

The integer division problem introduced in Exercise 10.1.5 on page 248 and pursued in Sections 10.2 and 10.3 provides another illustration of the use of **while**-$I2$. The annotated inner loop of the program is shown in Figure 10.12. Notice it is now natural to state the specification and development in terms of $\div$ and **mod** .

Binary trees

As a further example of a decomposition proof, the binary tree problem is picked up from Section 8.3. An exercise offers the challenge of developing a loop-based solution to the problem. This subsection explores how a recursive program can be developed. In particular, the topic of parameter passing 'by reference' ('by variable', 'by location') is considered. Recall (cf. Section 3.4) that sharing has been avoided so far by insisting that the parameters to operations themselves are passed by value. Particularly in the case of recursion, this mode is sometimes unacceptable for performance reasons. The development of this example shows that the effect of 'by location' parameters can be simulated with external variables. No formal rules are given here but the overall argument is presented so that the recursive program is easy to create.

The development of *Setrep* in Section 5.2 employs recursive functions (e.g. *isin*) which can be used in the specifications of operations. The disadvantage of this approach is that it does not lend itself to the form of recursion which is intended in the recursive program. In particular, the code to be presented here uses location parameters. Rather than mirror the development of *Setrep*, quotation of post-conditions is used in the development of *Mrep*:

$Mrep = [Mnode]$

$$\begin{aligned} Mnode :: \; & lt \; : Mrep \\ & mk : Key \\ & md : Data \\ & rt \; : Mrep \end{aligned}$$

$$\textbf{inv } (mk\text{-}Mnode(lt, mk, md, rt)) \triangleq$$
$$\quad (\forall lk \in collkeys(lt) \cdot lk < mk) \wedge (\forall rk \in collkeys(rt) \cdot mk < rk)$$

The (read-only) search operation is specified:

$$\begin{aligned} & FINDB \; (k{:}\, Key) \; d{:}\, Data \\ & \textbf{ext rd } t \; : \; Mrep \\ & \textbf{pre } k \in collkeys(t) \\ & \textbf{post let } mk\text{-}Mnode(lt, mk, md, rt) = t \textbf{ in} \\ & \qquad k = mk \wedge d = md \vee \\ & \qquad k < mk \wedge post\text{-}FINDB(k, lt, d) \vee \\ & \qquad mk < k \wedge post\text{-}FINDB(k, rt, d) \end{aligned}$$

The proof that *FINDB* satisfies the specification *FIND* uses – in addition to Lemmas 8.16 and 8.17 – the following induction rule.

Axiom 10.1

$$\boxed{\text{A10.1}} \; \frac{\begin{array}{l} p(\textbf{nil}); \\ mk \in Key, \; md \in Data, \; lt, rt \in Mrep, \; p(lt), \\ inv\text{-}Mnode(mk\text{-}Node(lt, mk, md, rt)), \; p(rt) \vdash \\ \qquad\qquad p(mk\text{-}Mnode(lt, mk, md, rt)) \end{array}}{t \in Mrep \; \vdash \; p(t)}$$

The insertion operation on *Mrep* is specified:

$$\begin{aligned} & INSERTB \; (k{:}\, Key, d{:}\, Data) \\ & \textbf{ext wr } t \; : \; Mrep \\ & \textbf{pre } k \notin collkeys(t) \\ & \textbf{post } \overleftharpoon{t} = \textbf{nil} \wedge t = mk\text{-}Mnode(\textbf{nil}, k, d, \textbf{nil}) \vee \\ & \qquad \overleftharpoon{t} \in Mnode \wedge \\ & \qquad \textbf{let } mk\text{-}Mnode(\overleftharpoon{lt}, mk, md, \overleftharpoon{rt}) = \overleftharpoon{t} \textbf{ in} \\ & \qquad k < mk \wedge \\ & \qquad\qquad \exists lt \in Mrep \cdot \\ & \qquad\qquad\; post\text{-}INSERTB(k, d, \overleftharpoon{lt}, lt) \wedge t = mk\text{-}Mnode(lt, mk, md, \overleftharpoon{rt}) \vee \\ & \qquad mk < k \wedge \\ & \qquad\qquad \exists rt \in Mrep \cdot \\ & \qquad\qquad\; post\text{-}INSERTB(k, d, \overleftharpoon{rt}, rt) \wedge t = mk\text{-}Mnode(\overleftharpoon{lt}, mk, md, rt) \end{aligned}$$

This completes the development of operations on *Mrep* which can now be taken as a specification of the next step of design. The tree-like objects of *Mrep* cannot be directly constructed in a language like Pascal. Instead, each node must be created on the heap; nested trees must be represented by pointers. Pascal-like objects can be defined by:

$$Root = [Ptr]$$

$$Heap = Ptr \xrightarrow{m} Mnoder$$

$$\begin{aligned} Mnoder :: \; & lp & : & \; [Ptr] \\ & mk & : & \; Key \\ & md & : & \; Data \\ & rp & : & \; [Ptr] \end{aligned}$$

It is clear that the *Heap* relation should be well-founded (cf. Section 6.3) and that all *Ptrs* contained in *Mnoders* should be contained in the domain of the *Heap*. The retrieve function can then be defined:

$$\begin{array}{l} retr\text{-}Mrep : Root \times Heap \to Mrep \\ retr\text{-}Mrep(r, h) \quad \triangle \\ \quad \textbf{if } r = \textbf{nil} \\ \quad \textbf{then nil} \\ \quad \textbf{else let } mk\text{-}Mnoder(lp, mk, md, rp) = h(r) \textbf{ in} \\ \qquad mk\text{-}Mnode(retr\text{-}Mrep(lp, h), mk, md, retr\text{-}Mrep(rp, h)) \end{array}$$

The function:

$$collkeysh \colon Root \times Heap \to Key\text{-}\mathbf{set}$$

is an obvious derivative of *collkeys*.

The find operation on *Heap* is specified:

$$\begin{array}{l} FINDH \; (k \colon Key) \; d \colon Data \\ \textbf{ext rd } p \; : \; Ptr, \\ \quad \textbf{rd } h \; : \; Heap \\ \textbf{pre } k \in collkeysh(p, h) \\ \textbf{post let } mk\text{-}Mnoder(lp, mk, md, rp) = h(p) \textbf{ in} \\ \qquad k = mk \wedge d = md \vee \\ \qquad k < mk \wedge post\text{-}FINDH(k, lp, h, d) \vee \\ \qquad mk < k \wedge post\text{-}FINDH(k, rp, h, d) \end{array}$$

This is fairly simple because the pointer can be passed by value and is thus a read-only external variable. In the insert operation, the pointer can be changed in the case that a new node is created. Thus, in addition to the obvious write access on the heap itself, the pointer is shown as an external variable to which the operation has read and write access. In the actual code, this is achieved by using a parameter passed 'by location'.

$$
\begin{array}{l}
\mathit{INSERTRH}\ (k{:}\,\mathit{Key}, d{:}\,\mathit{Data})\\
\textbf{ext wr}\ h\ :\ \mathit{Heap},\\
\quad\ \ \textbf{wr}\ p\ :\ \mathit{Ptr}\\
\textbf{pre}\ k \notin \mathit{collkeysh}(p, h)\\
\textbf{post}\ \overleftarrow{p} = \textbf{nil} \wedge p \notin \textbf{dom}\ \overleftarrow{h} \wedge\\
\qquad h = \overleftarrow{h} \cup \{p \mapsto \mathit{mk\text{-}Mnoder}(\textbf{nil}, k, d, \textbf{nil})\} \vee\\
\qquad \overleftarrow{p} \neq \textbf{nil} \wedge\\
\qquad \textbf{let}\ \mathit{mk\text{-}Mnoder}(\overleftarrow{lp}, mk, md, \overleftarrow{rp}) = \overleftarrow{h}(\overleftarrow{p})\ \textbf{in}\\
\qquad k < mk \wedge\\
\qquad\quad (\exists hi \in \mathit{Heap}, lp \in \mathit{Ptr} \cdot\\
\qquad\qquad \mathit{post\text{-}INSERTRH}(k, d, \overleftarrow{h}, \overleftarrow{lp}, hi, lpi) \wedge\\
\qquad\qquad h = hi \dagger \{\overleftarrow{p} \mapsto \mu(h(\overleftarrow{p}), lp \mapsto lpi)\} \wedge p = \overleftarrow{p}) \vee\\
\qquad mk < k \wedge\\
\qquad\quad (\exists hi \in \mathit{Heap}, rp \in \mathit{Ptr} \cdot\\
\qquad\qquad \mathit{post\text{-}INSERTRH}(k, d, \overleftarrow{h}, \overleftarrow{rp}, hi, rpi) \wedge\\
\qquad\qquad h = hi \dagger \{\overleftarrow{p} \mapsto \mu(h(\overleftarrow{p}), rp \mapsto rpi)\} \wedge p = \overleftarrow{p})
\end{array}
$$

The Pascal equivalent of the data objects there is:

$$
\begin{array}{l}
\textbf{type}\ \ \mathit{Ptr} = \uparrow \mathit{Binoderep}\\
\mathit{Binnoderep} =\\
\quad\ \ \textbf{record}\\
\qquad\quad lp{:}\,\mathit{Ptr}\\
\qquad\quad mk{:}\,\mathit{Key}\\
\qquad\quad md{:}\,\mathit{Data}\\
\qquad\quad rp{:}\,\mathit{Ptr}\\
\quad\ \ \textbf{end}
\end{array}
$$

The *FINDBH* function can be coded (with auxiliary functions *findbhn* and *depth* for the assertions) as shown in Figure 10.13.

Exercise 10.4.1 Consider the two programs given for factorial in Exercise 10.3.3 on page 261. One of them can be proved more conveniently with **while**-$I2$ than with **while**-I as used: write this as an annotated program. What happens if you try to reformulate the other one with **while**-$I2$.

Exercise 10.4.2 (*) Continue (down to code) the development of B-Trees started in Exercise 8.3.1 on page 203.

Exercise 10.4.3 (*) Write one or more versions of programs to sum the elements in a vector. Experiment with **while**-I and **while**-$I2$

Exercise 10.4.4 (*) Develop a loop version of the binary tree example of the last subsection.

function $FINDBH(k: Key)\, d: Data$
ext rd $rt: Ptr$, **rd** $h: Heap$
pre $k \in collkeysh(rt, h)$
  var $p: Ptr$;
  begin
  $p := rt$
  ;
  pre $k \in collkeysh(p, h)$
    while $k \neq p \uparrow mk$ **do**
    inv $k \in collkeysh(p, h)$
      with $p \uparrow$ **do**
      if $k < mk$
      then $p := lp$
      else $p := rp$;
    rel $findbhn(k, p) = findbhn(k, \overleftarrow{p}) \wedge depth(p) < depth(\overleftarrow{p})$
  post $p = findbhn(k, p)$
  ;
  $FINDBH := p \uparrow md$
post $d = md(findbhn(k, \overleftarrow{rt}))$
end

Figure 10.13 Development of *FIND*

11

A Small Case Study

Formalization is an experimental science.
Dana Scott

The main purpose of this chapter is to pull together the strands of the development method presented in the book: one example is used to indicate the text to be created for specification, design of data structures (and verification thereof), and design of code (and its verification). There is no pretension as to size in calling this a 'case study'. Clearly, textbooks are not the ideal receptacles for industrial size applications. (Apart from anything else, this author's own experience in industry convinces him that a medium more dynamic than a printed book would be required!) This example has purposefully been chosen to be small so as to explore all of the stages of a development. The companion case studies book [JS90] includes significant fragments of larger problems and the *Teacher's Notes* contains a host of references to industrial use.

A subsidiary purpose of this chapter is to develop a (slightly mixed) analogy on the roles of proof in mathematics and in the design of computer systems. It is made clear above that one should not talk about a program 'being correct' but only of its 'satisfying a (formal) specification'. The obvious analogy then is to regard the claim that a program satisfies its specification as the statement of a theorem and to regard all of the intervening stages of development and the detail of discharging the relevant proof obligations as the proof of the theorem. Many objections can be raised to this attempted analogy. Here, three main differences with theorems and proofs in mathematics are considered. Firstly, including the code in the statement of the theorem results in texts which are large when compared with whole papers – if not books; they certainly bear no relation to the length of the statement of mathematical theorems. Secondly, the proof – which is even larger – is denied the structure (of lemmas, etc.) beloved of mathematical presentations. Thirdly, there is almost no precedent in mathematics for proofs at the level of detail used even in this chapter.

A different analogy is needed. It is perhaps more appropriate to regard the specification as the statement of a theorem that an implementation exists. The (multistage) development is then a proof of this claim. This naturally leads one to view the choice of steps of development as the major decompositions of the argument. This comparison gives a much more realistic estimate of the amount of intellectual effort required to find the proper joints at which to break a problem.

What then is to be made of the sorts of detailed proofs which occupy so much of this book? Clearly, there is a pedagogic need to begin work on proofs with easily understandable examples. Furthermore, it is precisely the hindrance of the low level of detail required which can be ameliorated by the development of 'theories of data types' as illustrated below. But it would still be useful to have a mathematical analogy for a task which does appear to occupy so much time in formal program development. It is perhaps (and this is where the analogy becomes mixed) useful to compare intermediate steps of design such as the creation of a loop – together with its invariant and relation – with integration in calculus; this naturally prompts a comparison between the detailed use of a proof rule and the differentiation with which careful mathematicians check their integrand. This analogy gives a rationale for the level of (somewhat shallow) detail required in discharging proof obligations and emphasizes the need for mechanical support. It is the intention in this chapter, however, to use less formal proofs for the algorithms themselves than for the data type theories.

11.1 Partitions of a fixed set

Partitions revisited

The task for which an implementation is sought in this chapter is a variant of the 'equivalence relation' problem used in Chapters 4, 6 and 8. The changes from the set of operations used above both present new interest and open the way to a particularly efficient implementation.

The set of operations might be motivated by the need to keep track of equivalent component numbers in a manufacturing environment. Equivalences over some fixed set X are created by an $EQUATE$ operation and pairs of $e_1, e_2 \in X$ are tested for equivalence by $TEST$; initially, the whole set of X is present but no two unequal elements are considered to be equivalent.

This section introduces the objects (*Part*) which are used in the Section 11.2 as the basis of the operation specifications.

$$Part = (X\textbf{-set})\textbf{-set}$$
$$\textbf{inv}\ (p) \triangleq \bigcup p = X \wedge \textit{is-prdisj}(p) \wedge \{\,\} \notin p$$

Remember that:

$$\textit{is-prdisj} : (X\textbf{-set})\textbf{-set} \to \mathbb{B}$$
$$\textit{is-prdisj}(ss) \quad \triangleq \quad \forall s_1, s_2 \in ss \cdot s_1 = s_2 \vee \textit{is-disj}(s_1, s_2)$$

$$\textit{is-disj} : X\textbf{-set} \times X\textbf{-set} \to \mathbb{B}$$
$$\textit{is-disj}(s_1, s_2) \quad \triangleq \quad s_1 \cap s_2 = \{\,\}$$

Notice that the first conjunct of *inv-Part* is an addition to the invariant for *Partition* of earlier chapters; it expresses the fact that the equivalence relations considered in this chapter are over some fixed set.

Some lemmas

As well as the objects themselves, some theory is developed.

Lemma 11.1 The finest partition of X is the set which contains unit sets each of which contains one element of X.

$$\boxed{\text{L11.1}}\ \frac{}{\{\{x\} \mid x \in X\} \in Part}$$

A proof of Lemma 11.1 is given on page 272. This lemma is straightforward and the proof is not given very formally. Of more interest is the proof that merging sets within a partition yields a partition. (A similar result was suggested, but not proven, in Section 4.2.) With a truth-valued function $t : X\textbf{-set} \to \mathbb{B}$ the merging is achieved using:

$$merge : Part \times (X\textbf{-set} \to \mathbb{B}) \to Part$$
$$merge(p, t) \quad \triangleq \quad \{s \in p \mid \neg\, t(s)\} \cup \{\bigcup\{s \in p \mid t(s)\}\}$$

Lemma 11.2 The claim that merging preserves the property of being a partition can be written:

$$\boxed{\text{L11.2}}\ \frac{p \in Part;\ t : X\textbf{-set} \to \mathbb{B};\ \exists s \in p \cdot t(s);\ p' = merge(p, t)}{p' \in Part}$$

A proof is given on page 273. Notice how the third hypothesis is needed at step 19 to ensure that empty sets cannot arise by t being false on all sets.

from *definitions*		
1	$\{\{x\} \mid x \in X\} \in (X\text{-}\mathbf{set})\text{-}\mathbf{set}$	*Set*
2	$\bigcup\{\{x\} \mid x \in X\}$	
	$= \{x \mid x \in X\}$	*Set*
3	$= X$	*Set*
4	$\textit{is-prdisj}(\{\{x\} \mid x \in X\})$	*is-prdisj*, *Set*
5	$s \in \{\{x\} \mid x \in X\} \Leftrightarrow \exists x \in X \cdot s = \{x\}$	*Set*
6	$\forall x \in X \cdot \{x\} \neq \{\,\}$	*Set*
7	$\{\,\} \notin \{\{x\} \mid x \in X\}$	5,6
infer	$\{\{x\} \mid x \in X\} \in \textit{Part}$	1,3,4,7,*Part*

Lemma 11.1

Pursuing the analogy about steps of development, the level of abstraction in this step has been useful to establish key properties of the final program.

11.2 Specification

The operations

Having constructed the theory of *Part*, it is now a simple task to specify the equivalence relation problem.

The initial partition is the 'finest' in which no two unequal elements are considered to be equivalent:

$$p_0 = \{\{x\} \mid x \in X\}$$

Lemma 11.1 shows that $p_0 \in \textit{Part}$.

The equivalence of elements is tested by:

$\textit{TEST}\ (e_1 : X, e_2 : X)\ r : \mathbb{B}$
ext rd $p\ :\ \textit{Part}$
post $r \;\Leftrightarrow\; \exists s \in p \cdot \{e_1, e_2\} \subseteq s$

Since this operation has only read access to p, its satisfiability relies only on the type correctness of *post-TEST*: this is trivial to see.

The operation which records in p that elements have been equated (and which reflects the consequences thereof) is more challenging. Its specification is:

from	$t\colon X\textbf{-set} \to \mathbb{B},\ p \in Part,\ \exists s \in p \cdot t(s),\ p' = merge(p)$	
1	$p' = \{s \in p \mid \neg\, t(s)\} \cup \{\bigcup\{s \in p \mid t(s)\}\}$	h,$merge$
2	$p \in (X\textbf{-set})\textbf{-set}$	h,$Part$
3	$\{s \in p \mid \neg\, t(s)\} \in (X\textbf{-set})\textbf{-set}$	2,h,Set
4	$\bigcup\{s \in p \mid t(s)\} \in X\textbf{-set}$	2,h,Set
5	$\{\bigcup\{s \in p \mid t(s)\}\} \in (X\textbf{-set})\textbf{-set}$	4,Set
6	$p' \in (X\textbf{-set})\textbf{-set}$	1,3,5,Set
7	$\bigcup p'$	
	$= \bigcup\{s \in p \mid \neg\, t(s)\} \cup \bigcup\{\bigcup\{s \in p \mid t(s)\}\}$	1,$\bigcup$
8	$= \bigcup\{s \in p \mid \neg\, t(s)\} \cup \bigcup\{s \in p \mid t(s)\}$	Set
9	$= \bigcup(\{s \in p \mid \neg\, t(s)\} \cup \{s \in p \mid t(s)\})$	Set
10	$= \bigcup p$	Set
11	$= X$	h,$Part$
12	$is\text{-}prdisj(p)$	h,$Part$
13	$is\text{-}prdisj(\{s \in p \mid \neg\, t(s)\})$	12,$is\text{-}prdisj$
14	$is\text{-}prdisj(\{\bigcup\{s \in p \mid t(s)\}\})$	$is\text{-}prdisj$
15	$\forall s \in \{s \in p \mid \neg\, t(s)\} \cdot is\text{-}disj(s, \{\bigcup\{s \in p \mid t(s)\}\})$	12,$is\text{-}prdisj$
16	$is\text{-}prdisj(p')$	1,$is\text{-}prdisj$,13,14,15
17	$\{\,\} \notin p$	h,$Part$
18	$\{\,\} \notin \{s \in p \mid \neg\, t(s)\}$	17,Set
19	$\bigcup\{s \in p \mid t(s)\} \neq \{\,\}$	17,Set,h
20	$\{\,\} \notin \{\bigcup\{s \in p \mid t(s)\}\}$	19,Set
21	$\{\,\} \notin p'$	1,18,20,Set
infer	$p' \in Part$	$Part$,6,11,16,21

Lemma 11.2

$EQUATE\ (e_1 \colon X, e_2 \colon X)$
ext wr $p\ :\ Part$
post $p = \{s \in \overleftarrow{p} \mid e_1 \notin s \wedge e_2 \notin s\} \cup \{\bigcup\{s \in \overleftarrow{p} \mid e_1 \in s \vee e_2 \in s\}\}$

Lemma 11.2 can be used to show that $EQUATE$ is satisfiable by observing that $e_1 \in s \vee e_2 \in s$ (whose negation by de Morgan's laws is $e_1 \notin s \wedge e_2 \notin s$) can be used in place of t which must be true for one or more $s \in p$ because of the invariant which ensures that all elements of X are present in a set. It follows, therefore, that:

Theorem 11.3 $EQUATE$ is satisfiable.

$$\forall \overleftarrow{p} \in Part, e_1, e_2 \in X \cdot \exists p \in Part \cdot post\text{-}EQUATE(e_1, e_2, \overleftarrow{p}, p)$$

This specification, following the opening analogy, is the statement of a theorem that an implementation exists. The task now is to find an efficient one.

Properties of the specification

As has been done with examples above, it is useful to check that the formal specifications of these operations satisfy intuitively acceptable properties. One might show:[1]

Theorem 11.4 Any element is equivalent to itself in any partition:

$$post\text{-}TEST(e_1, e_2, p, r) \ \vdash\ e_1 = e_2 \ \Rightarrow\ r$$

Theorem 11.5 In the initial state, such trivial equalities are the only tests which yield true:

$$p = \{\{x\} \mid x \in X\},\ post\text{-}TEST(e_1, e_2, p, r) \ \vdash\ r \ \Rightarrow\ (e_1 = e_2)$$

Theorem 11.6 Property 11.4 is called 'reflexivity'; 'symmetry' can be expressed by:

$$post\text{-}TEST(e_1, e_2, p, r_a),\ post\text{-}TEST(e_2, e_1, p, r_b) \ \vdash\ r_a \ \Leftrightarrow\ r_b$$

Theorem 11.7 In a similar way, the fact that the recorded relation is 'transitive' in any state can be expressed by:

[1] Type information such as $\overleftarrow{p}, p \in Part$; $e_i \in X$; $r_i \in \mathbb{B}$ has been omitted in all of these rules.

from	$e_1, e_2 \in X,\ \overleftarrow{p}, p \in Part,\ r \in \mathbb{B},$ $post\text{-}TEST(e_1, e_2, \overleftarrow{p}, r),\ post\text{-}EQUATE(e_1, e_2, \overleftarrow{p}, p)$	
1	$\neg\, r \vee r$	h,$\mathbb{B}$
2	**from** $\neg\, r$	
	infer $\neg\, r \vee (p = \overleftarrow{p})$	$\vee$-I(h2)
3	**from** r	
3.1	$\exists s \in \overleftarrow{p} \cdot \{e_1, e_2\} \subseteq s$	h,h3,*post-TEST*
3.2	**from** $s_a \in \overleftarrow{p}, \{e_1, e_2\} \subseteq s_a$	
3.2.1	$is\text{-}prdisj(\overleftarrow{p})$	h,*Part*
3.2.2	$\forall s_b \in \overleftarrow{p} \cdot s_b = s_a \vee is\text{-}disj(\{e_1, e_2\}, s_b)$	3.2.1,h3.2,*is-prdisj*
3.2.3	p	
	$= \{s \in \overleftarrow{p} \mid s \neq s_a\} \cup \{\bigcup\{s \in \overleftarrow{p} \mid s = s_a\}\}$	*post-EQUATE*,h3.2,3.2.2
3.2.4	$= \{s \in \overleftarrow{p} \mid s \neq s_a\} \cup \{s_a\}$	*Set*
	infer $= \overleftarrow{p}$	h3.2,*Set*
3.3	$p = \overleftarrow{p}$	$\exists$-E(3.1,3.2)
	infer $\neg\, r \vee (p = \overleftarrow{p})$	$\vee$-I(3.3)
infer	$\neg\, r \vee (p = \overleftarrow{p})$	$\vee$-E(1,2,3)

Lemma 11.8

$$
\begin{aligned}
& post\text{-}TEST(e_1, e_2, p, r_a), \\
& post\text{-}TEST(e_2, e_3, p, r_b), \\
& post\text{-}TEST(e_1, e_3, p, r_c) \vdash \\
& \qquad\qquad r_a \wedge r_b \;\Rightarrow\; r_c
\end{aligned}
$$

Theorem 11.8 The fact that equating two equivalent elements does not change the state is expressed:

$$post\text{-}TEST(e_1, e_2, \overleftarrow{p}, r),\ post\text{-}EQUATE(e_1, e_2, \overleftarrow{p}, p) \vdash \neg\, r \vee (p = \overleftarrow{p})$$

Proofs of the above results rely on fairly routine expansion of the definitions; as an example, Lemma 11.8 is proved on page 275.

Theorem 11.9 The fact that *EQUATE* does record the transitive consequences can be written:

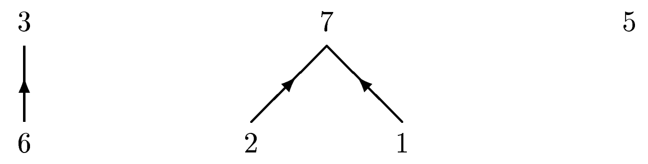

Figure 11.1 Fischer/Galler Trees

$$\begin{aligned}&\textit{post-TEST}(e_1, e_2, \overleftarrow{p}, r_a),\\ &\textit{post-EQUATE}(e_2, e_3, \overleftarrow{p}, p),\\ &\textit{post-TEST}(e_1, e_3, p, r_b) \vdash\\ &\qquad\qquad r_a \Rightarrow r_b\end{aligned}$$

11.3 A theory of forests

The Fischer/Galler idea

The description in Section 6.1 uses *Partrep*. Viewed as a specification, there is no worry about efficiency. But, as an implementation, the searching implied in *post-EQUATE* would be unacceptable for large collections of elements. The map provides fast response to *TEST* operations but not to *EQUATE*. The need to implement equivalence relations over very large collections of data has given rise to considerable research. The aim is to find a way of implementing both *TEST* and *EQUATE* efficiently. The technique, known after the names of its authors as the Fischer/Galler algorithm, employs a clever data structure in order to achieve efficiency. The basic idea is that equivalent elements should be collected into trees. These trees can be searched from any element to find a root. Two elements are equivalent if, and only if, they have the same roots. These trees – cf. Figure 11.1 – are unlike those formed from recursive abstract syntax definitions: there, the essential operations are to break up the trees into their sub-components. To *EQUATE* two elements it is necessary only to 'graft' the root of one element onto some point in the tree of the other element. Notice that it is essential that the grafted tree is taken by the root so that all equivalent elements are carried over.

A map model

The basic idea then is to use a representation of $X \xrightarrow{m} X$. There is a decision to be made about how the 'roots' are to be represented. Two alternatives are to make root elements map to themselves or to leave them out of the domain of the map. Either choice has advantages and disadvantages and some experimentation is needed to select the approach which results in the clearest presentation: although they are isomorphic, the choice between them affect the presentation of the theory. Representing root elements by mapping to themselves makes the map total and obviates the need for a case distinction in *post-EQUATE*.[2] Marking roots by their absence from the domain of the map makes it easier to discuss its well-foundedness and it is this choice which is followed here. Therefore, the set of roots can be determined by:

$$\begin{aligned}
&roots : (X \xrightarrow{m} X) \to X\text{-}\mathbf{set} \\
&roots(m) \quad \triangle \quad X - \mathbf{dom}\, m
\end{aligned}$$

But how do we know there are roots, or more generally, how do we know that there are no 'loops'? After all, $\{1 \mapsto 2, 2 \mapsto 1\} \in (\mathbb{N} \xrightarrow{m} \mathbb{N})$. Such loops would make it impossible to locate the roots of arbitrary elements. What is needed here is a notion of 'well-foundedness' that says the relation is such that one cannot follow its links for ever. There are several ways of expressing this idea.[3] One approach is to say that for all non-empty subsets of the domain of the map there must be at least one element which is mapped to an element not in the set:

$$\forall s \subseteq \mathbf{dom}\, m \cdot s \neq \{\,\} \;\Rightarrow\; \exists e \in s \cdot m(e) \notin s$$

(Note that a slight liberty with notation is taken here but $\forall s \subseteq X \cdot p(s)$ can be rewritten as $\forall s \in (X\text{-}\mathbf{set}) \cdot p(s)$.) If any set of maplets (including the unit set) were to represent a loop, their domain would be an s which prevented the above universal quantification from holding. The above formulation is perfectly usable but a higher-level of expression can be achieved if the same basic idea is expressed as:

$$\forall s \subseteq \mathbf{dom}\, m \cdot s \neq \{\,\} \;\Rightarrow\; \neg\,(\mathbf{rng}\,(s \lhd m) \subseteq s)$$

Lifting this definition to the relational view simplifies some of the proofs which follow.

Thus, formally:

$$Forest = X \xrightarrow{m} X$$

---

[2]This representation was used in [Jon79] and by several other authors.

[3]For general functions $f\colon X \to X$ the constraint is often expressed in mathematics books by saying that there must not exist a function $g\colon \mathbb{N} \to X$ such that $f(g(i)) = g(i+1)$ for all i.

$$\textbf{inv}\ (m) \triangleq \forall s \subseteq \textbf{dom}\, m \cdot s \neq \{\,\} \ \Rightarrow\ \neg\,(\textbf{rng}\,(s \lhd m) \subseteq s)$$

It is then possible to define:

$$root : X \times Forest \to X$$
$$root(e, f) \ \triangleq\ \textbf{if}\ e \in roots(f)\ \textbf{then}\ e\ \textbf{else}\ root(f(e), f)$$

That this function is total over *Forest* (but not over arbitrary $X \xrightarrow{m} X$) follows from the invariant.

The empty *Forest* is:

$$f_0 = \{\,\}$$

and satisfies *inv-Forest* because the only $s \subseteq \textbf{dom}\,\{\,\}$ is $\{\,\}$ which vacuously satisfies the implication.

A theory of forests

Of more interest is the way in which trees are grafted onto each other to define new *Forests* from old. It is pointed out above that the effect of *EQUATE* can only be achieved if the root of the tree to be grafted is found; trees will remain shorter if the graft is also made onto the root of the other tree. Since this also simplifies the reasoning, updates for this special case are considered (but see Exercise 11.5.1 on page 289).

Lemma 11.10 The key result is:

$$\boxed{\text{L11.10}}\ \frac{\overleftarrow{f} \in Forest;\ \{r_1, r_2\} \subseteq roots(\overleftarrow{f});\ r_1 \neq r_2;\ f = \overleftarrow{f} \cup \{r_1 \mapsto r_2\}}{f \in Forest}$$

The proof given on page 279 is argued at the element level. (Notice that $r_1 \notin \textbf{dom}\, f$ follows from the fact that r_1 is a root; thus f is intended to be like $\overleftarrow{f} \in Forest$ except that r_1 has been grafted onto r_2.) The level of reasoning in subsequent proofs can be heightened by defining:

$$collapse : Forest \to (X \xrightarrow{m} X)$$
$$collapse(f) \ \triangleq\ \{e \mapsto root(e, f) \mid e \in X\}$$

The well-definedness of *collapse* follows from the totality of *root* over *Forests*. Notice that, taking $Pid = X$, this function creates the *Partrep* of Section 6.1 from *Forest*.

The *collapse* function has some interesting properties.

Lemma 11.11 The fact that:

from $\overleftarrow{f} \in \mathit{Forest},\ \{r_1, r_2\} \subseteq \mathit{roots}(\overleftarrow{f}),\ r_1 \neq r_2,$

$f = \overleftarrow{f} \cup \{r_1 \mapsto r_2\}$

1 $r_1, r_2 \in X$ — roots(h)

2 $\overleftarrow{f} \in X \xrightarrow{m} X$ — h,Forest

3 $r_1 \notin \mathbf{dom}\ \overleftarrow{f}$ — roots(h)

4 $f \in X \xrightarrow{m} X$ — h,1,2,3,$\cup$

5 $\mathit{inv\text{-}Forest}(\overleftarrow{f})$ — h,Forest

6 $\forall s \subseteq \mathbf{dom}\ \overleftarrow{f} \cdot s \neq \{\,\} \Rightarrow \neg(\mathbf{rng}\,(s \lhd \overleftarrow{f}) \subseteq s)$ — $\mathit{inv\text{-}Forest}$,5

7 **from** $s \subseteq \mathbf{dom}\, f$

7.1 **from** $s \neq \{\,\}$

7.1.1 $s \subseteq (\mathbf{dom}\, f - \{r_1\}) \vee r_1 \in s$ — h7,Set

7.1.2 **from** $s \subseteq (\mathbf{dom}\, f - \{r_1\})$

7.1.2.1 $s \lhd f = s \lhd \overleftarrow{f}$ — h, h7.1.2,Map

7.1.2.2 $s \subseteq \mathbf{dom}\ \overleftarrow{f}$ — h,h7.1.2,Map

7.1.2.3 $s \neq \{\,\} \Rightarrow \neg(\mathbf{rng}\,(s \lhd \overleftarrow{f}) \subseteq s)$ — $\forall$-E(6,7.1.2.2)

7.1.2.4 $\neg(\mathbf{rng}\,(s \lhd \overleftarrow{f}) \subseteq s)$ — $\Rightarrow$-E(h7.1,7.1.2.3)

infer $\neg(\mathbf{rng}\,(s \lhd f) \subseteq s)$ — $=$-subs(7.1.2.4,7.1.2.1)

7.1.3 **from** $r_1 \in s$

7.1.3.1 $r_2 \notin \mathbf{dom}\ \overleftarrow{f}$ — roots,h

7.1.3.2 $\mathbf{dom}\, f = \mathbf{dom}\ \overleftarrow{f} \cup \{r_1\}$ — h,Map

7.1.3.3 $r_2 \notin \mathbf{dom}\, f$ — 7.1.3.1,7.1.3.2,h

7.1.3.4 $r_2 \notin s$ — h7.1.3.3,h7

infer $\neg(\mathbf{rng}\,(s \lhd f) \subseteq s)$ — h7.1.3,7.1.3.4,h,Map

infer $\neg(\mathbf{rng}\,(s \lhd f) \subseteq s)$ — $\vee$-E(7.1.1,7.1.2,7.1.3)

7.2 $\delta(s \neq \{\,\})$ — h7,Set

infer $s \neq \{\,\} \Rightarrow \neg(\mathbf{rng}\,(s \lhd f) \subseteq s)$ — $\Rightarrow$-I(7.1,7.2)

8 $\forall s \subseteq \mathbf{dom}\, f \cdot s \neq \{\,\} \Rightarrow \neg(\mathbf{rng}\,(s \lhd f) \subseteq s)$ — $\forall$-I(7)

9 $\mathit{inv\text{-}Forest}(f)$ — $\mathit{inv\text{-}Forest}$,8

infer $f \in \mathit{Forest}$ — Forest,4,9

Lemma 11.10

$$
\begin{array}{llr}
\textbf{from } f \in \mathit{Forest},\ \{r_1, r_2\} \subseteq \mathit{roots}(f),\ r_1 \neq r_2 & & \\
1 & \textbf{dom}\,(\mathit{collapse}(f \bigcup \{r_1 \mapsto r_2\}) \rhd \{r_2\}) & \\
 & \quad = \mathit{collect}(r_2, f \cup \{r_1 \mapsto r_2\}) & \mathit{Lemma} \\
2 & \quad = \mathit{collect}(r_1, f) \cup \mathit{collect}(r_2, f) & \mathit{Lemma} \\
\textbf{infer} & \quad = \textbf{dom}\,(\mathit{collapse}(f) \rhd \{r_1, r_2\}) & \mathit{Lemma}
\end{array}
$$

$$
\begin{array}{llr}
\textbf{from } f \in \mathit{Forest},\ \{r_1, r_2\} \subseteq \mathit{roots}(f),\ r_1 \neq r_2 & & \\
1 & \textbf{from } r \in \mathit{roots}(f),\ r \neq r_1,\ r \neq_2 & \\
1.1 & \quad \textbf{dom}\,(\mathit{collapse}(f \cup \{r_1 \mapsto r_2\}) \rhd \{r\}) & \\
 & \qquad = \mathit{collect}(r, f \cup \{r_1 \mapsto r_2\}) & \mathit{Lemma} \\
1.2 & \qquad = \mathit{collect}(r, f) & \mathit{Lemma} \\
 & \textbf{infer} \qquad = \textbf{dom}\,(\mathit{collapse}(f) \rhd \{r\}) & \mathit{Lemma} \\
\textbf{infer} & \mathit{above} &
\end{array}
$$

$$
\begin{array}{llr}
\textbf{from } f \in \mathit{Forest},\ e \in X & & \\
1 & r = \mathit{root}(e, f) & \\
 & \quad \Leftrightarrow\ e \in \{e \in X \mid \mathit{root}(e, f) = r\} & \mathit{Set} \\
2 & \quad \Leftrightarrow\ e \in \textbf{dom}\,(\{e \mapsto \mathit{root}(e, f) \mid e \in X\} \rhd \{r\}) & \mathit{Set} \\
\textbf{infer} & \quad \Leftrightarrow\ e \in \textbf{dom}\,(\mathit{collapse}(f) \rhd \{r\}) & \mathit{collapse}
\end{array}
$$

Properties of *collapse*

L11.11 $\overline{\forall e \in X \cdot (\mathit{collapse}(f))(e) = \mathit{root}(e, f)}$

follows immediately from its definition.

Another useful function – which finds all elements with a common root – is:

$\mathit{collect} : X \times \mathit{Forest} \to X\textbf{-set}$
$\mathit{collect}(r, f) \quad \triangleq \quad \{e \in S \mid \mathit{root}(e, f) = r\}$
$\textbf{pre } r \in \mathit{roots}(f)$

This can be seen to be total for *roots*.

Lemma 11.12 The relationship between *collect* and *collapse* should be clear:

$$\boxed{\text{L11.12}}\ \frac{f \in \mathit{Forest};\ r \in \mathit{roots}(f)}{\mathit{collect}(r,f) = \mathbf{dom}\,(\mathit{collapse}(f) \rhd \{r\})}$$

The operator $\rhd$ is a range restriction defined as:

$$m \rhd s \triangleq \{d \mapsto m(d) \mid d \in \mathbf{dom}\, m \wedge m(d) \in s\}$$

A plethora of properties can now be established:

Lemma 11.13

$$\boxed{\text{L11.13}}\ \frac{f \in \mathit{Forest};\ e \in X}{e \in \mathit{collect}(\mathit{root}(e,f),f)}$$

Lemma 11.14

$$\boxed{\text{L11.14}}\ \frac{f \in \mathit{Forest};\ \{r, r_1, r_2\} \subseteq \mathit{roots}(f);\ r_1 \neq r_2;\ r \neq r_1;\ r \neq r_2}{\mathit{collect}(r, f \cup \{r_1 \mapsto r_2\}) = \mathit{collect}(r,f)}$$

Lemma 11.15

$$\boxed{\text{L11.15}}\ \frac{f \in \mathit{Forest};\ \{r_1, r_2\} \subseteq \mathit{roots}(f);\ r_1 \neq r_2}{\mathit{collect}(r_2, f \cup \{r_1 \mapsto r_2\}) = \mathit{collect}(r_1,f) \cup \mathit{collect}(r_2,f)}$$

Lemma 11.16

$$\boxed{\text{L11.16}}\ \frac{f \in \mathit{Forest};\ \{r_1, r_2\} \subseteq \mathit{roots}(f);\ r_1 \neq r_2}{\mathit{is\text{-}disj}(\mathit{collect}(r_1,f), \mathit{collect}(r_2,f))}$$

Lemma 11.17 The preceding can then be raised to the *collapse* level as follows:

$$\boxed{\text{L11.17}}\ \frac{f \in \mathit{Forest};\ \{r, r_1, r_2\} \subseteq \mathit{roots}(f);\ r_1 \neq r_2;\ r \neq r_1;\ r \neq r_2}{\mathbf{dom}\,(\mathit{collapse}(f \cup \{r_1 \mapsto r_2\}) \rhd \{r\}) = \mathbf{dom}\,(\mathit{collapse}(f) \rhd \{r\})}$$

Lemma 11.18

$$\boxed{\text{L11.18}}\ \frac{f \in \mathit{Forest};\ \{r_1, r_2\} \subseteq \mathit{roots}(f);\ r_1 \neq r_2}{\mathbf{dom}\,(\mathit{collapse}(f \cup \{r_1 \mapsto r_2\}) \rhd \{r_2\}) = \mathbf{dom}\,(\mathit{collapse}(f) \rhd \{r_1, r_2\})}$$

Lemma 11.19

$$\boxed{\text{L11.19}}\ \frac{f \in \mathit{Forest};\ e \in X}{r = \mathit{root}(e,f) \;\Leftrightarrow\; e \in \mathbf{dom}\,(\mathit{collapse}(f) \rhd \{r\})}$$

The proofs are sketched on page 280.

Building such theories is the only way of avoiding having to tackle each proof from scratch; this *tabula rasa* situation is a major inhibitor to the use of proofs in program development. Although only shown to a limited extent in the next section, these lemmas could support a range of algorithms for this *Partition* problem and even the use of a *Forest* representation for other tasks. Such collections should be built independently of particular program developments.

Exercise 11.3.1 Do some of the above proofs.

Exercise 11.3.2 An alternative approach could be developed around:

$$
\begin{array}{l}
\mathit{is\text{-}before} : X \times X \times \mathit{Forest} \to \mathbb{B} \\
\mathit{is\text{-}before}(e, d, f) \quad \triangleq \\
\quad \textbf{if } e \in \mathit{roots}(f) \textbf{ then false else if } d = e \textbf{ then true else } \mathit{is\text{-}before}(f(e), d, f)
\end{array}
$$

$$
\begin{array}{l}
\mathit{trace} : X \times \mathit{Forest} \to X\textbf{-set} \\
\mathit{trace}(e, f) \quad \triangleq \quad \textbf{if } e \in \mathit{roots}(f) \textbf{ then } \{e\} \textbf{ else } \{e\} \cup \mathit{trace}(f(e), f)
\end{array}
$$

Develop a suitable set of lemmas to support the proofs in the next section.

11.4 The Fischer/Galler algorithm

Adequacy

The data structure of the preceding section (*Forest*) can now be used to provide a representation for the *Part* of Section 11.1 and thus a way of modelling the operations in Section 11.2. As explained in Chapter 8, the first step is to formally relate the two types with a retrieve function:

$$
\begin{array}{l}
\mathit{retr\text{-}Part} : \mathit{Forest} \to \mathit{Part} \\
\mathit{retr\text{-}Part}(f) \quad \triangleq \quad \{\mathit{collect}(r, f) \mid r \in \mathit{roots}(f)\}
\end{array}
$$

Notice that:

$$\mathit{retr\text{-}Part}(f) = \{\textbf{dom}\,(\mathit{collapse}(f) \rhd \{r\}) \mid r \in \mathit{roots}(f)\}$$

Theorem 11.20 It is then necessary to prove adequacy:

$$\forall p \in \mathit{Part} \cdot \exists f \in \mathit{Forest} \cdot \mathit{retr\text{-}Part}(f) = p$$

For $p \in \mathit{Part}$, it is clear that $p \in (X\textbf{-set})\textbf{-set}$ then:

$$\bigcup\{\{e \mapsto \mathit{min}(s) \mid e \in (s - \{\mathit{min}(s)\})\} \mid s \in p\}$$

is of type $(X \xrightarrow{m} X)$ because *inv-Part* guarantees that the sets $s \in p$ are non-empty and disjoint; the invariant *inv-Forest* holds trivially (notice *collapse* is an identity on these squashed trees); and *retr-Part* gives the required result.

It should now be clear that the restriction of $\{\,\} \notin p$ is necessary to ensure adequacy: the representation has no way of distinguishing between the presence and absence of anything corresponding to the empty set.

Justifying the operations

Theorem 11.21 It is easy to see that the initial states relate:

$$\boxed{\text{L11.21}}\;\frac{f_0 = \{\,\}}{retr\text{-}Part(f_0) = \{\{x\} \mid x \in X\}}$$

since all $x \in X$ are *roots* in the empty map.

The *TEST* operation is now specified as:

$TEST\ (e_1{:}\,X, e_2{:}\,X)\ r{:}\,\mathbb{B}$
ext rd $f\ :\ Forest$
post $r\ \Leftrightarrow\ (root(e_1, f) = root(e_2, f))$

This looks straightforward and, picking up our introductory analogy, represents the major insight (or, in terms of the analogy advanced at the beginning of this chapter, 'integrand'); the check requires the detailed work of generating and discharging the relevant proof obligations. The satisfiability of *TEST* on *Forest* is trivial because *root* is applied to appropriate arguments. There is no domain rule to be discharged since *pre-TEST* is, by convention, **true**.

Theorem 11.22 The interesting result is therefore to show:

$$(\exists s \in retr\text{-}Part(f) \cdot e_1 \in s \wedge e_2 \in s)\ \Leftrightarrow\ root(e_1, f) = root(e_2, f)$$

which is straightforward (given the lemmas) – see page 284.

This concludes the justification for *TEST* (on *Forest*). Clearly, more work is to be expected for *EQUATE*. Its specification is:

$EQUATE\ (e_1{:}\,X, e_2{:}\,X)$
ext wr $f\ :\ Forest$
post $root(e_1, \overleftharpoon{f}) = root(e_2, \overleftharpoon{f}) \wedge f = \overleftharpoon{f} \vee$
$\quad root(e_1, \overleftharpoon{f}) \neq root(e_2, \overleftharpoon{f}) \wedge f = \overleftharpoon{f} \cup \{root(e_1, \overleftharpoon{f}) \mapsto root(e_2, \overleftharpoon{f})\}$

In fact, there is a trap for the unwary here: if the post-condition were written just as $f = \overleftharpoon{f} \cup \{r_1 \mapsto r_2\}$ it would be possible when $root(e_1, \overleftharpoon{f}) = root(e_2, \overleftharpoon{f})$ to create loops in the $X \xrightarrow{m} X$ and thus violate *inv-Forest*. It is for this reason that so many of the lemmas in the preceding section needed the hypothesis $r_1 \neq r_2$.

from $f \in Forest,\ e_1, e_2 \in X$

$$\exists s \in retr\text{-}Part(f) \cdot e_1 \in s \wedge e_2 \in s$$

$$\Leftrightarrow\ \exists s \in \{collect(r, f) \mid r \in roots(f)\} \cdot e_1 \in s \wedge e_2 \in s \qquad retr\text{-}Part$$

infer $\Leftrightarrow\ root(e_1, f) = root(e_2, f)$ *Lemma*

Lemma 11.22

The satisfiability of *EQUATE* on *Forest* follows immediately from Lemma 11.10. There is, again, no domain condition to be considered.

Theorem 11.23 The more interesting part of the result proof obligation becomes:

$$\overleftarrow{f} \in Forest,\ e_1, e_2 \in X,\ \overleftarrow{p} = retr\text{-}Part(\overleftarrow{f}),$$
$$r_1 = root(e_1, \overleftarrow{f}),\ r_2 = root(e_2, \overleftarrow{f}),\ r_1 \neq r_2 \ \vdash$$
$$retr\text{-}Part(\overleftarrow{f} \cup \{r_1 \mapsto r_2\}) =$$
$$\{s \in \overleftarrow{p} \mid e_1 \notin s \wedge e_2 \notin s\} \cup \{\bigcup\{s \in \overleftarrow{p} \mid e_1 \in s \vee e_2 \in s\}\}$$

This proof is given on page 285.

The definition of *post-EQUATE* is overspecific in that it would be possible to graft the trees in the other order. A non-deterministic specification could be constructed in order to avoid this commitment. It would even be possible to graft the root of one tree onto some arbitrary point in the other. There is, however, a considerable incentive to keep the trees as short as possible. That is, the depth of any branch of the tree must be kept as low as possible. This follows from the use of the *root* function in both of the main operations. It would be ideal if trees could be kept to a maximum depth of one. Irrespectively of the order in which *EQUATE* is made to graft the trees, they can become deeper than this ideal. The overall efficiency of the Fischer/Galler algorithm is, however, very good. The search time is proportional to the average depth of a tree – rather than the number of elements.

Exercise 11.4.1 Repeat the third part of Exercise 6.1.4 on page 145 on a state using *Forest*; also specify *ELS* but comment on the implementation problem with this operation.

Exercise 11.4.2 (*) Another representation for the equivalence relation application would be to have two different data structures. One of these would support the *TEST* operation and would store the map discussed in Section 6.1; the other

from $\overleftarrow{f} \in Forest,\ e_1, e_2 \in X,\ \overleftarrow{p} = retr\text{-}Part(\overleftarrow{f}),$
$\quad r_1 = root(e_1, \overleftarrow{f}),\ r_2 = root(e_2, \overleftarrow{f}),\ r_1 \neq r_2$

1 $\quad retr\text{-}Part(\overleftarrow{f} \cup \{r_1 \mapsto r_2\})$

$\quad = \{\mathbf{dom}\,(collapse(\overleftarrow{f} \cup \{r_1 \mapsto r_2\}) \rhd \{r\}) \mid r \in (roots(\overleftarrow{f}) - \{r_1\})\}$ *retr-Part*,*Set*

2 $\quad = \{\mathbf{dom}\,(collapse(\overleftarrow{f} \cup \{r_1 \mapsto r_2\}) \rhd \{r\}) \mid r \in (roots(\overleftarrow{f}) - \{r_1, r_2\})\} \cup \{\mathbf{dom}\,(collapse(\overleftarrow{f} \cup \{r_1 \mapsto r_2\}) \rhd \{r_2\})\}$ *Set*

3 $\quad = \{\mathbf{dom}\,(collapse(\overleftarrow{f}) \rhd \{r\}) \mid r \in (roots(\overleftarrow{f}) - \{r_1, r_2\})\} \cup \{\mathbf{dom}\,(collapse(\overleftarrow{f} \cup \{r_1 \mapsto r_2\}) \rhd \{r_2\})\}$ L11.17

4 $\quad = \{\mathbf{dom}\,(collapse(\overleftarrow{f}) \rhd \{r\}) \mid r \in (roots(\overleftarrow{f}) - \{r_1, r_2\})\} \cup \{\mathbf{dom}\,(collapse(\overleftarrow{f}) \rhd \{r_1, r_2\})\}$ L11.18

5 $\quad = \{s \in \{\mathbf{dom}\,(collapse(\overleftarrow{f}) \rhd \{r\}) \mid r \in roots(f)\} \mid e_1 \notin s \wedge e_2 \notin s\} \cup \{\bigcup\{\mathbf{dom}\,(collapse(\overleftarrow{f}) \rhd \{r\}) \mid r \in roots(f)\} \mid e_1 \in s \vee e_2 \in s\}$ h,L11.19

infer $\quad = \{s \in \overleftarrow{p} \mid e_1 \notin s \wedge e_2 \notin s\} \cup \{\bigcup\{s \in \overleftarrow{p} \mid e_1 \in s \vee e_2 \in s\}\}$ *retr-Part*,h

Result rule for *EQUATE* on *Forest*

would link all elements in the same equivalence class into a ring – *EQUATE* can then locate all keys in the first data structure which need updating. Specify this development and justify its correctness.

11.5 Operation decomposition

Pascal data structures

The preceding section has brought the representation close to the level that could be used directly in a Pascal-like language; this section must show how to achieve the effect of the post-conditions in terms of primitive operations of the chosen implementation language. (As in Chapter 10, no particular language is intended but it should be clear how to translate what is written here into Pascal.) Clearly then code is required for $TEST$, $EQUATE$ and to create the initial state. It is convenient also to write a separate function for $ROOT$.

Assuming that the type X is a subset of $\mathbb{N}$ ($X = \{1, \ldots, n\}$), *Forests* can be represented in an **array** providing there is some way of representing *roots*. Remember that in Section 11.3 it was decided to denote a root by its not being in the domain of the map. This is one of the ways in which the convenient mathematical abstraction of Chapter 6 is more general than the arrays of those programming languages which essentially just pass on to the programmer the restrictions of addressing from von Neumann architecture. In this case, however, it is easy to circumvent the difficulty by making the array:

$$a\text{: } \mathbf{array}\ X\ \mathbf{to}\ X_0$$

with $X_0 = \{0, \ldots, n\}$ and redefining:

$$roots(a) \quad \triangleq \quad \{i \in X \mid a[i] = 0\}$$

Because of its use of *roots*, the function *root* needs no revision. A new function, which determines the 'depth' or distance from the root is required in the argument below:

$$depth(e, a) \quad \triangleq \quad \mathbf{if}\ e \in roots(a)\ \mathbf{then}\ 0\ \mathbf{else}\ depth(a[e], a) + 1$$

Although it would be easy to provide, no formal argument about this revised representation of *Forest* is given here: the design step is considered to be small enough that it can be made safely without such formality. Of course, as is always the case in the development method presented here, it is clear what would need to be done to provide progressively more formality (i.e. begin with *retr-Forest*).

The initialization of the array can be achieved by:

$$\mathbf{for}\ i = 1\ \mathbf{to}\ n\ \mathbf{do}\ a[i] := 0$$

which achieves the condition that $roots(a) = X$.

Function $ROOT$

As mentioned above, it is convenient to separate a function to locate the root of an element: an annotated program for $ROOT$ is shown in Figure 11.2. A few comments on its correctness annotations might be helpful. All assertions of the form $a = \overleftarrow{a}$ have been omitted because $ROOT$ only has read access to a. (Clearly, in a complete support system it would be necessary to check that such constraints were respected by the code.) The essence of the **while** construct is to compute the root of v so *toend* (in inference rule **while**-$I2$) is $v = root(\overleftarrow{v}, a)$. The result of the loop can be combined with the ;-I rule and the

$ROOT(e{:}\,X)\ X$
ext rd a: **array** X **to** X_0
assert *inv-Forest*
begin
var $v{:}\,X$;
pre true
  $v := e$;
  pre *true*
  while $a[v] \neq 0$ **do**
  inv $depth(v, a) \in \mathbb{N}$
    $v := a[v]$
  toend $v = root(\overleftarrow{v}, a)$
  post $v = root(\overleftarrow{v}, a)$
post $v = root(e, a)$
$ROOT := v$
end

Figure 11.2 Annotated code for $ROOT$

initializing assignment $v := e$ to justify the overall condition $v = root(e, a)$. The termination of the loop follows from the decrease at each iteration of the *depth* of v. The fact that this is a natural number (i.e. the tree has no loops) follows from the invariant.

Remaining code

Given the $ROOT$ function, it is easy to program both $TEST$ and $EQUATE$. The annotated code for $TEST$ is shown in Figure 11.3. It is necessary to note that $ROOT$ has read-only access to a in order to carry forward the information about v_1 to the second assertion. This information is also necessary in order to check that $TEST$ respects its read-only constraint.

The annotated code for $EQUATE$ is given in Figure 11.4. Similar observations to those above about preserving the root assertions hold here. In addition, it is necessary to comment on the change in *post-EQUATE* from using map union to map override in defining the relationship between a and $\overleftarrow{a}$. It is a property of maps that the change in this direction is always valid (cf. Lemma 6.8 on page 150) and it more clearly represents the change made to the array.

The code presented in this section satisfies the specification given in Section 11.2. It is far easier to see that this is true having related the *Forest* type to

$TEST(e_1: X, e_2: X)\ \mathbb{B}$
ext rd a: **array** X **to** X_0
assert *inv-Forest*
begin
var $v_1, v_2: X$;
pre true
  $v_1 := ROOT(e_1)$;
  assert $v_1 = root(e_1, a)$
  $v_2 := ROOT(e_2)$;
  assert $v_1 = root(e_1, a) \wedge v_2 = root(e_2, a)$
  $TEST := (v_1 = v_2)$
post $TEST \;\Leftrightarrow\; (root(e_1, a) = root(e_2, a))$
end

Figure 11.3 Annotated code for $TEST$

$EQUATE(e_1: X, e_2: X)$
ext wr a: **array** X **to** X_0
assert *inv-Forest*
begin
var $v_1, v_2: X$;
pre true
  $v_1 := ROOT(e_1, a)$;
  assert $v_1 = root(e_1, a)$
  $v_2 := ROOT(e_2, a)$;
  assert $v_1 = root(e_1, a) \wedge v_2 = root(e_2, a)$
  if $v_1 \neq v_2$ **then** $a[v_1] := v_2$
post $root(e_1, a) \neq root(e_2, a) \wedge a = \overleftharpoon{a} \dagger \{v_1 \mapsto v_2\} \vee$
  $root(e_1, a) = root(e_2, a) \wedge a = \overleftharpoon{a}$
end

Figure 11.4 Annotated code for $EQUATE$

Part than if one attempts to read the code alone. Thus, to pick up the analogy from the beginning of this chapter, the major steps of specification, representation choice and code present the overall proof of the theorem that an (efficient) implementation exists; the lemmas and loops are like integrands whose value is cross-checked by detailed proofs of the created proof obligations. Even here, the weight of this burden would be shared when the results in the theories were used in other algorithms.

Exercise 11.5.1 (*) Short bushy trees take less steps to search than tall thin ones. It is for this reason that the graft is performed onto the root of e_2 rather than onto e_2 itself. Convince yourself that, even so, the algorithms given above can – with worst case data – result in tall thin trees. Develop a modification of *EQUATE* which compresses the tree each time it is traced back to its root. (A presentation of this algorithm is given in [Dij76]. The theories presented in this chapter have been used in a variety of other justifications including the design of a concurrent tree compression routine in [Jon83].)

Exercise 11.5.2 (*) Repeat the whole development of Sections 11.3–11.5 using a forest representation with loops at the roots.

12

Postscript

> If we try to solve society's problems without overcoming the confusion and aggression in our own state of mind, then our efforts will only contribute to the basic problems, instead of solving them.
> Chögyam Trungpa

The decision to write a personal postscript to this book was partly prompted by my involvement in a panel discussion on Social Responsibility at the TAPSOFT conference in Berlin. Computer systems are now so widely used that computer scientists must consider where they stand on issues relating to the systems they build. We should not expect others to accept our judgements, but we should provoke discussion and be prepared to accept criticism. A crucial issue is the reliance being put on computer systems. The probability of random (physical) hardware errors has been decreased significantly over the last twenty years, but software (and hardware) *design* errors persist. One clear personal responsibility is not to oversell our ideas. This postscript attempts to put the proposals made in this book into a slightly wider context.

One must first recognize that there are many problems associated with the development of computer systems. Some of these problems have nothing at all to do with specifications (formal or otherwise).

The material relating to specifications in this book attempts to show how mathematical notation can be used to increase the precision of a specification. The mathematical notation can, when used with care, achieve conciseness of expression as well as precision. I believe that these ideas are important. But a major issue relating to specifications is whether they match the user's requirements. The idea of proving properties of formal specifications is proposed above. But it is also conceded that this can never ensure a match with the, inherently informal, requirements. One can argue that this match can only be tested in the same way in which a scientific theory is tested. It is also possible to claim that Popper's arguments for refutability are a support for formality on the specification side of the comparison – and experience supports this claim. But the fact that there is no way of proving that a system matches the user's requirements should force us to consider, in every system with which we are involved, the danger of a mismatch.

The material in this book relating to design aims to provide developers with ways to increase their confidence that the systems they create satisfy the specifications. This must be a part of a software engineer's training. With machine-checked proofs, an enormous increase in confidence would be justified, but it must be understood that nothing can ever provide absolute certainty of correctness. The same is, of course, true of physical systems. Designing a system requires comparing probabilities of error in different sub-systems.

There is a great danger associated with people's perception of new concepts. If improved methods are used to tackle the same sort of problems previously handled by *ad hoc* methods, the systems created could be far safer. If, on the other hand, the improved methods are used to justify tackling systems of even greater complexity, no progress will have been made.

A

Glossary of Symbols

Function Specification

$f\ (d{:}\,D)\ r{:}\,R$
pre $\ldots d \ldots$
post $\ldots d \ldots r \ldots$

Operation Specification

$OP\ (d{:}\,D)\ r{:}\,R$
ext rd $e_1 : T_1$,
wr $e_2 : T_2$
pre $\ldots d \ldots e_1 \ldots e_2 \ldots$
post $\ldots d \ldots e_1 \ldots \overleftarrow{e_2} \ldots r \ldots e_2 \ldots$

Functions

$f{:}\,D_1 \times D_2 \to R$	signature
$f(d)$	application
if ... **then** ... **else** ...	conditional
let $x = \ldots$ **in** ...	local definition

Numbers

$\mathbb{N}_1$	$\{1, 2, \ldots\}$
$\mathbb{N}$	$\{0, 1, 2, \ldots\}$
$\mathbb{Z}$	$\{\ldots, -1, 0, 1, \ldots\}$
$\mathbb{Q}$	rational numbers
$\mathbb{R}$	real numbers
$+, -, *, \uparrow, <$	normal (infix) arithmetic operators
abs	(prefix) absolute value
mod	(infix) modulus

Logic

$\mathbb{B}$	$\{\mathbf{true}, \mathbf{false}\}$
$\neg E$	negation (not)
$E_1 \wedge E_2$	conjunction (and)
	E_1, E_2 are conjuncts
$E_1 \vee E_2$	disjunction (or)
	E_1, E_2 are disjuncts
$E_1 \;\Rightarrow\; E_2$	implication
	E_1 antecedent, E_2 consequent
$E_1 \;\Leftrightarrow\; E_2$	equivalence
$\forall x \in S \cdot E$	universal quantifier [1]
$\exists x \in S \cdot E$	existential quantifier
$\exists!\, x \in S \cdot E$	unique existence
$\Gamma \vdash E$	sequent
	Γ hypothesis, E conclusion
$\dfrac{\Gamma}{E}$	inference rule
$\dfrac{E_1}{E_2}$ (double line)	bi-directional inference rule

Composite Objects

$::$	compose
nil	omitted object
$mk\text{-}N(\ldots)$	generator
$s_1(o)$	selector
$\mu(o, s_1 \mapsto t)$	modify a component

[1] With all of the quantifiers, the scope extends as far as possible to the right; no parentheses are required but they can be used for extra grouping.

Sets

T-**set**	all finite subsets of T
$\{t_1, t_2, \ldots, t_n\}$	set enumeration
$\{\,\}$	empty set
$\{x \in S \mid p(x)\}$	set comprehension
$\{i, \ldots, j\}$	subset of integers (from i to j inclusive)
$t \in S$	set membership
$t \notin S$	$\neg\,(t \in S)$
$S_1 \subseteq S_2$	set containment (subset of)
$S_1 \subset S_2$	strict set containment
$S_1 \cap S_2$	set intersection [2]
$S_1 \cup S_2$	set union
$S_1 - S_2$	set difference
$\bigcup SS$	distributed union
card S	cardinality (size) of a set

Maps

$D \xrightarrow{m} R$	finite maps
$D \xleftrightarrow{m} R$	One-one map
dom m	domain
rng M	range
$\{d_1 \mapsto r_1, d_2 \mapsto r_2, \ldots, d_n \mapsto r_n\}$	map enumeration
$\{\,\}$	empty map
$\{d \mapsto f(d) \in D \times R \mid p(d)\}$	map comprehension
$m(d)$	application
m^{-1}	map inverse
$s \lhd m$	domain restriction
$s \mathrel{\unlhd\!\!\!\!-} m$	domain deletion
$m \rhd t$	range restriction
$m_1 \dagger m_2$	overwriting

[2] Intersection is higher priority than union.

Sequences

T^*	finite sequences
T^+	non-empty, finite sequences
len s	length
$[t_1, t_2, \ldots, t_n]$	sequence enumeration
$[\,]$	empty sequence
$s_1 \curvearrowright s_2$	concatenation
dconc ss	distributed concatenation
hd s	head
tl s	tail
inds s	indices
elems s	elements
$s(i, \ldots, j)$	sub-sequence

B

Glossary of Terms

Absorption An operator is absorptive if x **op** $x = x$ for all valid operands.

Abstract syntax An abstract syntax defines the structure of objects. The term was first used in the description of programming languages where objects which are defined abstract away from the details of the concrete syntax which has to include syntactic clues for parsing: in the abstract syntax only the necessary information content is present. The semantic definition of a language is normally based on its abstract syntax.

Abstraction The process of excluding unnecessary details so as to focus attention on the essential aspects of a system, problem, etc.

Adequacy The adequacy proof obligation – which is used in data reification – establishes that there is at least one representation for each abstract value.

ADJ diagram An ADJ diagram provides a graphical representation of the signatures of the operators of a data type.

Antecedent The left-hand side of an implication is its antecedent.

Application A function or map is applied to an element in its domain; the result is an element of the range.

Associativity An operator is associative if x **op** $(y$ **op** $z) = (x$ **op** $y)$ **op** z for all valid operands.

Backus-Naur Form (BNF) BNF is the notation used to define the concrete syntax of ALGOL 60; BNF or some variant thereof is now used in most language descriptions.

Bag A bag (also known as multiset) is an unordered collection of values where values can be contained more than once (thus it is possible to count the occurrences).

Basis In an inductive proof, the basis is the subsidiary proof that the required expression is true for the minimum element (or minimal elements) of the set of values.

Behaviour The behaviour of a data type determines (for a functional data type) the result of its operators and functions or (for a state-based data type) of its operations. In particular, for a collection of operations the behaviour is the relationship established between the inputs and outputs of the operations – these are the externally visible effects while the state changes are hidden from the user of the operations.

Bias *See* implementation bias.

BNF *See* Backus-Naur Form.

Body The body of a quantified expression is that expression following the raised dot.

Bound identifiers In a quantified expression the bound identifiers are those appearing after the quantifier; all free occurrences of the identifier in the body of such an expression are bound in the overall quantified expression. There are other ways of binding identifiers – for example, the names corresponding to the values of parameters and external variables are bound within an operation specification.

Cardinality The cardinality of a finite set is the number of elements contained in the set.

Commutativity An operator is commutative if x **op** $y = y$ **op** x for all valid operands.

Complete An axiomatization is complete with respect to a model if all statements which are true in that model can be proved from the axioms using the rules of inference.

Composite objects Composite objects are tagged Cartesian products; they are created by make-functions.

Composite type A composite type defines a set of composite objects.

Concatenation The concatenation operator creates a sequence from the elements of its two (sequence) operands; the result contains the elements of the first sequence followed by the elements of the second.

Conclusion In a sequent, the conclusion is the logical expression on the right of a turnstile.

Concrete syntax The concrete syntax of a language defines the set of strings which form sentences of the language. One notation for defining a concrete syntax is BNF.

Conjunction A logical expression whose principal operator is 'and' ($\wedge$) is a conjunction.

Consequent The right-hand side of an implication is its consequent.

Constraint The constraint of a quantified expression fixes the type of the identifier(s) bound by the quantifier; it governs the values over which the variable(s) ranges.

Contingent A logical expression is contingent if there are contexts in which it evaluates to **true** while in others it evaluates to **false**.

Contradiction A logical expression is a contradiction if there is no context in which it evaluates to **true**.

Data reification Abstract objects are reified to chosen representations in (the early stages of) system development from a specification. Chapter 8 describes how data reification steps are made in VDM.

Data type A data type is a set of values together with ways of manipulating those values; functional data types (e.g. natural numbers or sequences) have operators or functions whose results depend only on their arguments; state-based data types are manipulated by operations whose result is affected by and whose execution affects a state.

Data type invariant A data type invariant is a truth-valued function which defines a subset of a class of objects.

Decidable A logical calculus is decidable if an algorithm exists which can determine, for any expression of the calculus, whether the formula is true or not.

Decomposition *See* operation decomposition.

Definition (direct) A direct definition of a function provides a rule for computing the result of applying the function to its arguments.

Derived rule Derived rules are conclusions from an axiomatization of a theory which can be used in constructing further proofs.

Difference The difference of two sets is the set containing exactly those elements of the first set which are absent from the second.

Disjoint sets Two sets are disjoint if they have no common elements; a collection of sets is pairwise disjoint if any two (different) sets in the collection are disjoint.

Disjunction A logical expression whose principal operator is 'or' ($\vee$) is a disjunction.

Distributed union The distributed union of a set of sets is the set containing exactly those elements of the sets which are themselves elements of the operand.

Distributivity An operator (**opa**) is said to left distribute over another operator (**opb**) if, for all valid operands, x **opa** (y **opb** z) = (x **opa** y) **opb** (x **opa** z); and conversely for right distribution.

Domain The domain of a function (map) is the set of values to which the function (map) can be applied.

Equations The equations of a property-oriented specification provide the semantics of a data type (without giving a model).

Equivalence An equivalence is a logical expression whose principal operator is an equivalence symbol ($\Leftrightarrow$).

Equivalence relation An equivalence relation is a relation which is reflexive, symmetric and transitive.

Equivalent Two logical expressions are equivalent if they yield the same value for all possible values of their free variables.

Exception The specification of exceptions can be separated from the normal pre- and post-conditions as shown in Section 9.2.

Existential quantifier An existential quantifier ($\exists$) can be read as 'there exists (one or more)'.

Final interpretation The final interpretation of a (property-oriented) specification is one in which values are considered to be equivalent if and only if their denoting expressions cannot be proved to be different by deductions from the equations.

Formal language A formal language is one which has precise syntax and semantics.

Formal proof A formal proof is one in which all steps are stated precisely and completely; thus a formal proof can be checked by a computer program.

Free variables The free variables of an expression are the identifiers which occur in the expression but are not bound (e.g. by a quantifier).

Full abstraction A specification is fully abstract if it is not biased (with respect to a given collection of operations).

Function A function is a mapping between two sets of values (i.e. from elements in the domain to elements in the range).

Functional specification A functional specification defines the intended input/output behaviour of a computer system: *what* the system should do.

Generators The generators of a type are the functions which can, in suitable combinations, generate all values of the type (e.g. 0 and *succ* for the natural numbers).

Hypothesis A logical expression on the left of a sequent is (one of) its hypotheses.

Implementation bias A model-oriented specification is biased (towards certain implementations) if equality on the states cannot be defined in terms of the available operations; in other words, there are two, or more, state values which cannot be distinguished by the operations. (See Section 9.3 for a fuller discussion.)

Implication An implication is a logical expression whose principal operator is an implication sign ($\Rightarrow$).

Implicit specification An implicit specification characterizes *what* is to be done without (if possible) saying anything about *how* the result is to be achieved.

Indexing The application of a sequence to a valid index is called indexing; it yields an element of the sequence.

Induction rule An induction rule is an inference rule which facilitates proofs about infinite classes of (finite) objects; typically, there is a base case and an inductive step to be proved.

Induction step In an inductive proof, the inductive step shows that the required expression inherits over the successor function for the type.

Inductive hypothesis In the induction step of an inductive proof, the induction hypothesis is the assumption of the required property from which its inheritance has to be proved.

Inductive proof An inductive proof is one which uses the induction principle for a type.

Inference rule An inference rule consists of a number of hypotheses and a conclusion separated by a horizontal line; an appropriate instance of the conclusion is justified if corresponding matches can be made with the hypotheses. A bi-directional rule can also be used from bottom to top.

Initial interpretation The initial interpretation of a property-oriented specification is one in which values are considered to be equivalent if, and only if, their denoting expressions can be proven to be equal from the equations.

Intersection The intersection of two sets is the set containing exactly those elements contained in both sets.

Invariants *See* data type invariant or loop invariant.

Logic of partial functions (LPF) LPF is a logic which copes with undefined terms. The rules of this logic are given in Appendix C.

Loop invariant A loop invariant is basically just a data type invariant which defines the subset of states which can arise at the head of a loop construct like **while**.

LPF *See* logic of partial functions.

Make-function Each composite type has an associated make-function which forms elements of the type from elements of the sets of values for the fields of the composite object.

Map Map values define a finite (many-to-one) relationship between two sets; the map can be applied to elements in its domain to find the corresponding element in the range.

Maplet The ordered pairs of an explicitly given map are written as maplets with the two values separated by a small arrow ($\mapsto$).

Model oriented A model-oriented specification of a data type defines the behaviour of its operators in terms of a class of objects known as its state; this state is a 'hidden' type in the sense that it is not a part of the visible behaviour of the operations.

Model theory A model theory for a calculus associates its formulae with a collection of mathematical objects.

Module A module in BSI-VDM combines a state with a collection of operations; such modules correspond to data types.

Modus ponens *Modus ponens* is an inference rule which from $(E_1 \Rightarrow E_2)$ and E_1 justifies E_2.

Monotone A function is monotone with respect to some ordering if its application respects that ordering.

Multiset *See* bag.

Natural deduction Natural deduction is a particular style for presenting formal proofs in propositional and predicate calculus: inference rules for the introduction and elimination of each operator are given.

Negation A negation is a logical expression whose principal operator is 'not' $(\neg)$.

Non-determinism An operation whose specification permits more than one result for a particular argument is said to be non-deterministic.

Operation The term 'operation' is used for a program or piece thereof (often a procedure); an operation depends on and changes external variables (its state).

Operation decomposition Operations are decomposed into constructs which combine operations (e.g. **while** loops); proof obligations to check operation decomposition are given in Chapter 10.

Operation quotation Operations of one module (data type) can be used in the specifications of another module by quotation of pre- and post-conditions.

Operator Common functions are written as infix or prefix operators in order to shorten expressions and make the statement of algebraic properties clearer.

Partial function A partial function is one which is not defined for all of the values indicated in the domain part of its signature; the values to which it can be safely applied are defined by a pre-condition.

Partition A partition of a set S is a set of pairwise disjoint subsets of S whose union is S.

Post-condition The post-condition of a function or operation is a truth-valued function which defines the required relation between input and output.

Power set The power set of a set S is the set of all subsets of S.

Pre-condition The pre-condition of a function is a truth-valued function which defines the elements of the domain of a partial function (operation) for which the existence of a result is guaranteed. The pre-condition of an operation defines the state/inputs to which the operation can be applied.

Predicate A predicate is a truth-valued expression which may contain free variables.

Predicate calculus The expressions of the predicate calculus are built up from truth-valued functions, propositional operators and quantifiers.

Proof obligations Claims such as 'this piece of code satisfies that specification' give rise to proof obligations; if formal notation is used, these proof obligations are sequents to be proved.

Proof theory A proof theory for a calculus provides a way of deducing formulae; deductions begin with (instances of) axioms and use the given rules of inference.

Proper subset One set is a *proper* subset of another set if it is a subset and if the second set contains some elements absent from the first set.

Property oriented A property-oriented specification of a data type consists of a signature and a collection of equations.

Proposition An expression which, in classical logic, has the value true or false; in LPF, propositions can be undefined by virtue of undefined terms.

Propositional calculus The expressions of the propositional calculus are built up from propositions and the operators $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$; laws relate expressions and form a calculus.

Quantifiers Symbols of the predicate calculus: $\forall$ 'for all', $\exists$ 'there exists (one or more)', $\exists!$ 'there exists exactly one'.

Quoting The specification of one data type can be made to depend on the specification of another by quoting the pre- and post-conditions of its operators.

Range The range of a function is a specified set which contains the results of function application.

Recursive definition (abstract syntax) A recursively defined abstract syntax defines a class of finite, but arbitrarily deeply nested, objects by using the name of the class being defined within its definition.

Recursive definition (function) A recursive definition of a function is one in which the name of the function being defined is used within the definition.

Reflexivity A relation R is reflexive if, for all x, $(x, x) \in R$.

Reification Reification is the development of an abstract data type to a (more) concrete representation.

Relation A relation can be viewed as a subset of the Cartesian product of two sets. Many of the relations of interest in this book (e.g. equivalence relations) are such that the same set (X) constitutes the domain and range; such relations are said to be 'on X'.

Retrieve function A retrieve function relates a representation to an abstraction by mapping the former to the latter. Retrieve functions provide the basic link for data reification proofs.

Rigorous arguments A rigorous argument outlines how a proof could be constructed; the reason for accepting such an argument is the knowledge of how it could be made formal.

Satisfiability The use of implicit specification gives rise to a proof obligation known as satisfiability: for all acceptable inputs there must be some possible result.

Satisfy (specification) An implementation is said to satisfy a specification if, over the range of values required by the (pre-condition of the) specification, the implementation produces results which agree with the (post-condition of the) specification.

Satisfy (truth-valued function) Values satisfy a truth-valued function if its application to those values yields the value **true**.

Selectors The selectors for a composite type can be applied to values of that type to yield values of the components.

Semantics The semantics of a language are its meaning.

Sequence A sequence is an ordered collection of values in which values can occur more than once; elements are of a specified type and the sequence itself is of finite size.

Sequent A sequent consists of a list of logical expressions (the assumptions), followed by a turnstile, followed by another logical expression (its conclusion); it is to be read as a claim that, in all contexts where all of the assumptions are true, the conclusion can be deduced.

Set A set is an unordered collection of distinct objects.

Set comprehension A set can be defined by set comprehension to contain all elements satisfying some property.

Signature The signature of a function gives its domain and range.

Specification Strictly, a precise statement of all external characteristics of a system used here as a shorthand for 'functional specification'.

State A state is a collection of variables; the state of a state-based data type is such that the externals of all of its operations have compatible names and types with the state.

Structural induction Structural induction provides a way of generating induction rules for composite types.

Subset One set is a subset of another set if all of the elements of the first set are contained in the second. A set is thus a subset of itself.

Sufficiently abstract A model-oriented specification is said to be sufficiently abstract if it is not biased towards some particular implementations.

Symmetry A relation R is symmetric if, for all x and y, $(x, y) \in R \;\Rightarrow\; (y, x) \in R$.

Syntax *See* abstract syntax/concrete syntax.

Tautology A logical expression which evaluates to **true** for any values of its constituent propositions is a tautology.

Term A term is an expression involving constants, identifiers and operators; such a term denotes a value.

Transitivity A relation R is transitive if for all x, y and z, $(x, y) \in R \wedge (y, z) \in R \;\Rightarrow\; (x, z) \in R$.

Truth table A truth table is a tabular presentation of truth values which can be used either to define propositional operators or to verify facts about propositional expressions.

Truth-valued function A truth-valued function is one whose range is the truth values ($\mathbb{B}$).

Turnstile The turnstile ($\vdash$) symbol is used to record that the conclusion can be deduced from the hypotheses.

Union The union of two sets is the set containing exactly the elements contained in either (or both) sets.

Universal quantifier The universal quantifier ($\forall$) can be read as 'for all'.

VDM *See* Vienna Development Method.

Vienna Development Method (VDM) VDM is the name given to a collection of notation and concepts which grew out of the work of the IBM Laboratory, Vienna. The original application was the denotational description of programming languages. The same specification technique has been applied to many other systems. Design rules which show how to prove that a design satisfies its specification have been developed.

Well-founded A well-founded relation is one in which there are no infinite descending chains.

C

Rules of Logic

$$\boxed{\wedge\text{-}defn}\ \frac{\neg(\neg E_1 \vee \neg E_2)}{E_1 \wedge E_2}$$

$$\boxed{\Rightarrow\text{-}defn}\ \frac{\neg E_1 \vee E_2}{E_1 \;\Rightarrow\; E_2}$$

$$\boxed{\Leftrightarrow\text{-}defn}\ \frac{(E_1 \;\Rightarrow\; E_2) \wedge (E_2 \;\Rightarrow\; E_1)}{E_1 \;\Leftrightarrow\; E_2}$$

$$\boxed{\forall\text{-}defn}\ \frac{\neg(\exists x \in X \cdot \neg E(x))}{\forall x \in X \cdot E(x)}$$

$$\boxed{\vee\text{-}I}\ \frac{E_i}{E_1 \vee \cdots \vee E_n}\ 1 \le i \le n$$

$$\boxed{\wedge\text{-}I}\ \frac{E_1;\ \cdots;\ E_n}{E_1 \wedge \cdots \wedge E_n}$$

$$\boxed{\neg\vee\text{-}I}\ \frac{\neg E_1;\ \cdots;\ \neg E_n}{\neg(E_1 \vee \cdots \vee E_n)}$$

$$\boxed{\neg\wedge\text{-}I}\ \frac{\neg E_i}{\neg(E_1 \wedge \cdots \wedge E_n)}\ 1 \le i \le n$$

$$\boxed{\Rightarrow\text{-}I}\ \frac{E_1 \;\vdash\; E_2;\ \delta(E_1)}{E_1 \;\Rightarrow\; E_2}$$

$$\boxed{\Rightarrow vac\text{-}I}\ \frac{\neg\, E_1}{E_1 \ \Rightarrow \ E_2}$$

$$\boxed{\Rightarrow vac\text{-}I}\ \frac{E_2}{E_1 \ \Rightarrow \ E_2}$$

$$\boxed{\Leftrightarrow\text{-}I}\ \frac{E_1 \wedge E_2}{E_1 \ \Leftrightarrow \ E_2}$$

$$\boxed{\Leftrightarrow\text{-}I}\ \frac{\neg\, E_1 \wedge \neg\, E_2}{E_1 \ \Leftrightarrow \ E_2}$$

$$\boxed{\exists\text{-}I}\ \frac{s \in X;\ E(s/x)}{\exists x \in X \cdot E(x)}$$

$$\boxed{\neg\,\exists\text{-}I}\ \frac{x \in X \ \vdash \ \neg\, E(x)}{\neg\,(\exists x \in X \cdot E(x))}$$

$$\boxed{\forall\text{-}I}\ \frac{x \in X \ \vdash \ E(x)}{\forall x \in X \cdot E(x)}$$

$$\boxed{\neg\forall\text{-}I}\ \frac{s \in X;\ \neg\, E(s/x)}{\neg\,(\forall x \in X \cdot E(x))}$$

$$\boxed{\vee\text{-}E}\ \frac{E_1 \vee \cdots \vee E_n;\ E_1 \ \vdash \ E;\ \cdots;\ E_n \ \vdash \ E}{E}$$

$$\boxed{\wedge\text{-}E}\ \frac{E_1 \wedge \cdots \wedge E_n}{E_i}\ 1 \leq i \leq n$$

$$\boxed{\neg\,\vee\text{-}E}\ \frac{\neg\,(E_1 \vee \cdots \vee E_n)}{\neg\, E_i}\ 1 \leq i \leq n$$

$$\boxed{\neg\,\wedge\text{-}E}\ \frac{\neg\,(E_1 \wedge \cdots \wedge E_n);\ \neg\, E_1 \ \vdash \ E;\ \cdots;\ \neg\, E_n \ \vdash \ E}{E}$$

$$\boxed{\Rightarrow\text{-}E}\ \frac{E_1 \ \Rightarrow \ E_2;\ E_1}{E_2}$$

$$\boxed{\Leftrightarrow\text{-}E}\ \frac{E_1 \ \Leftrightarrow \ E_2}{E_1 \wedge E_2 \vee \neg E_1 \wedge \neg E_2}$$

$$\boxed{\neg\neg\text{-}I/E}\ \frac{E}{\neg\neg E}$$

$$\boxed{\exists\text{-}E}\ \frac{\exists x \in X \cdot E(x);\ \ y \in X, E(y/x)\ \vdash\ E_1}{E_1}\ y \text{ is arbitrary}$$

$$\boxed{\neg\exists\text{-}E}\ \frac{\neg(\exists x \in X \cdot E(x));\ \ s \in X}{\neg E(s/x)}$$

$$\boxed{\forall\text{-}E}\ \frac{\forall x \in X \cdot E(x);\ \ s \in X}{E(s/x)}$$

$$\boxed{\neg\forall\text{-}E}\ \frac{\neg(\forall x \in X \cdot E_1(x));\ \ y \in X, \neg E_1(y/x)\ \vdash\ E_2}{E_2}\ y \text{ is arbitrary}$$

$$\boxed{contr}\ \frac{E_1;\ \neg E_1}{E_2}$$

$$\boxed{\vee\text{-}comm}\ \frac{E_1 \vee E_2}{E_2 \vee E_1}$$

$$\boxed{\wedge\text{-}comm}\ \frac{E_1 \wedge E_2}{E_2 \wedge E_1}$$

$$\boxed{\vee\text{-}ass}\ \frac{(E_1 \vee E_2) \vee E_3}{E_1 \vee (E_2 \vee E_3)}$$

$$\boxed{\wedge\text{-}ass}\ \frac{E_1 \wedge (E_2 \wedge E_3)}{(E_1 \wedge E_2) \wedge E_3}$$

$$\boxed{\vee\text{-}subs}\ \frac{E_1 \vee \cdots \vee E_i \vee \cdots \vee E_n;\ \ E_i\ \vdash\ E}{E_1 \vee \cdots \vee E \vee \cdots \vee E_n}$$

$$\boxed{\wedge\text{-}subs}\ \frac{E_1 \wedge \cdots \wedge E_i \wedge \cdots \wedge E_n;\ \ E_i\ \vdash\ E}{E_1 \wedge \cdots \wedge E \wedge \cdots \wedge E_n}$$

$$\boxed{\vee\wedge\text{-}dist}\ \frac{E_1 \vee E_2 \wedge E_3}{(E_1 \vee E_2) \wedge (E_1 \vee E_3)}$$

$$\boxed{\wedge\vee\text{-}dist}\ \frac{E_1 \wedge (E_2 \vee E_3)}{E_1 \wedge E_2 \vee E_1 \wedge E_3}$$

$$\boxed{\vee\text{-}deM}\ \frac{\neg\,(E_1 \vee E_2)}{\neg\,E_1 \wedge \neg\,E_2}$$

$$\boxed{\wedge\text{-}deM}\ \frac{\neg\,(E_1 \wedge E_2)}{\neg\,E_1 \vee \neg\,E_2}$$

$$\boxed{\exists\text{-}deM}\ \frac{\neg\,(\exists x \in X \cdot E(x))}{\forall x \in X \cdot \neg\,E(x)}$$

$$\boxed{\forall\text{-}deM}\ \frac{\neg\,(\forall x \in X \cdot E(x))}{\exists x \in X \cdot \neg\,E(x)}$$

$$\boxed{\Rightarrow\text{-}contrp}\ \frac{E_1 \ \Rightarrow\ E_2}{\neg\,E_2 \ \Rightarrow\ \neg\,E_1}$$

$$\boxed{\text{L1.19}}\ \frac{E_1 \vee E_2 \ \Rightarrow\ E_3}{(E_1 \ \Rightarrow\ E_3) \wedge (E_2 \ \Rightarrow\ E_3)}$$

$$\boxed{\text{L1.20}}\ \frac{E_1 \ \Rightarrow\ (E_2 \ \Rightarrow\ E_3)}{E_1 \wedge E_2 \ \Rightarrow\ E_3}$$

D

Properties of Data

D.1 Natural numbers

$$\boxed{\mathbb{N}\text{-}ind}\ \frac{p(0);\quad n \in \mathbb{N},\ p(n) \vdash p(n+1)}{n \in \mathbb{N} \vdash p(n)}$$

$$\boxed{\mathbb{N}\text{-}indp}\ \frac{p(0);\quad n \in \mathbb{N}_1, p(n-1) \vdash p(n)}{n \in \mathbb{N} \vdash p(n)}$$

$$\boxed{\mathbb{N}\text{-}cind}\ \frac{n \in \mathbb{N},\ (\forall m \in \mathbb{N} \cdot m < n \Rightarrow p(m)) \vdash p(n)}{n \in \mathbb{N} \vdash p(n)}$$

D.2 Finite sets

$$\boxed{\odot\text{-}comm}\ \frac{e_1, e_2 \in X;\ s \in X\text{-}\mathbf{set}}{e_1 \odot (e_2 \odot s) = e_2 \odot (e_1 \odot s)}$$

$$\boxed{\odot\text{-}abs}\ \frac{e \in X;\ s \in X\text{-}\mathbf{set}}{e \odot (e \odot s) = e \odot s}$$

$$\boxed{Set\text{-}ind}\ \frac{p(\{\,\});\quad e \in X,\ s \in X\text{-}\mathbf{set},\ p(s) \vdash p(e \odot s)}{s \in X\text{-}\mathbf{set} \vdash p(s)}$$

$$\boxed{Set\text{-}ind2}\ \frac{p(\{\,\});\quad s \in X\text{-}\mathbf{set},\ e \in s,\ p(s - \{e\}) \vdash p(s)}{s \in X\text{-}\mathbf{set} \vdash p(s)}$$

$$\boxed{\cup\text{-}b}\ \frac{s \in X\text{-}\mathbf{set}}{\{\,\} \cup s = s}$$

$$\boxed{\cup\text{-}i}\ \frac{e \in X;\ s_1, s_2 \in X\text{-}\mathbf{set}}{(e \odot s_1) \cup s_2 = e \odot (s_1 \cup s_2)}$$

$$\boxed{\text{L4.5}}\ \frac{s \in X\text{-}\mathbf{set}}{s \cup \{\,\} = s}$$

$$\boxed{\cup\text{-}ass}\ \frac{s_1, s_2, s_3 \in X\text{-}\mathbf{set}}{(s_1 \cup s_2) \cup s_3 = s_1 \cup (s_2 \cup s_3)}$$

$$\boxed{\cup\text{-}comm}\ \frac{s_1, s_2 \in X\text{-}\mathbf{set}}{s_1 \cup s_2 = s_2 \cup s_1}$$

$$\boxed{\cup\text{-}idem}\ \frac{s \in X\text{-}\mathbf{set}}{s \cup s = s}$$

$$\boxed{\cap\text{-}b}\ \frac{s \in X\text{-}\mathbf{set}}{\{\,\} \cap s = \{\,\}}$$

$$\boxed{\cap\text{-}i}\ \frac{e \in X;\ s_1, s_2 \in X\text{-}\mathbf{set};\ e \in s_2}{(e \odot s_1) \cap s_2 = e \odot (s_1 \cap s_2)}$$

$$\boxed{\cap\text{-}i}\ \frac{e \in X;\ s_1, s_2 \in X\text{-}\mathbf{set};\ e \notin s_2}{(e \odot s_1) \cap s_2 = s_1 \cap s_2}$$

$$\boxed{\text{L??}}\ \frac{s \in X\text{-}\mathbf{set}}{s \cap \{\,\} = \{\,\}}$$

$$\boxed{\cap\text{-}ass}\ \frac{s_1, s_2, s_3 \in X\text{-}\mathbf{set}}{(s_1 \cap s_2) \cap s_3 = s_1 \cap (s_2 \cap s_3)}$$

$$\boxed{\bigcup\text{-}b}\ \frac{}{\bigcup\{\,\} = \{\,\}}$$

$$\boxed{\bigcup\text{-}i}\ \frac{s \in X\text{-}\mathbf{set},\ ss \in (X\text{-}\mathbf{set})\text{-}\mathbf{set}}{\bigcup(s \odot ss) = s \cup \bigcup ss}$$

$$\boxed{\text{L4.9}}\ \frac{ss_1, ss_2 \in (X\text{-}\mathbf{set})\text{-}\mathbf{set}}{\bigcup(ss_1 \cup ss_2) = \bigcup ss_1 \cup \bigcup ss_2}$$

$$\boxed{\text{d-b}}\ \frac{s \in X\text{-}\mathbf{set}}{\{\,\} - s = \{\,\}}$$

$$\boxed{\text{d-i}}\ \frac{e \in X;\ s_1, s_2 \in X\text{-}\mathbf{set};\ e \notin s_2}{(e \odot s_1) - s_2 = e \odot (s_1 - s_2)}$$

$$\boxed{\text{d-i}}\ \frac{e \in X;\ s_1, s_2 \in X\text{-}\mathbf{set};\ e \in s_2}{(e \odot s_1) - s_2 = s_1 - s_2}$$

$$\boxed{\in\text{-b}}\ \frac{}{\neg\,\exists e \in X \cdot e \in \{\,\}}$$

$$\boxed{\in\text{-i}}\ \frac{e_1, e_2 \in X,\ s \in X\text{-}\mathbf{set}}{e_1 \in (e_2 \odot s)\ \Leftrightarrow\ e_1 = e_2 \vee e_1 \in s}$$

D.3 Finite maps

$$\boxed{\odot\text{-}pri}\ \frac{d \in D;\ r_1, r_2 \in R;\ m \in D \xrightarrow{m} R}{\{d \mapsto r_1\} \odot (\{d \mapsto r_2\} \odot m) = \{d \mapsto r_1\} \odot m}$$

$$\boxed{\odot\text{-}comm}\ \frac{d_1, d_2 \in D;\ r_1, r_2 \in R;\ m \in D \xrightarrow{m} R;\ d_1 \neq d_2}{\{d_1 \mapsto r_1\} \odot (\{d_2 \mapsto r_2\} \odot m) = \{d_2 \mapsto r_2\} \odot (\{d_1 \mapsto r_1\} \odot m)}$$

$$\boxed{Map\text{-}ind}\ \frac{\begin{array}{l} p(\{\,\}); \\ d \in D,\ r \in R,\ m \in (D \xrightarrow{m} R),\ p(m),\ d \notin \mathbf{dom}\, m \vdash \\ \hfill p(\{d \mapsto r\} \odot m) \end{array}}{m \in (D \xrightarrow{m} R)\ \vdash\ p(m)}$$

$$\boxed{\dagger\text{-}b}\ \frac{m \in (D \xrightarrow{m} R)}{m \dagger \{\,\} = m}$$

$$\boxed{\dagger\text{-}i}\ \frac{d \in D;\ r \in R;\ m_1, m_2 \in (D \xrightarrow{m} R)}{m_1 \dagger (\{d \mapsto r\} \odot m_2) = \{d \mapsto r\} \odot (m_1 \dagger m_2)}$$

$$\boxed{\text{L6.6}}\ \frac{m \in (D \xrightarrow{m} R)}{\{\,\} \dagger m = m}$$

$$\boxed{\dagger\text{-}ass}\ \frac{m_1, m_2, m_3 \in (D \xrightarrow{m} R)}{m_1 \dagger (m_2 \dagger m_3) = (m_1 \dagger m_2) \dagger m_3}$$

$$\boxed{\mathbf{dom}\text{-}b}\ \frac{}{\mathbf{dom}\,\{\,\} = \{\,\}}$$

$$\boxed{\mathbf{dom}\text{-}i}\ \frac{d \in D;\ r \in R;\ m \in (D \xrightarrow{m} R)}{\mathbf{dom}\,(\{d \mapsto r\} \odot m) = \{d\} \cup \mathbf{dom}\, m}$$

$$\boxed{\lhd\text{-}b}\ \frac{s \in D\text{-}\mathbf{set}}{s \lhd \{\,\} = \{\,\}}$$

$$\boxed{\lhd\text{-}i}\ \frac{s \in D\text{-}\mathbf{set};\ m \in (D \xrightarrow{m} R);\ d \in D;\ r \in R;\ d \notin s}{s \lhd (\{d \mapsto r\} \odot m) = s \lhd m}$$

$$\boxed{\lhd\text{-}i}\ \frac{s \in D\text{-}\mathbf{set};\ m \in (D \xrightarrow{m} R);\ d \in D;\ r \in R;\ d \in s}{s \lhd (\{d \mapsto r\} \odot m) = \{d \mapsto r\} \odot (s \lhd m)}$$

$$\boxed{⩤\text{-}b}\ \frac{s \in D\text{-}\mathbf{set}}{s ⩤ \{\,\} = \{\,\}}$$

$$\boxed{⩤\text{-}i}\ \frac{s \in D\text{-}\mathbf{set};\ m \in (D \xrightarrow{m} R);\ d \in D;\ r \in R;\ d \notin s}{s ⩤ (\{d \mapsto r\} \odot m) = \{d \mapsto r\} \odot (s ⩤ m)}$$

$$\boxed{⩤\text{-}i}\ \frac{s \in D\text{-}\mathbf{set};\ m \in (D \xrightarrow{m} R);\ d \in D;\ r \in R;\ d \in s}{s ⩤ (\{d \mapsto r\} \odot m) = s ⩤ m}$$

$$\boxed{\cup\text{-}b}\ \frac{m \in (D \xrightarrow{m} R)}{\{\,\} \cup m = m}$$

$$\boxed{\cup\text{-}i}\ \frac{m_1, m_2 \in (D \xrightarrow{m} R);\ d \in D;\ r \in R;\ is\text{-}disj\,(\{d\} \cup \mathbf{dom}\, m_1, \mathbf{dom}\, m_2)}{(\{d \mapsto r\} \odot m_1) \cup m_2 = \{d \mapsto r\} \odot (m_1 \cup m_2)}$$

$$\boxed{\text{L6.11}}\ \frac{m_1, m_2 \in (D \xrightarrow{m} R)}{\mathbf{dom}\,(m_1 \dagger m_2) = \mathbf{dom}\, m_1 \cup \mathbf{dom}\, m_2}$$

$$\boxed{\cup_m\text{-}ass}\ \frac{m_1, m_2, m_3 \in (D \xrightarrow{m} R)}{(m_1 \cup m_2) \cup m_3 = m_1 \cup (m_2 \cup m_3)}$$

$$\boxed{\cup_m\text{-}comm}\ \frac{m_1, m_2 \in (D \xrightarrow{m} R)}{m_1 \cup m_2 = m_2 \cup m_1}$$

$$\boxed{\text{L6.8}}\ \frac{m_1, m_2 \in (D \xrightarrow{m} R);\ \ is\text{-}disj(\mathbf{dom}\, m_1, \mathbf{dom}\, m_2)}{m_1 \dagger m_2 = m_1 \cup m_2}$$

D.4 Finite sequences

$$\boxed{Seq\text{-}ind}\ \frac{\begin{array}{l} p([\,]); \\ e \in X,\ t \in X^*,\ p(t) \vdash p(cons(e,t)) \end{array}}{t \in X^* \vdash p(t)}$$

$$\boxed{Seq\text{-}ind2}\ \frac{\begin{array}{l} p([\,]); \\ t \in X^+,\ p(\mathbf{tl}\, t) \vdash p(t) \end{array}}{t \in X^* \vdash p(t)}$$

$$\boxed{\curvearrowright\text{-}b}\ \frac{s \in X^*}{[\,] \curvearrowright s = s}$$

$$\boxed{\curvearrowright\text{-}i}\ \frac{e \in X;\ s_1, s_2 \in X^*}{cons(e, s_1) \curvearrowright s_2 = cons(e, s_1 \curvearrowright s_2)}$$

$$\boxed{\curvearrowright\text{-}ass}\ \frac{s_1, s_2, s_3 \in X^*}{(s_1 \curvearrowright s_2) \curvearrowright s_3 = s_1 \curvearrowright (s_2 \curvearrowright s_3)}$$

$$\boxed{\mathbf{len}\text{-}b}\ \frac{}{\mathbf{len}\,[\,] = 0}$$

$$\boxed{\mathbf{len}\text{-}i}\ \frac{e \in X, s \in X^*}{\mathbf{len}\ cons(e, s) = \mathbf{len}\, s + 1}$$

$$\boxed{\text{L7.6}}\ \frac{s_1, s_2 \in X^*}{\mathbf{elems}\,(s_1 \curvearrowright s_2) = (\mathbf{elems}\, s_1) \cup (\mathbf{elems}\, s_2)}$$

$$\boxed{\textit{rev-b}}\ \frac{}{rev([\,]) = [\,]}$$

$$\boxed{\textit{rev-i}}\ \frac{e \in X;\ s \in X^*}{rev(cons(e, s)) = rev(s) \frown [e]}$$

$$\boxed{\text{L7.11}}\ \frac{s \in X^*}{rev(rev(s)) = s}$$

E

Proof Obligations

E.1 Satisfiability

Functions:

$$\forall d \in D \cdot \mathit{pre\text{-}f}(d) \;\Rightarrow\; \exists r \in R \cdot \mathit{post\text{-}f}(d, r)$$

Operations:

$$\forall \overleftarrow{\sigma} \in \Sigma \cdot \mathit{pre\text{-}OP}(\overleftarrow{\sigma}) \;\Rightarrow\; \exists \sigma \in \Sigma \cdot \mathit{post\text{-}OP}(\overleftarrow{\sigma}, \sigma)$$

Remember the role of invariants in such proofs.

E.2 Satisfaction of specification

Functions:

$$\forall d \in D \cdot \mathit{pre\text{-}f}(d) \;\Rightarrow\; f(d) \in R \wedge \mathit{post\text{-}f}(d, f(d))$$

Operations:

$$\begin{array}{l}
\forall \overleftarrow{\sigma} \in \Sigma \cdot \\
\qquad \mathit{pre\text{-}OP}(\overleftarrow{\sigma}) \;\Rightarrow \\
\qquad\qquad (\exists \sigma \in \Sigma \cdot (\overleftarrow{\sigma}, \sigma) \in OP) \wedge \\
\qquad\qquad\quad (\forall \sigma \in \Sigma \cdot (\overleftarrow{\sigma}, \sigma) \in OP \;\Rightarrow\; \mathit{post\text{-}OP}(\overleftarrow{\sigma}, \sigma))
\end{array}$$

E.3 Data reification

Adequacy:

$$\forall a \in A \cdot \exists r \in R \cdot \mathit{retr}(r) = a$$

Initial state:

$$\mathit{retr}(r_0) = a_0$$

Domain:

$$\forall r \in R \cdot \mathit{pre}\text{-}A(\mathit{retr}(r)) \;\Rightarrow\; \mathit{pre}\text{-}R(r)$$

Result:

$$\begin{array}{l} \forall \overleftarrow{r}, r \in R \cdot \\ \qquad \mathit{pre}\text{-}A(\mathit{retr}(\overleftarrow{r})) \wedge \mathit{post}\text{-}R(\overleftarrow{r}, r) \;\Rightarrow\; \mathit{post}\text{-}A(\mathit{retr}(\overleftarrow{r}), \mathit{retr}(r)) \end{array}$$

E.4 Operation decomposition

$$\boxed{:=\text{-}I}\ \frac{}{\{\mathbf{true}\}\ x := e\ \{x = \overleftarrow{e}\}}$$

$$\boxed{:=\text{-}\mathit{pres}}\ \frac{}{\{E\}\ x := e\ \{E\}}\ x \text{ does not occur free in } E$$

$$\boxed{;\text{-}I}\ \frac{\{\mathit{pre}_1\}\ S_1\ \{\mathit{pre}_2 \wedge \mathit{post}_1\};\ \{\mathit{pre}_2\}\ S_2\ \{\mathit{post}_2\}}{\{\mathit{pre}_1\}\ (S_1; S_2)\ \{\mathit{post}_1 \mid \mathit{post}_2\}}$$

$$\boxed{\mathbf{if}\text{-}I}\ \frac{\{\mathit{pre} \wedge \mathit{test}\}\ \mathit{TH}\ \{\mathit{post}\};\ \{\mathit{pre} \wedge \neg\, \mathit{test}\}\ \mathit{EL}\ \{\mathit{post}\};\ \mathit{pre} \;\Rightarrow\; \delta_l(\mathit{test})}{\{\mathit{pre}\}\ (\mathbf{if}\ \mathit{test}\ \mathbf{then}\ \mathit{TH}\ \mathbf{else}\ \mathit{EL})\ \{\mathit{post}\}}$$

$$\boxed{\mathbf{while}\text{-}I}\ \frac{\{\mathit{inv} \wedge \mathit{test}\}\ S\ \{\mathit{inv} \wedge \mathit{sofar}\};\ \mathit{inv} \;\Rightarrow\; \delta_l(\mathit{test})}{\{\mathit{inv}\}\ \mathbf{while}\ \mathit{test}\ \mathbf{do}\ S\ \mathbf{end}\ \{\mathit{inv} \wedge \neg\, \mathit{test} \wedge (\mathit{sofar} \vee \mathit{iden})\}}\ \mathit{sofar} \text{ is twf}$$

$$\boxed{\mathbf{block}\text{-}I}\ \frac{\{\mathit{pre} \wedge v = e\}\ S\ \{\mathit{post}\}}{\{\mathit{pre}\}\ \mathbf{begin}\ \ \mathbf{var}\ v := e; S\ \mathbf{end}\ \{\exists v \cdot \mathit{post}\}}$$

$$\boxed{\mathit{weaken}}\ \frac{\mathit{pre}_s \;\Rightarrow\; \mathit{pre};\ \{\mathit{pre}\}\ S\ \{\mathit{post}\};\ \mathit{post} \;\Rightarrow\; \mathit{post}_w}{\{\mathit{pre}_s\}\ S\ \{\mathit{post}_w\}}$$

$$\boxed{\mathit{pre}}\ \frac{\{\mathit{pre}\}\ S\ \{\mathit{post}\}}{\{\mathit{pre}\}\ S\ \{\overleftarrow{\mathit{pre}} \wedge \mathit{post}\}}$$

F

Syntax of VDM Specifications

This appendix contains parts of the 'Mathematical Syntax' for those parts of VDM used in this book. It is derived from [BSI89] but, since that document is still evolving, some predictions as to its final form have been made. In some cases, alternatives present in [BSI89] have been removed because they are not used in this book.

The proposed concrete representation for the BSI VDM specification language is defined by a context-free grammar which conforms to the BSI standard for grammars which is described by means of a BNF notation which employs the following special symbols:

,	the concatenate symbol
=	the define symbol
|	the definition separator symbol (lower precedence than concatenate)
[]	enclose optional syntactic items
{ }	enclose syntactic items which may occur zero or more times
' '	single quotes are used to enclose terminal symbols
meta identifier	non-terminal symbols are written in lower-case letters (possibly including spaces)
;	terminator symbol to denote the end of a rule
..	used (within brackets) to describe a range of terminal symbols, e.g. ('a'..'z','A'..'Z'). Note that ',' in this context means 'and', not 'concatenate'.

F.1 Documents

document = modules
| definitions ;

modules = module, { module } ;

F.2 Modules

module = ‘**module**’, identifier, interface, definitions, ‘**end**’, identifier ;

F.3 Interfaces

interface = [module parameters],
[import definition list],
[instantiation instance list],
[export module signature] ;

module parameters = ‘**parameters**’, module signature ;

import definition list = ‘**imports**’, import definition, { ‘,’, import definition } ;

import definition = ‘**from**’, identifier, ‘:’, module signature ;

instantiation instance list = ‘**instantiation**’, instantiation instance,
{ ‘,’, instantiation instance } ;

instantiation instance = identifier, ‘**as**’, instance ;

export module signature = ‘**exports**’, module signature ;

module signature = { signatures } ;

signatures = type signatures
| value signatures
| function signatures
| operation signatures ;

type signatures = ‘**types**’, type description, { ‘,’, type description } ;

type description = name
| type definition ;

value signatures = ‘**values**’, value description, { ‘,’, value description } ;

value description = name list, ‘:’, type ;

function signatures = ‘**functions**’, function signature,
{ ‘,’, function signature } ;

function signature = name list, ‘:’, function type ;

operation signatures = ‘**operations**’, operation signature,
{ ‘,’, operation signature } ;

operation signature = name list, ‘:’, operation type, [‘**using**’, name] ;

instance = identifier, ‘(’, [substitution], ‘)’, module signature ;

substitution = substitute, { ‘,’, substitute } ;

substitute = identifier, ‘→’, name ;

F.4 Definitions

definitions = [‘**definitions**’, definition block, { [‘;’], definition block }] ;

definition block = type definitions
| state definition
| value definitions
| function definitions
| operation definitions ;

Type definitions

type definitions = ‘**types**’, type definition, { [‘;’], type definition } ;

type definition = identifier, ‘=’, type, [invariant]
| identifier, ‘::’, field list, [invariant]
| identifier, is not yet defined ;

type = bracketed type
| type name
| basic type
| quote type

| composite type
| union type
| set type
| seq type
| map type
| function type
| optional type
| product type
| type variable ;

bracketed type = '(', type, ')' ;

type name = name ;

basic type = '$\mathbb{B}$' | '$\mathbb{N}$' | '$\mathbb{N}_1$' | '$\mathbb{Z}$' | '$\mathbb{R}$' ;

quote type = quote literal ;

composite type = '**compose**', identifier, '**of**', field list, '**end**' ;

field list = field, { field } ;

field = [identifier, ':'], type ;

union type = type, '|', type ;

set type = type, '-set' ;

seq type = seq0 type
| seq1 type ;

seq0 type = type, '*' ;

seq1 type = type, '+' ;

map type = general map type
| bijective map type ;

general map type = type, '$\xrightarrow{m}$', type ;

bijective map type = type, '$\xleftrightarrow{m}$', type ;

function type = type, '→', type
| '()', '→', type ;

optional type = '[', type, ']' ;

product type = type, '×', type ;

type variable = '@', identifier ;

is not yet defined = '**is**', '**not**', '**yet**', '**defined**' ;

State definitions

state definition = '**state**', identifier, '**of**', field list,
[invariant], [initialization], '**end**' ;

invariant = '**inv**', invariant initial function ;

initialization = '**init**', invariant initial function ;

invariant initial function = pattern, '$\triangleq$', expression ;

Value definitions

value definitions = '**values**', value definition, { [';'], value definition } ;

value definition = identifier, ['=', expression], [':', type] ;

Function definitions

function definitions = '**functions**', function definition,
{ [';'], function definition } ;

function definition = function heading, function body ;

function heading = function signature heading
| function colon heading ;

function signature heading = identifier, ':', function type,
identifier, parameter list, [identifier] ;

function colon heading = identifier, parameter type list, [identifier type pair] ;

parameter type list = parameter types, { parameter types } ;

identifier type pair = [identifier, ':'], type ;

parameter types = '(', [pattern type pair list], ')' ;

pattern type pair list = pattern list, ':', type, { ',', pattern list,':', type } ;

parameter list = parameters, { parameters } ;

parameters = '(', [pattern list], ')' ;

function body = explicit function
| function post ;

explicit function = '$\triangleq$', expression, ['**pre**', expression] ;

function post = ['**pre**', expression], '**post**', expression ;

Operation definitions

operation definitions = '**operations**', operation definition,
{ [';'], operation definition } ;

operation definition = operation heading, operation body ;

operation heading = operation signature heading
| operation colon heading ;

operation signature heading = identifier, ':', operation type, identifier,
parameter list, [identifier] ;

operation type = type, '$\xrightarrow{o}$', [type]
| '()', '$\xrightarrow{o}$', [type] ;

operation colon heading = identifier, parameter type list,
[identifier type pair], externals ;

operation body = operation post ;

operation post = ['**pre**', expression], '**post**', expression, [exceptions] ;

externals = external, var information, { var information } ;

external = '**external**' | '**ext**' ;

var information = mode, state name list, ':', type ;

mode = '**read**' | '**write**' | '**rd**' | '**wr**' ;

state name list = name, { ',', name } ;

exceptions = errors, error list ;

errors = '**errs**' | '**errors**' ;

error list = error, { error } ;

error = identifier, ':', expression, '→', expression ;

F.5 Expressions

expression list = expression, { ',', expression } ;

expression = complex expression
| unary expression
| binary expression
| general quantified expression
| iota expression
| set expression
| sequence expression
| map expression
| record expression
| apply expression
| simple expression
| literal
| names ;

Complex expressions

complex expression = let expression
| if expression
| cases expression ;

let expression = '**let**', equal definition list, '**in**', expression ;

if expression = ‘**if**’, expression, ‘**then**’, expression, { elsif expression },
‘**else**’, expression ;

elsif expression = ‘**elseif**’, expression, ‘**then**’, expression ;

cases expression = ‘**cases**’, expression, ‘:’, cases expression alternatives,
[‘,’, others expression], ‘**end**’ ;

cases expression alternatives = cases expression alternative,
{ ‘,’, cases expression alternative } ;

cases expression alternative = case pattern, ‘$\rightarrow$’,expression ;

others expression = ‘**others**’, ‘$\rightarrow$’, expression ;

Unary expressions

unary expression = prefix expression | map inverse ;

prefix expression = unary operator, expression ;

unary operator = ‘+’
| ‘$-$’
| ‘**abs**’
| ‘**floor**’
| ‘$\neg$’
| ‘**card**’
| ‘$\bigcup$’
| ‘**hd**’
| ‘**tl**’
| ‘**len**’
| ‘**elems**’
| ‘**inds**’
| ‘**conc**’
| ‘**dom**’
| ‘**rng**’ ;

map inverse = expression, ‘$^{-1}$’ ;

Binary expressions

binary expression = infix expression | set range expression ;

infix expression = expression, binary operator, expression ;

binary operator = ‘$+$’
| ‘$-$’
| ‘$*$’
| ‘$/$’
| ‘**rem**’
| ‘**mod**’
| ‘$\uparrow$’
| ‘**div**’
| ‘$\cup$’
| ‘$\cap$’
| ‘$-$’
| ‘$\subseteq$’
| ‘$\subset$’
| ‘$\in$’
| ‘$\notin$’
| ‘$\frown$’
| ‘$\cup$’
| ‘$\dagger$’
| ‘$\lhd$’
| ‘$⩤$’
| ‘$\rhd$’
| ‘$⩥$’
| ‘$\wedge$’
| ‘$\vee$’
| ‘ $\Rightarrow$ ’
| ‘ $\Leftrightarrow$ ’
| ‘$=$’
| ‘$\neq$’
| ‘$<$’
| ‘$\leq$’
| ‘$>$’
| ‘$\geq$’ ;

set range expression = ‘{’, expression, ‘,’, ‘. . . ’, ‘,’, expression, ‘}’ ;

Quantified expressions

general quantified expression = quantified expression
| exists unique expression ;

quantified expression = quantifier, bind list, '$\cdot$', expression ;

quantifier = '$\forall$' | '$\exists$' ;

bind list = bind, { ',', bind } ;

exists unique expression = '$\exists!$', bind, '$\cdot$', expression ;

bind = set bind ;

set bind = pattern, '$\in$', expression ;

iota expression = 'ι', bind, '$\cdot$', expression ;

Set expressions

set expression = set enumeration
| set comprehension ;

set enumeration = '{}' | '{', expression list, '}' ;

set comprehension = '{', bind, '|', expression, '}' ;

Sequence expressions

sequence expression = sequence enumeration
| sequence comprehension
| subsequence ;

sequence enumeration = '[]' | '[', expression list, ']' ;

sequence comprehension = '[', sequence apply, '$\in$', expression, '|',
expression, ']' ;

subsequence = expression, '(', expression, ',', '$\ldots$', ',', expression, ')' ;

Map expressions

map expression = map enumeration
| map comprehension ;

map enumeration = '{}' | '{', maplet list, '}' ;

maplet list = maplet, { ',', maplet } ;

maplet = expression, '$\mapsto$', expression ;

map comprehension = '{', maplet, '$\in$', expression, '|', expression, '}' ;

Record expressions

record expression = record constructor
| record modifier ;

record constructor = 'mk-', name, '(', [expression list], ')' ;

record modifier = 'μ', '(', expression, record modification,
{ record modification }, ')' ;

record modification = identifier, '$\mapsto$', expression ;

Apply expressions

apply expression = function apply
| sequence apply
| map apply
| field select ;

function apply = expression, '(', [expression list], ')' ;

sequence apply = expression, '(', expression, ')' ;

map apply = expression, '(', expression list, ')' ;

field select = identifier, '(', expression, ')' ;

Simple expressions

simple expression = bracketed expression ;

bracketed expression = '(', expression, ')' ;

F.6 Names

names = name | oldname ;

name list = name, { ‘,’, name } ;

name = { identifier, ‘.’ }, identifier ;

old name = $\overleftarrow{\text{identifier}}$;

identifier = mark, { mark | digit | prime | hyphen } ;

mark = letter | greek ;

prime = ‘′’ ;

hyphen = ‘-’ ;

greek = (‘α’ .. ‘Ω’) ;

(* Any component of an identifier except the first mark can also be either a subscript or a superscript. *)

F.7 Literals

literal = undefined literal
| nil literal
| Boolean literal
| numeral
| character literal
| text literal
| quote literal ;

undefined literal = ‘**undefined**’ ;

nil literal = ‘**nil**’ ;

Boolean literal = ‘**true**’ | ‘**false**’ ;

numeral = natural number, [‘$\times$10’, integer literal (* as a superscript *)] ;

integer literal = [‘+’ | ‘$-$’], natural number ;

natural number = digit, { digit } ;

digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;

character literal = '", char, '" ;

text literal = '"', meta string, '"' ;

meta string = { char } ;

char = '""' | character − ('"') ;

quote literal = distinguished letter, { distinguished letter } ;

upper case letter = ('A' .. 'Z') ;

lower case letter = ('a' .. 'z') ;

distinguished letter = ('A' .. 'Z') ;

odd character = (* to be defined *) ;

character = upper case letter | lower case letter | digit | odd character ;

letter = upper case letter | lower case letter ;

F.8 Patterns

pattern list = pattern, { ',', pattern } ;

pattern = pattern identifier
| match value
| record pattern ;

pattern identifier = identifier | '-' ;

match value = '(', expression, ')' ;

record pattern = [name], '(', [pattern list], ')' ;

F.9 Comments

comments = brief comment ;

brief comment = '--', character, { character }, new line character ;

Bibliography

[BBH$^+$74] H. Bekič, D. Bjørner, W. Henhapl, C.B. Jones, and P. Lucas. A formal definition of a PL/I subset. Technical Report 25.139, IBM Laboratory, Vienna, 1974.

[BM79] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979.

[BSI89] BSI. *VDM Specification Language: Proto-Standard*, 1989. IST/5/50.

[Dij76] E.W. Dijkstra. *A Discipline of Programming.* Prentice Hall, 1976.

[Dro87] R.G. Dromey. Derivation of sorting algorithms from a specification. *The Computer Journal*, 30(6):512–518, 1987.

[Hay87] I. Hayes, editor. *Specification Case Studies.* Prentice Hall International, 1987.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.

[HS72] P. Henderson and R. Snowdon. An experiment in structured programming. *BIT*, 12, 1972.

[Jac83] M.A. Jackson. *System Development.* Prentice Hall International, 1983.

[JL88] C.B. Jones and P.A. Lindsay. A support system for formal reasoning: Requirements and status. In R. Bloomfield, L. Marshall, and R. Jones, editors, *VDM'88: VDM—The Way Ahead*, pages 139–152. Springer-Verlag, 1988. Lecture Notes in Computer Science, Vol. 328.

[Jon79] C.B. Jones. Constructing a theory of a data structure as an aid to program development. *Acta Informatica*, 11:119–137, 1979.

[Jon80] C.B. Jones. *Software Development: A Rigorous Approach.* Prentice Hall International, Englewood Cliffs, NJ, 1980.

[Jon83] C.B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP '83*, pages 321–332. North-Holland, 1983.

[JS90] C.B. Jones and R.C.F. Shaw, editors. *Case Studies in Systematic Software Development*. Prentice Hall International, 1990.

[Lin88] P.A. Lindsay. A survey of mechanical support for formal reasoning. *Software Engineering Journal*, 3(1), 1988.

[Luc68] P. Lucas. Two constructive realizations of the block concept and their equivalence. Technical Report TR 25.085, IBM Laboratory, Vienna, 1968.

[MJ84] F.L. Morris and C.B. Jones. An early program proof by Alan Turing. *Annals of the History of Computing*, 6(2):139–143, 1984.

[RT89] B. Ritchie and P. Taylor. The interactive proof editor: An experiment in interactive theorem proving. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer-Verlag, 1989.

[Wei75] F. Weissenböck. A formal interface specification. Technical Report TR 25.141, IBM Laboratory, Vienna, 1975.

Index of Functions and Operations

Index of Types

General Index