Chapter 2

Relations

Our treatment of the logic of engineering design has been sparsely illustrated by examples of water tanks and of computer programs, but it is applicable in principle to any branch of technology. We have allowed specifications and products to be formalised as predicates with arbitrary alphabets, and these predicates may be assembled by an arbitrary combination of the operators of the predicate calculus, including conjunction and even negation. Indeed, at the level of specifications, negation could be the most important contribution to safety of a control system; for example, a common safety requirement can be simply and most reliably expressed by stating that

"it must *not* explode".

In this chapter we begin to specialise our study to the particular classes of predicate that are relevant to the specification and development of programs in a simple sequential programming language. For a program in such a language, the relevant observations come in pairs, with one observation of the values of all global variables before program execution, and one observation of their values after termination. Any set of such pairs therefore constitutes a *relation* in the usual sense, and this is the word we choose to apply to the corresponding predicate describing program behaviour. The relational calculus provides most of the concepts and laws that we need both for sequential programming and for all the more complex programming paradigms treated in later chapters. We deliberately omit both relational converse ($^\cup$) and negation ($\neg$) because they are not directly implementable. The inverse of an operation cannot be computed by running a program backwards, and an explosion cannot be prevented by just negating a program which deliberately causes one.

Definition 2.0.1 (Relation)

A relation is a pair $(\alpha P, P)$, where P is a predicate containing no free variables

other than those in αP, and

$$\alpha P \;=\; in\alpha P \cup out\alpha P$$

where $in\alpha P$ is a set of undashed variables standing for initial values, and $out\alpha P$ is a set of dashed variables standing for final values. □

In all familiar programming languages, the global variables observable at the start of execution of a block of program are the same as those observable at the end. In this case the output alphabet is obtained just by putting a dash on all the variables of the input alphabet, as suggested by the equation

$$out\alpha P \;=\; in\alpha' P$$

The relation is then called *homogeneous*. The classical relational calculus deals solely with homogeneous relations. Our treatment is more general, although we will not need to make explicit mention of non-homogeneous relations again until Section 9 of this chapter.

Having formally associated an alphabet with every predicate, we will adopt a number of conventions to prevent the formality from becoming too obtrusive. For example, an important class of predicates are those with no variables in their output alphabet. They are the only kind of predicate that can actually appear in a computer program: they can be tested in the initial state of the program, and will yield the value *true* or *false*. (For simplicity, we assume they never fail to terminate.) We call them *conditions*, and denote them by lower case italics

$$b, c, \ldots, \mathit{true}$$

This distinguishes them from more general predicates describing programs, which generally mention both dashed and undashed variables. These will be denoted by upper case or bold

$$P, Q, \ldots, \mathbf{true}$$

Similar remarks apply to expressions which appear on the right of assignments in a program: they have no dashed variables, their evaluation always terminates, and they are denoted by

$$e, f, g, \ldots, 37$$

These letters can also stand for *lists* of expressions. We will use the letters x, y, z, v to stand for lists of *distinct* variables in the input alphabets, and their variants x', y', z', v' to stand for the corresponding lists of dashed variables. So the form

$$x := e$$

could stand for a multiple assignment of many values to many variables, and it

will be implicitly assumed that the lists are of matching length. The lists v and v' are special: they are assumed to contain *all* the variables of the input and output alphabets respectively.

Substitution will also be treated quite informally. Whenever appropriate, some or all of the free variables of a predicate, term or condition will be made explicit in brackets

$$P(x), Q(x'), e(x) \text{ or } b(x)$$

If f is now a list of expressions (of the right length, of course) then

$$P(f), Q(f), e(f) \text{ or } b(f)$$

stands for the result of replacing each occurrence of any variable in x (or x') by the expression which occupies the corresponding position in the list f. Implicitly, we ensure that there is no collision between free variables of f and bound variables in the context where f is being substituted. Sometimes we will use an explicit notation for substitution. $P[e/x]$ is the result of substituting expressions e for variables x in P.

The specialisation of the class of predicates to those which are relations is motivated by the desire to develop a theory which is relevant to computer programming. The primitive components of our programs are assignments, and these were described in the last chapter. In this chapter we will describe how larger programs are built up from the primitives by combining them in sequential compositions, conditionals and disjunctions. These operators have been selected for ease and efficiency of implementation on a sequential computer. Finally, recursion is introduced as a general method of specifying how portions of a program can be repeated, with the number of repetitions being determined by the initial values of the variables.

In the normal fashion of mathematics, each definition is followed by a collection of simple theorems that can be proved from it. These laws are intended to appeal to experienced programmers as obviously true of a tightly disciplined programming language in which all expressions terminate and none of them has side-effects; such an appeal is essential to the memorisation, recall and fluent use of the laws. It also lends credibility to the definitions on which they are based. Conversely, an obviously false law will serve like an unsuccessful scientific experiment: it will refute the theory from which it has been derived.

2.1 Conditional

Any non-trivial program requires a facility to select between alternative actions in accordance with the truth or falsity of some testable condition b. The restriction that b contains no dashed variables ensures that it can be tested before starting either of the actions. If P and Q are predicates describing two fragments of program with the same alphabet, then the conditional

$$P \lhd b \rhd Q$$

describes a program which behaves like P if the initial value of b is true, or like Q if the initial value of b is false. It can be defined as a simple truth function in the propositional calculus.

Definition 2.1.1 (Conditional)

$$\begin{aligned} P \lhd b \rhd Q &=_{df} (b \wedge P) \vee (\neg b \wedge Q), \text{ if } \alpha b \subseteq \alpha P = \alpha Q \\ \alpha(P \lhd b \rhd Q) &=_{df} \alpha P \end{aligned}$$ □

The more usual notation for a conditional is

if b **then** P **else** Q

We have chosen an infix notation $\lhd b \rhd$ because it simplifies expression of the relevant algebraic laws.

L1 $P \lhd b \rhd P = P$ (cond idemp)

L2 $P \lhd b \rhd Q = Q \lhd \neg b \rhd P$ (cond symm)

L3 $(P \lhd b \rhd Q) \lhd c \rhd R = P \lhd b \wedge c \rhd (Q \lhd c \rhd R)$ (cond assoc)

L4 $P \lhd b \rhd (Q \lhd c \rhd R) = (P \lhd b \rhd Q) \lhd c \rhd (P \lhd b \rhd R)$ (cond distr)

L5 $P \lhd true \rhd Q = P = Q \lhd false \rhd P$ (cond unit)

Proof These laws can be proved by propositional calculus; the easiest way is to consider separately the case when b is true and when b is false. □

The laws have been named by descriptions of familiar algebraic properties. A binary operator is *idempotent* when application to two identical operands gives a result equal to both of them, as shown in the first law. An operator is *symmetric* when an exchange of its two operands does not change the result. However, in the case of **L2**, the operator also has to be changed by negating the condition, so the symmetry is *skewed.* An *associative* operator is one that permits rearrangement of its brackets when applied twice. **L3** gives a slightly stronger form of this property for the conditional. The fourth law states the distribution of any conditional choice operator $\lhd b \rhd$ through the conditional $\lhd c \rhd$, for any condition c. The final law clearly expresses the criterion for making a choice between two alternatives of a

conditional based on the value of the condition. It is called a *unit* law, because it indicates the operand (the unit) for which the application of the operator makes no difference. These laws are strong enough to prove additional useful laws, as shown below.

L6 $P \lhd b \rhd (Q \lhd b \rhd R) = P \lhd b \rhd R$

Proof

$$\begin{aligned} & LHS && \{\textbf{L2}\} \\ = \; & (Q \lhd b \rhd R) \lhd \neg b \rhd P && \{\textbf{L3}\} \\ = \; & Q \lhd \mathit{false} \rhd (R \lhd \neg b \rhd P) && \{\textbf{L2} \text{ and } \textbf{L5}\} \\ = \; & RHS && \Box \end{aligned}$$

L7 $P \lhd b \rhd (P \lhd c \rhd Q) = P \lhd b \vee c \rhd Q$

Proof

$$\begin{aligned} & LHS && \{\textbf{L2}\} \\ = \; & (Q \lhd \neg c \rhd P) \lhd \neg b \rhd P && \{\textbf{L3} \text{ and } \textbf{L1}\} \\ = \; & Q \lhd \neg c \wedge \neg b \rhd P && \{\textbf{L2}\} \\ = \; & RHS && \Box \end{aligned}$$

These laws would be just as easy to prove directly by the propositional calculus. The algebraic style of proof has been preferred, because it reveals something of the structure of our theory, with only a few laws taken as basic, and many more derivable as theorems. This has the advantage that both axioms and theorems can often be used again in other mathematical theories, even those that use different definitions of the operators involved.

Exercise 2.1.2 (Mutual distribution)

For any truth-functional operator $\odot$, prove that

$$(P \odot Q) \lhd b \rhd (R \odot S) \;=\; (P \lhd b \rhd R) \odot (Q \lhd b \rhd S)$$

A similar principle of mutual distribution is called the *interchange law* in category theory [116] or the *abides* principle in functional programming [26]. □

2.2 Composition

The most characteristic combinator of a sequential programming language is sequential composition, often denoted by semicolon. If P and Q are predicates describing behaviours of two programs, their sequential composition

$$P; Q$$

describes a program which may be executed by first executing P, and when P ter-

minates then Q is started. The final state of P is passed on as the initial state of Q, but this is only an intermediate state of $(P;Q)$, and cannot be directly observed. All we know is that it exists. The formal definition of composition therefore uses existential quantification to hide the intermediate observation, and to remove the variables which record it from the list of free variables of the predicate. In order to do this, we need to introduce a fresh set of variables v_0 to denote the hidden observation. These replace the output variables v' of P and the input variables v of Q, so these lists must correspond exactly in length and in type.

Definition 2.2.1 (Sequential composition)

$$\begin{aligned} P(v');Q(v) &=_{df} \exists v_0 \bullet P(v_0) \wedge Q(v_0), \quad \text{provided } out\alpha P = in\alpha' Q = \{v'\} \\ in\alpha(P(v');Q(v)) &=_{df} in\alpha P \\ out\alpha(P(v');Q(v)) &=_{df} out\alpha Q \end{aligned}$$ □

The bound variables v_0 record the intermediate values of the program variables v, and so represent the intermediate state as control passes from P to Q. But this operational explanation is far more detailed than necessary. A clever implementation is allowed to achieve the defined effect by more direct means, without ever passing through any of the possible intermediate states. That is the whole purpose of a more abstract definition of the programming language.

In spite of the complexity of its definition, sequential composition obeys some simple, familiar and obvious algebraic laws. For example, it is associative; to execute three programs in order, one can either execute the first program followed by the other two, or the first two programs followed by the third. Finally, sequential composition distributes leftward (but not rightward) over the conditional. This asymmetry arises because the condition b is allowed to mention only the initial values of the variables, and not the final (dashed) variables.

L1 $P;(Q;R) = (P;Q);R$ (; assoc)

L2 $(P \lhd b \rhd Q);R = (P;R) \lhd b \rhd (Q;R)$ (;-cond left distr)

Proof Hint: Move the existential quantifier outward over predicates that do not contain the quantified variable. □

2.3 Assignment

The assignment is the basic action in all procedural programming languages. Its meaning was explained in the previous chapter, but here we deal more carefully with the phenomenon of alphabets. Every assignment should technically be marked by its alphabet A, which is needed to define an important part of its meaning – that all the variables not mentioned on the left hand side remain unchanged.

Definition 2.3.1 (Assignment)

Let $A = \{x, y, \ldots, z, x', y', \ldots, z'\}$, and let $\alpha e \subseteq A$.

$$x :=_A e \;=_{df}\; (x' = e \wedge y' = y \wedge \ldots \wedge z' = z)$$
$$\alpha(x :=_A e) \;=_{df}\; A \qquad \square$$

Having introduced the complexity of alphabets, we will very seldom subscript an assignment.

The following laws express the basic properties of assignment: that variables not mentioned on the left of := remain unchanged, that the order of the listing is immaterial, and that evaluation of an expression or a condition uses the value most recently assigned to its variables.

L1 $(x := e) = (x, y := e, y)$

L2 $(x, y, z := e, f, g) = (y, x, z := f, e, g)$

L3 $(x := e;\ x := f(x)) = (x := f(e))$

L4 $x := e;\ (P \lhd b(x) \rhd Q) = (x := e; P) \lhd b(e) \rhd (x := e; Q)$

In most programming languages there is a command that has no effect at all! It always terminates, and leaves the values of *all* the variables unchanged. Although its lack of effect can be described fully by the assignment $x := x$, and although it is totally useless for all practical purposes, it is extremely useful for reasoning about programs. We therefore denote it by a special hollow symbol Π_A (pronounced "skip"), where A is its alphabet.

Definition 2.3.2 (Skip)

$$\Pi_A \;=_{df}\; (v' = v), \qquad \text{where } A = \{v, v'\}$$
$$\alpha\Pi_A \;=_{df}\; A \qquad \square$$

The ineffectiveness of Π is neatly captured by its most important algebraic property: that it is the unit for sequential composition.

L5 $P; \Pi_{\alpha P} = P = \Pi_{\alpha P}; P$ (; unit)

In future, we will drop the alphabet subscript on Π, on the grounds that it can be restored in any way that satisfies the constraints imposed by the definition of the neighbouring operators and operands, including the constraint that both sides of an equation or inequation must have the same alphabet.

Assignment to a subscripted variable has the same form as assignment to a simple variable. If w is an array-valued variable (single dimensional for simplicity)

$$w[i] := e$$

denotes an assignment of the value of e to just the i^{th} element of the array. Because this is the only element that changes, implementation is easy and efficient. But in principle, the value of an array variable is a finite function from its index range to its element type, and the change to the value of a single element changes the value of the whole array. The new function is quite similar to the old: it maps all the indices except one to the same value, and just the assigned index is mapped to a new value. Such a change is described by the overriding operator $\oplus$, defined by

$$(w \oplus x)[i] \;=_{df}\; x[i] \lhd i \in \mathit{domain}(x) \rhd w[i]$$

A subscripted assignment to $w[i]$ is mathematically equivalent to an assignment

$$w := w \oplus \{i \mapsto e\}$$

where $\{i \mapsto e\}$ is the singleton function with domain $\{i\}$ that maps i to e.

Definition 2.3.3 (Array assignment)

$$w[i] := e \;=_{df}\; w := w \oplus \{i \mapsto e\}$$ □

2.4 Non-determinism

Non-determinism was described in the previous chapter as simply disjunction of predicates. If P and Q are predicates describing the behaviour of programs with the same alphabet, then we introduce the notation

$$P \sqcap Q$$

to stand for a program which is executed by executing either P or Q, but with no indication which one will be chosen. The introduction of the new notation $\sqcap$ emphasises that the alphabets of both the operands must be the same.

Definition 2.4.1 (Non-deterministic choice)

$$P \sqcap Q \;=_{df}\; P \vee Q, \qquad \text{provided that } \alpha P = \alpha Q$$

$$\alpha(P \sqcap Q) \;=_{df}\; \alpha P$$ □

As an operator of our programming language, non-deterministic choice may be easily implemented by arbitrary selection of either of the operands, and the selection may be made at any time, either before or after the program is compiled or even after it starts execution. The non-deterministic choice satisfies a number of laws common to other forms of choice, including the conditional.

L1 $P \sqcap Q = Q \sqcap P$ ($\sqcap$ symm)

L2 $P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap R$ ($\sqcap$ assoc)

L3 $P \sqcap P = P$ ($\sqcap$ idemp)

Proof From symmetry, associativity and idempotency of disjunction. □

The first law states that it does not make any difference in what order a choice is offered, that is the operator $\sqcap$ is symmetric. The second one is an associative law: a choice between three alternatives can be offered as first a choice between one alternative and the other two, followed (if necessary) by a choice between the other two; it does not matter in which way the choices are grouped. The final law says that a choice between a program and itself offers no choice at all, that is $\sqcap$ is idempotent. Using these laws we can prove that $\sqcap$ distributes through itself.

L4 $P \sqcap (Q \sqcap R) = (P \sqcap Q) \sqcap (P \sqcap R)$ ($\sqcap$ distr)

Proof
$$\begin{aligned} & RHS && \{\textbf{L1} \text{ and } \textbf{L2}\} \\ = \; & (P \sqcap P) \sqcap (Q \sqcap R) && \{\textbf{L3}\} \\ = \; & LHS && \end{aligned}$$
□

All programming combinators defined so far distribute through $\sqcap$. This means that separate consideration of each case is adequate for all reasoning about non-determinism. Finally, $\sqcap$ also distributes through the conditional.

L5 $P \lhd b \rhd (Q \sqcap R) = (P \lhd b \rhd Q) \sqcap (P \lhd b \rhd R)$ (cond-$\sqcap$ distr)

L6 $(P \sqcap Q); R = (P; R) \sqcap (Q; R)$ (;-$\sqcap$ left distr)

L7 $P; (Q \sqcap R) = (P; Q) \sqcap (P; R)$ (;-$\sqcap$ right distr)

L8 $P \sqcap (Q \lhd b \rhd R) = (P \sqcap Q) \lhd b \rhd (P \sqcap R)$ ($\sqcap$-cond distr)

Proof Propositional and predicate calculus. □

As a consequence of distribution through the non-deterministic choice, all program combinators defined so far are monotonic in all arguments; for example,

$$[X; Y \Rightarrow X'; Y'] \text{ whenever } [X \Rightarrow X'] \text{ and } [Y \Rightarrow Y']$$

A further consequence is that all functions defined solely in terms of these programming operators will be monotonic. But not all monotonic operators distribute through $\sqcap$. Consider an operator which executes its argument more than once, for example a squaring operator defined by

$$P^2 =_{df} P; P$$

This is certainly monotonic. But if it is applied to a non-deterministic argument, a different choice may be made when each copy of the argument is executed. Ex-

pansion of the definition and distribution of ; through $\sqcap$ gives

$$\begin{aligned}(Q \sqcap R)^2 &= (Q;Q) \sqcap (Q;R) \sqcap (R;Q) \sqcap (R;R) \\ &= (Q^2 \sqcap R^2) \sqcap (Q;R) \sqcap (R;Q)\end{aligned}$$

This is in general weaker than $Q^2 \sqcap R^2$, but that is not surprising, because it is already known for any monotonic function F that

$$[F(P) \sqcap F(Q) \Rightarrow F(P \sqcap Q)]$$

A product or component P can be made less deterministic by adding the possibility that it behaves like Q. But in general an increase in non-determinism makes the thing worse, no matter what you want to use it for. It is harder to predict how it will behave; it is more difficult to use and control, and more likely to go wrong. This fact is expressed in the simple law

$$[P \Rightarrow (P \sqcap Q)]$$

Taking this argument to extreme, the worst component of all is totally unpredictable and totally uncontrollable; as a result, it is useless for any purpose whatever. There is no limit to the range of its possible behaviour or misbehaviour. The only predicate that it satisfies is the weakest predicate of all, the one that is always true. In a programming language, a program like this is sometimes called ABORT or CHAOS, but we will adopt a more compact notation.

Definition 2.4.2 (Abort)

$$\begin{aligned}\bot_A &=_{df} \mathbf{true} \\ \alpha\bot_A &=_{df} A\end{aligned}$$ □

The notation has been chosen as suggestive of the bottom (weakest) element of the implication ordering

$$[\bot_A \Leftarrow P], \quad \text{for all } P \text{ with alphabet } A$$

In engineering practice, it may well be impossible to find or to build a physical component or assembly whose behaviour is so totally arbitrary. But there are other ways in which a product may become totally undependable, for example foodstuff that has been mis-handled in storage, or a bottle of medicine that has lost its label. The only safe thing to do with it is to throw it away. Since there is no reason to make distinctions in uselessness, these objects may all be represented by the single mathematical abstraction $\bot$. An alternative view is that a non-deterministic product might actually be better, because at least it might exhibit a desirable behaviour which an incorrect deterministic product could never do. This view is attractive, but only because it panders to wishful thinking, a pleasant habit that cannot be allowed in the professional engineer. The only allowable view is that

non-determinism is resolved by a demon which is dedicated to the implementation of Murphy's law

"if it can go wrong it will".

2.5 The complete lattice of relations

The combinator $\sqcap$ may be used to describe non-deterministic behaviour in terms of any finite set of n more deterministic behaviours

$$P_1 \sqcap P_2 \sqcap \ldots \sqcap P_n$$

The disjunction of an infinite set of predicates asserts the existence of at least one of the predicates that is true. Like existential quantification in the predicate calculus, it is impossible to define infinite disjunction in terms of the finite disjunctions of the propositional calculus. It is therefore necessary to introduce the concept by an axiom rather than a formal definition, and that is the role of **L1** below. It is one of the few laws in the book which cannot be proved from definitions, simply because no definition can be given. Fortunately, this axiom alone is strong enough for all the proofs that we need about the properties of $\sqcap$. In fact, it is so strong that there is provably at most one operator that satisfies it. Thus the axiom merely postulates the existence of the operation that it indirectly defines.

Let S_A be some set of predicates, all with alphabet A (which we will henceforth omit). Then $\bigsqcap S$ describes a system that behaves in any of the ways described by any of the relations in S. The law governing this construction applies equally to infinite and to finite non-determinism.

L1 $[P \Leftarrow \bigsqcap S]$ **iff** $([P \Leftarrow X]$ for all X in $S)$

The quantification "for all X in S" on the right hand side of this law is necessary. The square brackets mean universal quantification over all observations described by X, and denoted by free variables in the alphabet of X. Quantification over predicates like X is different; it is known as *higher order* quantification, and will be explained more fully in the next section. The law **L1** for unbounded non-determinism may be easier to understand when it is split into two parts that are together equivalent to it.

L1A $[\bigsqcap S \Leftarrow X]$ for all X in S

L1B If $[P \Leftarrow X]$ for all X in S, then $[P \Leftarrow \bigsqcap S]$

The first of these states that the disjunction of a set is a lower bound of all its members (in the implication ordering), and the second states that it is the greatest such lower bound. The least upper bound operator is often abbreviated to *glb*.

The concept of arbitrary, perhaps infinite, non-determinism has been introduced not as a programming notation but as an aid to reasoning about programs. We will see in Chapter 5 that it is actually impossible to define infinite non-determinism within the notations of a programming language, so when it is needed we cannot avoid just assuming its existence and just postulating its properties. This is done implicitly by introducing the notation in an axiom.

If infinite disjunction is not possible to implement as a program, an empty disjunction is even worse: it yields the predicate **false** which can certainly never be implemented

$$\sqcap\{\} = \mathbf{false}$$

If it did exist, it would satisfy *every* specification

$$[\mathbf{false} \Rightarrow P], \quad \text{for all } P$$

Such a miracle we do not believe in. Indeed, it must remain forever inaccessible to science, because it can never give rise to any observation. Any theory which implies its existence must be to some degree unrealistic, and use of such a theory in engineering practice must involve some element of risk. Nevertheless the miracle is so useful as a mathematical abstraction in reasoning about programs that we give it a special symbol.

Definition 2.5.1 (Miracle)

$$\top_A \ =_{df} \ \mathbf{false}$$

$$\alpha\top_A \ =_{df} \ A$$ □

The notation suggests the top (strongest) element in the implication ordering. We will later see some intuitive content for $\top_A$: it stands for a product that can never be used, because its conditions of use are impossible to satisfy, or its instructions are impossible to carry out. Think of a computer with no on-switch, or a bottle of medicine that cannot be opened. A claim for miraculous powers of such a product would be impossible to refute.

In predicate calculus, the dual of disjunction is conjunction. Because of the danger of contradiction, this certainly cannot be included in a programming language, but in reasoning about programs, it is just as useful, and can just as usefully be extended to arbitrary sets of relations. The conjunction of all relations in a set S is denoted by $\sqcup S$, and gives the least upper bound (*lub*) of the set S. Its postulated defining property is dual to that of disjunction, with the implications in the opposite direction

$$[P \Rightarrow \sqcup S] \quad \textbf{iff} \quad ([P \Rightarrow X] \text{ for all } X \text{ in } S)$$

A system is described by $\bigsqcup S$ just if it is described by every relation in S. The least upper bound of an empty set is the bottom of the lattice

$$\bigsqcup\{\} = \mathbf{true}$$

A mathematical space with an ordering that has the *lub* and the *glb* of all subsets of its elements is known as a *complete lattice*, and we have postulated that the space of relations deserves this title. Furthermore it is *distributive* in the sense that *finite* disjunction distributes into arbitrary conjunction.

L2 $(\bigsqcup S) \sqcap Q = \bigsqcup\{P \sqcap Q \mid P \in S\}$

Many properties of conjunction can be translated to a similar property of disjunction, by reversing the direction of implication (which leaves equations unchanged). Such a property is called *dual*, and the distribution law gives a good example.

L3 $(\bigsqcap S) \sqcup Q = \bigsqcap\{P \sqcup Q \mid P \in S\}$

These distributive properties deserve special names. A function F will be called *disjunctive* if it distributes through arbitrary disjunctions, and *conjunctive* if it distributes through arbitrary conjunctions. For example, sequential composition is disjunctive in each of its two arguments

L4 $(\bigsqcap S); Q = \bigsqcap\{P; Q \mid P \in S\}$ (; left univ disj)

L5 $R; (\bigsqcap S) = \bigsqcap\{R; P \mid P \in S\}$ (; right univ disj)

2.6 Recursion

To explain recursion, we will need to formalise a concept of quantification over variables whose values range over relations with a given alphabet A. We shall use capital letters $X_A, Y_A, \ldots$ for this purpose (but usually leave out the subscript A). Of course, the value of X_A is a relation which also contains free variables in the alphabet A. These variables stand for observations; they are written in lower case, and are subject to quantification by square brackets. In this section, we make no mention of observation variables. They are all encapsulated within a single capital letter, which is known as a *second-order variable* to indicate its purpose and status as standing for a predicate rather than an observation.

Let X be a variable to stand for a call of the recursive program which we are about to define. X is then used, perhaps more than once, to build up the body of the program, using assignments and the various combinators of the programming language. The result may be seen as a formula $F(X)$, whose value will be known as soon as the value of X is known. The engineer may prefer to think of $F(X)$ as an assembly with a vacant slot labelled X, into which some component of an

appropriate kind may later be plugged. Now the magic of a recursive definition is to declare that X is just the same as the result of applying F to X itself, or in symbols, it satisfies the equation

$$X = F(X)$$

Such an X is called a *fixed point* of the function F because application of F leaves it unchanged. The same trick can be played with two or more unknowns in a definition by *mutual recursion.* This is expressed by simultaneous equations, for example

$$X = F(X,Y) \quad \text{and} \quad Y = G(X,Y)$$

But for simplicity we shall concentrate on single recursions; multiple recursion is treated fully in Section 9.4.

How does the magic work? Logicians and philosophers, ever since Aristotle, have not allowed themselves to define an unknown X in terms of itself by mentioning it on the right hand side of its own definition. Engineers are even more sceptical: How can the object $F(X)$ be plugged into itself as one of its own components? In physical reality, such feedback can lead to discontinuities, singularities, race conditions, explosions, etc.

Fortunately, computing scientists have found an effective way in practice to implement recursively defined programs by means of a stack. Even earlier, mathematicians discovered how to solve the theoretical paradox of apparently circular definition. The solution we present here is due to Tarski [175]. It depends crucially on the fact that relations are members of a complete lattice, and that all the operators that interest us are monotonic. As a result, any function F which is defined solely in terms of these operators will also be monotonic. Tarski's famous theorem shows that every such function has a fixed point, defined by a formula to be given below.

But there may well be more than one fixed point: Tarski has proved that there will be a complete lattice of them. For example, the miracle $\top$ is a fixed point of sequential composition, because it satisfies

$$X = P; X$$

Because it is the strongest of all relations, it is the strongest of all the fixed points. But we want the recursion to be implementable, which $\top$ certainly is not. We therefore take the other obvious alternative, the *weakest* of the fixed points – the one that is easiest to implement, although most difficult to use. The effect of this decision is that erroneous recursions (those that do not terminate when executed) are revealed as totally useless: they give rise to the worst of all behaviours, namely $\bot$. The weakest fixed point of a monotonic function F will be denoted by μF. It

can be implemented as a single non-recursive call of a parameterless procedure with name X and with body $F(X)$. Occurrences of X within $F(X)$ are implemented as recursive calls on the same procedure. The mathematical definition is much more abstract.

Definition 2.6.1 (Weakest fixed point)

$$\mu F =_{df} \sqcap\{X \mid [X \Rightarrow F(X)]\}$$

where the join operator $\sqcap$ is applied to the set of all solutions of $[X \Rightarrow F(X)]$. A definition using the equation $[X = F(X)]$ would give the same result. □

The following laws state that this formula is indeed a fixed point of F, and that it is the weakest one.

L1 $[Y \Rightarrow \mu F]$ whenever $[Y \Rightarrow F(Y)]$ (weakest fixed point)

L2 $[F(\mu F) \equiv \mu F]$ (fixed point)

Proof of **L1**

$$\begin{array}{lll} & [Y \Rightarrow F(Y)] & \{\text{by set theory}\} \\ \Rightarrow & Y \in \{X \mid [X \Rightarrow F(X)]\} & \{2.5\mathbf{L1A}\} \\ \Rightarrow & [Y \Rightarrow \mu F] & \end{array}$$

The following proof for **L2** is due to Dijkstra and Scholten [52].

$$\begin{array}{lll} & \mathit{true} & \{\mathbf{L1}\} \\ \equiv & \forall X \bullet [X \Rightarrow F(X)] \Rightarrow [X \Rightarrow \mu F] & \{F \text{ is monotonic}\} \\ \equiv & \forall X \bullet [X \Rightarrow F(X)] \Rightarrow [F(X) \Rightarrow F(\mu F)] & \{\Rightarrow \text{ is transitive}\} \\ \Rightarrow & \forall X \bullet [X \Rightarrow F(X)] \Rightarrow [X \Rightarrow F(\mu F)] & \{2.5\mathbf{L1B}\} \\ \Rightarrow & [\mu F \Rightarrow F(\mu F)] & \{F \text{ is monotonic}\} \\ \Rightarrow & [\mu F \Rightarrow F(\mu F)] \wedge [F(\mu F) \Rightarrow F(F(\mu F))] & \{\mathbf{L1}, \text{ with } F(\mu F) \text{ for } Y\} \\ \Rightarrow & [\mu F \Rightarrow F(\mu F)] \wedge [F(\mu F) \Rightarrow \mu F] & \\ \equiv & [\mu F \equiv F(\mu F)] & \end{array}$$ □

The second law **L2** is a mathematical formulation of the "copy rule", which makes a copy of the whole procedure in the place of each one of its calls. This is often given as an explanation to programmers of the meaning of a procedure call. In the case of recursions that terminate properly, there is only a single fixed point, and the copy rule gives its complete meaning. But the mathematics must also deal with the risk of non-termination, because that is the only way the mathematics can help in avoiding that risk. That is why the first law is also needed.

It is usually convenient in a programming language to combine the definition of the function F with the application of the operator μ. The body of the recursive procedure is written as a program text containing occurrences of the higher order

variable X

$$\ldots X \ldots X \ldots$$

A call of the corresponding procedure is obtained just by putting $\mu X \bullet$ in front of this

$$\mu X \bullet \ldots X \ldots X \ldots$$

The name X is thereby declared as the name of a recursive procedure, and its occurrences within the text are interpreted accordingly as recursive calls.

Definition 2.6.2

$$\mu X \bullet \ldots X \ldots X \ldots \ =_{df} \ \mu F$$

where $F =_{df} \lambda X \bullet \ldots X \ldots X \ldots$. □

Example 2.6.3 (Iteration)

A simple common case of recursion is the iteration or while loop. If b is a condition

$$b * P \qquad \text{(while } b \text{ do } P\text{)}$$

repeats the program P as long as b is true before each iteration. More formally, it can be defined as the recursion

$$b * P \ =_{df} \ \mu X \bullet ((P; X) \lhd b \rhd I\!I) \qquad \square$$

An even simpler example (but hopefully less common) is the infinite recursion which never terminates

$$\mu X \bullet X$$

This is the weakest fixed point of the identity function, or in other words, the weakest solution of the trivial equation

$$X = X$$

It is therefore the weakest of all predicates, namely **true**. In engineering practice, a non-terminating program is the worst of all programs, and must be carefully avoided by any responsible engineer. That is our justification for practical use of a theory which equates any totally non-terminating program with a totally unpredictable one, which is the weakest in the lattice ordering.

Consider now the program which starts with an infinite loop and ends with the assignment of constants to all variables.

$$(\mu X \bullet X); (x, y, \ldots, z := 3, 12, \ldots, 17)$$

In any normal implementation, this would fail to terminate, and so be equal to $(\mu X \bullet X)$. Unfortunately, our theory gives the unexpected result

$$x' = 3 \wedge y' = 12 \wedge \ldots \wedge z' = 17$$

This is the same as if the prior non-terminating program had been omitted. To achieve this result, an implementation would have to execute the program backwards, starting with the assignment, and stopping as soon as the values of all the variables are known. While backward execution is not impossible (indeed, it is standard for a lazy functional language, as discussed in Section 9.3), it is certainly not efficient for a normal procedural language. Since we want to allow the conventional forward execution, we are forced to accept the practical consequence that the program

$$(\mu X \bullet X) ; P$$

will fail to terminate for any program P, and the same holds for

$$P ; (\mu X \bullet X)$$

Substituting $(\mu X \bullet X)$ by its value **true**, we observe in practice of all programs P that

$$\begin{aligned} \mathbf{true}; P &= \mathbf{true} \\ P; \mathbf{true} &= \mathbf{true} \end{aligned}$$

These laws state that **true** is a zero for sequential composition.

But these laws are certainly not valid for an arbitrary relation P (e.g. if P is **false**). As always in science, if a theory makes an incorrect prediction of the behaviour of an actual system, it is the theory that must take the blame. All the assumptions on which the theory is based may then be questioned, starting with the most questionable. In the case of recursion, the obvious first question is whether we were right to accept its definition as the *weakest* fixed point of the defining function. Maybe the *strongest* fixed point would be better.

Exercise 2.6.4

Prove the following fixed point rules [121]

(1) $\mu X \bullet F(G(X)) = F(\mu X \bullet G(F(X)))$

(2) If $[G(H(X)) \Rightarrow F(G(X))]$ for all X, then $[G(\mu H) \Rightarrow \mu F]$. □

2.7* Strongest fixed point

The strongest fixed point of a monotonic function on a complete lattice can be defined as the dual of the weakest.

Definition 2.7.1 (Strongest fixed point)

$$\nu F \ =_{df} \ \neg\mu X \bullet \neg F(\neg X) \qquad \square$$

Its properties are easily proved by duality: just reverse the direction of the arrows in the laws for the weakest fixed point.

L1 $[\nu F \Rightarrow S]$ whenever $[F(S) \Rightarrow S]$

L2 $[F(\nu F) \equiv \nu F]$

This new interpretation of recursion gives the entirely opposite view of non-termination

$$(\nu X \bullet X) = \mathbf{false}$$

This is a plausible view. One can argue that the reason why the set of observations is empty is because the implementation in practice fails to reach an observable state. Furthermore, the zero laws are immediately validated, because

L3 $\mathbf{false}; P = \mathbf{false} = P; \mathbf{false}$

A recursively defined program is much easier to prove correct using the strongest fixed point νF rather than the weakest fixed point μF. In order to prove that νF meets a specification S, all that is needed by **L1** is to prove $[F(S) \Rightarrow S]$. That is why several early theories of programming [85] have interpreted recursion as the strongest fixed point. Unfortunately the implication in 2.6**L1** for μF goes in the opposite direction. It could be used to prove that a program meets a recursively defined specification, but that is rarely what is wanted.

There are two serious objections to the use of the strongest fixed point. The first has already been mentioned: a non-terminating program implements falsity, which will satisfy *every* specification S

$$[\mathbf{false} \Rightarrow S]$$

This would invalidate the use of implication to model correctness of designs, requiring abandonment of our whole logic of engineering design. Even scientists are allowed to resist the questioning of their most basic assumptions! The second objection is more practical. It is embodied in the law

$$(\mathbf{false} \sqcap P) = P$$

This places a serious burden on the implementor of non-determinism: it requires that if one of the alternatives fails to terminate, the effect must be the same as that of the other alternative. The trouble is that it is impossible to know in advance whether or which of the programs is going to fail; an implementor must therefore execute both alternatives (e.g. by timesharing or running them in parallel), and take the result of whichever terminates first. This wastes all the computing resource allocated to the rejected alternative, and it is certainly not the intended implementation of non-determinism, which is much easier (just select the easier alternative whenever you like).

Although we reject the strongest fixed point as the proper meaning of recursion in a programming language, we can fortunately continue to use the very simple rule **L1** for proving its correctness. Of course, there is an additional proof obligation which is the subject of the remainder of this section: we need to show that the strongest and weakest fixed points are in fact the same; or in other words, that there is altogether only one fixed point. This will be the case for any recursion which is guaranteed in all circumstances to terminate.

A recursion which terminates only conditionally (on some condition C) can be proved correct in a similar way. In this case, it is sufficient to prove conditional equality of the two fixed points, that is

$$[C \wedge \mu F \equiv C \wedge \nu F]$$

This defines a concept of *approximate equality* modulo C: the two fixed points will appear equal to an observer whose observations are confined to the set C. But that is good enough if the specification itself is of the form $(C \Rightarrow S)$. The full proof rule for weakest fixed points can now be formalised.

Lemma 2.7.2

If $\quad [F(C \Rightarrow S) \;\Rightarrow\; (C \Rightarrow S)]$ and $[C \;\Rightarrow\; (\mu F \equiv \nu F)]$

then $[\mu F \;\Rightarrow\; (C \Rightarrow S)]$ □

The essential proof of conditional equality of fixed points is often conducted by a process of successive approximation. The condition C is expressed as an infinite disjunction of a weakening chain of predicates.

Definition 2.7.3 (Approximation chain)

A set of predicates $E = \{E_i \mid i \in \mathcal{N}\}$ is called an approximation chain for C if

$$E_0 = \textbf{false}$$
$$[E_i \Rightarrow E_{i+1}], \quad \text{for all } i \text{ in } \mathcal{N}$$
$$C = \bigvee_i E_i$$

□

If X is any predicate, $X \wedge E_n$ is regarded as the n^{th} approximation to $X \wedge C$, and we state the obvious lemma.

Lemma 2.7.4

If $\quad X \wedge E_n = Y \wedge E_n, \quad$ for all n in $\mathcal{N}$

then $\quad X \wedge (\bigvee_n E_n) = Y \wedge (\bigvee_n E_n)$ □

The most important use of an approximation chain E is to help in proving equality (modulo $\bigvee E$) of the strongest and weakest fixed points of a function F. For this, E must be chosen so that the $(n+1)^{\text{st}}$ approximation of $F(X)$ can be computed from just the n^{th} approximation to X. A function with this property is said to be E-constructive, and it has at most one fixed point modulo E.

Definition 2.7.5 (E-constructive)

Let E be an approximation chain for C. A function F is E-constructive if

$$F(X) \wedge E_{n+1} = F(X \wedge E_n) \wedge E_{n+1}$$

for all X and n. □

Theorem 2.7.6

If E is an approximation chain for C, and if F is E-constructive, then

$$C \wedge \mu F = C \wedge \nu F$$

Proof By Lemma 2.7.4, it is sufficient to show that

$$E_n \wedge \mu F = E_n \wedge \nu F, \quad \text{for all } n$$

This is done by induction on n:

(0) $E_0 \wedge \mu F = E_0 \wedge \nu F \quad \{E_0 = \textbf{false}\}$

(1)
$$\begin{aligned} E_{n+1} \wedge \mu F &= E_{n+1} \wedge F(\mu F) && \{\text{def constructive and 2.6}\mathbf{L2}\} \\ &= E_{n+1} \wedge F(E_n \wedge \mu F) && \{\text{induction}\} \\ &= E_{n+1} \wedge F(E_n \wedge \nu F) && \{\text{def constructive and } \mathbf{L2}\} \\ &= E_{n+1} \wedge \nu F \end{aligned}$$
□

This theorem describes the general condition for proving the termination of a recursion. It shows the contribution of the strongest fixed point in reasoning about recursive programs. But it does not give a solution to the problem described in Section 2.6; this is postponed to the next chapter, which the keen or impatient reader could immediately skip to. As so often in science, the solution requires a

slight complication of the theory, which has to recognise the relevance of additional factors, and introduce new variables to denote them. These stand for properties of the world which are perhaps not themselves directly observable. However, they can be inferred from more direct observations, and they are successful in predicting others. All the most fundamental concepts in science have been discovered in this way, as corrections to some simpler theory; examples range from friction and viscosity in mechanics to colour and charm in particle physics.

2.8* Preconditions and postconditions

A condition p has been defined as a predicate not containing dashed variables. It describes the values of global variables of a program Q before its execution starts. Such a condition is therefore called a *precondition* of the program. If r is a condition, let r' be the result of placing a dash on all its variables. As a result, r' describes the values of the variables of a program Q when it terminates. Such a condition is therefore said to be a *postcondition* for the program. Important aspects of the behaviour of a program can often be specified simply by means of a postcondition. Usually, this needs to be accompanied by a precondition, which the designer of the program has to *assume* to be satisfied at the start. The discharge of this assumption is the responsibility of the user of the program, or the designer of some previously executed part of the program, who will accept the precondition of the later part as a postcondition of the earlier part.

The overall specification of the program can often be formalised as a simple implication $(p \Rightarrow r')$. As usual, the correctness of a program Q is also interpreted as an implication, and the triple (precondition, program, postcondition) is known as a Hoare triple [85].

Definition 2.8.1 (Hoare triple)

$$p\{Q\}r =_{df} [Q \Rightarrow (p \Rightarrow r')] \qquad \square$$

This definition validates a number of classical proof rules for proving the correctness of a program.

Theorem 2.8.2 (Hoare proof rules)

L1 If $p\{Q\}r$ and $p\{Q\}s$ then $p\{Q\}(r \wedge s)$

L2 If $p\{Q\}r$ and $q\{Q\}r$ then $(p \vee q)\{Q\}r$

L3 If $p\{Q\}r$ then $(p \wedge q)\{Q\}(r \vee s)$

L4 $r(e)\{x := e\}r(x)$

L5 If $(p \wedge b)\{Q1\}r$ and $(p \wedge \neg b)\{Q2\}r$ then $p\{Q1 \lhd b \rhd Q2\}r$

L6 If $p\{Q1\}s$ and $s\{Q2\}r$ then $p\{Q1;Q2\}r$

L7 If $p\{Q1\}r$ and $p\{Q2\}r$ then $p\{Q1 \sqcap Q2\}r$

L8 If $b \wedge c\{Q\}c$ then $c\{\nu X \bullet Q; X \lhd b \rhd \Pi\}(\neg b \wedge c)$

L9 $false\{Q\}r$ and $p\{Q\}true$ and $p\{\mathbf{false}\}false$ and $p\{\Pi\}p$

Proof of **L8** Let $Y =_{df} c \Rightarrow \neg b' \vee c'$. By 2.7**L1**, it is sufficient to prove

$$[(Q;Y) \lhd b \rhd \Pi \Rightarrow Y]$$

Assume the antecedents

$$\begin{array}{lll} & (Q;Y \lhd b \rhd \Pi) \wedge c & \{b \wedge (P;Q) = (b \wedge p);Q\} \\ = & (b \wedge c \wedge Q);Y \vee (\neg b \wedge c \wedge \Pi) & \{b \wedge c\{Q\}c, c\{Y\}\neg b \wedge c, \textbf{L6} \text{ and } \textbf{L9}\} \\ \Rightarrow & \neg b' \wedge c' & \Box \end{array}$$

These proof rules use the preconditions and postconditions purely as program specifications; they permit reasoning about program correctness to be detached as quickly as possible from the text of the program. The mathematical hypotheses which need proof can be extracted mechanically with the aid of a verification condition generator (VCG) [109]. But conditions can play an even more vital role in explaining the meaning of a program if they are included at appropriate points as an integral part of the program documentation. Such a condition is asserted or expected to be true at the point at which it is written. It is known as a Floyd assertion [57]; formally, it is defined to have no effect if it is true, but to cause failure if it is false. Failure is something that everyone agrees must be avoided.

It is often important to distinguish who or what is responsible for a failure. If it is the designer or the programmer, failure should be represented by $\bot$, because this can never be proved to meet its specification. But if the failure is due to the failure of the environment to meet agreed commitments, then the appropriate representation is by $\top$. There is then no obligation on the designer to prove anything. Preconditions are a prime example of this kind of permitted assumption.

Definition 2.8.3 (Floyd assertion and assumption)

$$\begin{array}{lll} c_\bot & =_{df} & \Pi \lhd c \rhd \bot \quad \text{(assertion)} \\ c^\top & =_{df} & \Pi \lhd c \rhd \top \quad \text{(assumption)} \end{array} \quad \Box$$

Definition 2.8.1 of correctness can be rewritten in a number of different ways.

$$\begin{array}{rcl} [Q \Rightarrow (p \Rightarrow r')] & = & [p \Rightarrow (Q \Rightarrow r')] \\ & = & [p \Rightarrow (\forall v'.Q \Rightarrow r')] \\ & = & [p \Rightarrow \neg\exists v'(Q \wedge \neg r')] \\ & = & [p \Rightarrow \neg(Q;\neg r)] \end{array}$$

This last reformulation gives an answer to the question: What is the *weakest* precondition under which execution of Q is guaranteed to achieve the postcondition r' ?

Definition 2.8.4 (Weakest precondition)

$$Q\,\mathbf{wp}\,r \;=_{df}\; \neg(Q;\neg r) \qquad \square$$

The calculation of the weakest precondition may be assisted by the following laws.

L10 $(x := e)\,\mathbf{wp}\,r(x) \;=\; r(e)$

L11 $(P;Q)\,\mathbf{wp}\,r \;=\; P\,\mathbf{wp}\,(Q\,\mathbf{wp}\,r)$

L12 $(P \lhd b \rhd Q)\,\mathbf{wp}\,r \;=\; (P\,\mathbf{wp}\,r) \lhd b \rhd (Q\,\mathbf{wp}\,r)$

L13 $(P \sqcap Q)\,\mathbf{wp}\,r \;=\; (P\,\mathbf{wp}\,r) \wedge (Q\,\mathbf{wp}\,r)$

The weakest precondition satisfies a number of additional laws, known as *healthiness* conditions. Firstly, it is monotonic in its postcondition. An easier postcondition in general makes the precondition easier too.

L14 If $[r \Rightarrow s]$ then $[Q\,\mathbf{wp}\,r \Rightarrow Q\,\mathbf{wp}\,s]$

It is, however, *antimonotonic* in the program which is its first argument. A weaker program is less predictable, less controllable, and more likely to fail. It is therefore harder to satisfy the precondition for success of the weaker program.

L15 If $[Q \Rightarrow S]$ then $[S\,\mathbf{wp}\,r \Rightarrow Q\,\mathbf{wp}\,r]$

Antimonotonicity means that **wp** itself can never be included in an implemented programming language, and even when used in specifications or designs, it must never be involved in recursion through its first argument.

A stronger property than monotonicity is conjunctivity, defined as distribution through conjunction of an arbitrary set of postconditions.

L16 $Q\,\mathbf{wp}\,(\bigwedge R) = \bigwedge\{Q\,\mathbf{wp}\,r \mid r \in R\}$

This means that (when R is empty) our definition of **wp** differs slightly from the original definition given by Dijkstra [50], which does not satisfy

$$Q\,\mathbf{wp}\,true = true$$

This discrepancy is resolved by the method introduced in Chapter 3.

At the other extreme, one would expect that the achievement of the impossible postcondition *false* would also be impossible.

L17 $Q\,\mathbf{wp}\,false = false,$ provided $Q;\mathbf{true} = \mathbf{true}$

This is known as the law of the excluded miracle, and is usually quoted without the proviso. In the next chapter we will show that the omission of the proviso is justified, because all programs satisfy it anyway.

In specifying and designing a program, the postcondition conveys far more significant information than the precondition. It is therefore recommended practice to design a program backwards from the postcondition. Given a proposed design of a final program segment Q, it is possible (by calculation mostly) to deduce the *weakest* precondition under which Q will satisfy the postcondition r. This precondition can often be simplified by strengthening, and then it is taken as the postcondition in the design of the next preceding segment of the program. The systematic design continues until the calculated precondition is implied by the originally stated precondition for the whole program. Then the programming task is complete.

An interesting generalisation of the weakest precondition is the *weakest prespecification*. It is obtained merely by allowing the second argument of **wp** to be an arbitrary relation, containing both dashed and undashed variables. It gives an answer to the question: What is the weakest specification of a program P whose execution before Q is certain to meet specification S?

L18 $[(P;Q) \Rightarrow S]$ **iff** $[P \Rightarrow Q \,\mathbf{wp}\, S]$

The relevance of conditions in reasoning about programs was known to Turing and von Neumann [179]. They were rediscovered by Floyd (and called assertions) [57] and by Naur (and called generalised snapshots) [134]. Floyd suggested that the assertions encapsulate the meaning of a program, and Hoare suggested that the laws of reasoning with assertions should be accepted as an axiomatic definition of the meaning of the whole programming language [85]. The axiomatic approach was adopted also by Dijkstra in his development of weakest preconditions [50]. The approach of this book is not axiomatic: we prefer to prove the necessary laws as theorems based on an independent mathematical definition of the meaning of a program as a relation. In this we follow the example of standard mathematical practice, to provide a link between its purer and the more applied branches. The axioms postulated in one branch are proved as the theorems of a more basic theory.

Exercises 2.8.5

Prove the following laws for Floyd assertions and assumptions.

(1) $b_\perp ; c_\perp = (b \wedge c)_\perp = b_\perp \sqcap c_\perp$

(2) $b^\top ; c^\top = (b \wedge c)^\top = b^\top \sqcup c^\top$

(3) $b^\top ; b_\perp = b^\top$ □

2.9 Variable declarations

An essential characteristic of engineering is that the internal details of the working of its products are of no concern to their users, which is why they are usually concealed from observation and interference by an opaque casing and an instruction not to open it. In Chapter 1 it was shown that this concealment from observation is represented in the predicate calculus by quantification over the irrelevant free variables, which are then removed from the alphabet of the predicate. Section 2.2 gave an important example – the forgetting of the intermediate values of the variables on transition between the first and second operands of a sequential composition. In this section we extend the method of concealment from the temporal domain of program execution to the spatial domain of computer storage.

An essential characteristic of program design is that the efficient implementation of any non-trivial algorithm requires the invention and introduction of additional temporary variables; they store the results of intermediate calculation for repeated use on later occasions. The values of these variables are of no concern to the user of the program, and they should be concealed, as usual, by existential quantification. Since each program variable is represented in the predicate by two observational variables, one dashed and one undashed, both of these must be treated appropriately.

To introduce a new program variable x we use the form of *declaration*

$$\mathbf{var}\, x$$

which permits the variable x to be used in the portion of the program that follows it. The complementary operation (called *undeclaration*) takes the form

$$\mathbf{end}\, x$$

and terminates the region of permitted use of the variable x. The portion of program Q in which a variable x may be used is called its *scope*; it is bracketed on the left and on the right by the declaration and undeclaration

$$\mathbf{var}\, x;\ Q;\ \mathbf{end}\, x$$

In this case, x is called a *local* variable of Q, and Q is called a *block*. However, it is sometimes (e.g. in Section 4.4) beneficial both in theory and in practice to reason separately about unbracketed fragments of the form $(\mathbf{var}\, x;\ Q)$ and $(Q;\ \mathbf{end}\, x)$.

In programming languages, as in conventional mathematical reasoning, it is usual to associate a type T with each variable, and this is often done on the occasion of its declaration

$$\mathbf{var}\, x : T$$

The type T determines the range of possible values for the variable, and there is usually a syntactically enforceable guarantee that no assignment can take a variable's value outside its declared range. (Alternatively, such assignment is specified to lead to disaster.) In this section it is convenient to ignore all distinctions of type, so that the results will apply to languages with any kind of type structure.

Definition 2.9.1 (Declaration and undeclaration)

Let A be an alphabet which includes x and x'. Then

$$\begin{aligned} \mathbf{var}\, x &=_{df} \exists x \bullet \Pi_A \\ \mathbf{end}\, x &=_{df} \exists x' \bullet \Pi_A \\ \alpha(\mathbf{var}\, x) &=_{df} A \setminus \{x\} \\ \alpha(\mathbf{end}\, x) &=_{df} A \setminus \{x'\} \end{aligned}$$ □

Note that the alphabet constraints forbid the redeclaration of a variable within its own scope. For example, $\mathbf{var}\, x\,;\, \mathbf{var}\, x$ is disallowed because

$$x' \in out\alpha(\mathbf{var}\, x) \quad \text{but} \quad x \notin in\alpha(\mathbf{var}\, x)$$

and as a result the composition is undefined. We shall see later how this restriction may be relaxed, to permit nested and even recursive declarations of the same variable.

Declaration and undeclaration act exactly like existential quantification over their scopes

$$\begin{aligned} \mathbf{var}\, x; Q &= \exists x \bullet Q \\ Q; \mathbf{end}\, x &= \exists x' \bullet Q \end{aligned}$$

For convenience we allow variables to be declared together in a list provided they are all distinct

$$\mathbf{var}\, x, y, \ldots, z \quad \text{instead of} \quad \mathbf{var}\, x;\ \mathbf{var}\, y;\ \ldots;\ \mathbf{var}\, z$$

We will also allow initialisation to be combined with declaration

$$\mathbf{var}\, x := e \quad \text{instead of} \quad \mathbf{var}\, x; x := e$$

The algebraic laws for declaration closely match those for existential quantification. Both declaration and undeclaration are commutative.

L1 $(\mathbf{var}\, x\,;\, \mathbf{var}\, y) = (\mathbf{var}\, y\,;\, \mathbf{var}\, x) = \mathbf{var}\, x, y$

L2 $(\mathbf{end}\, x\,;\, \mathbf{end}\, y) = (\mathbf{end}\, y\,;\, \mathbf{end}\, x) = \mathbf{end}\, x, y$

L3 $(\mathbf{var}\, x\,;\, \mathbf{end}\, y) = (\mathbf{end}\, y\,;\, \mathbf{var}\, x)$, provided x and y are distinct

The initial value of a declared variable is arbitrarily non-deterministic.

L4 If T is the type of x, then $\mathbf{var}\, x = \sqcap\{\mathbf{var}\, x := k \mid k \in T\}$

Declaration and undeclaration distribute through a conditional as long as no interference occurs with the condition.

L5 If x is not free in b, then

$$\begin{aligned} \mathbf{var}\, x;\, (P \lhd b \rhd Q) &= (\mathbf{var}\, x;\, P) \lhd b \rhd (\mathbf{var}\, x;\, Q) \\ \mathbf{end}\, x;\, (P \lhd b \rhd Q) &= (\mathbf{end}\, x;\, P) \lhd b \rhd (\mathbf{end}\, x;\, Q) \end{aligned}$$

$\mathbf{var}\, x$ followed by $\mathbf{end}\, x$ has no effect whatsoever. In practice, there is a risk that available computer storage may run out; such unfortunate events may be modelled within the theory by introducing variables to account for resource utilisation, as in Example 7.2.1.

L6 $\mathbf{var}\, x;\, \mathbf{end}\, x = I\!I$

The next law states that the sequential composition of $\mathbf{end}\, x$ with $\mathbf{var}\, x$ has no effect whenever it is followed by an update of x that does not rely on the previous value of x.

L7 $(\mathbf{end}\, x;\, \mathbf{var}\, x := e) = (x := e)$, provided that x does not occur in e

Assignment to a variable just before the end of its scope is irrelevant.

L8 $(x := e; \mathbf{end}\, x) = \mathbf{end}\, x$

One of the purposes of introducing local variables into a program is to break up a complex calculation into a series of simpler calculations that can be understood and performed separately. For example,

$$y := y^2 \times (6y^4 + 1)/12$$

is logically equivalent to the block

$$\mathbf{var}\, x := y^2;\ y := x \times (6 \times x^2 + 1)/12;\ \mathbf{end}\, x$$

The declaration and undeclaration of x are essential to equalise the alphabets of the two programs; without equal alphabets they cannot even be compared, let alone proved equal.

A conventional digital computer is not capable of direct execution of generally bracketed expressions of a programming language. Reduction of the complexity of expressions is one of the essential tasks in the translation of a program from its high level language to the machine code of a computer that will execute it. In fact, many computers can execute only the simplest assignments, with just two or three

operands, and most of these operands must be selected from among the available machine registers. The registers are usually given names like a, b, c, which are assumed distinct from all variables of the programming language. A list of typical machine code instructions is given in Table 2.9.2.

name	effect
load x	$a, b := x, a$
store x	$x, a := a, b$
add	$a := a + b$
subtract	$a := a - b$
multiply	$a := a \times b$
divide	$a := a/b$

Table 2.9.2 Machine code instructions

In principle, the translation of a source program is accomplished by algebraic transformations, which reduce it to a form containing only instructions in the repertoire of the machine, as shown in Table 2.9.4. The registers of the machine are just components of an implementation on a particular machine. After they are hidden, the meaning of the machine code is the same as that of the high level program – or better.

Exercise 2.9.3 (Compilation of assignments)

Consider a programming language with suitable restrictions on the operations permitted in expressions. Prove that every sequence of assignments can be translated into a sequence of machine code instructions (Table 2.9.2), together with declarations of additional local variables standing for machine registers. □

var $a, b;$	$a, b := y, a;$	load y
	$a, b := z, a;$	load z
	$a := a \times b;$	multiply
	$a, b := w, a;$	load w
	$a := a + b;$	add
	$x, a := a, b$	store x
end a, b		

Table 2.9.4 Machine code for $(x := y \times z + w)$

If x and x' are in the alphabet of Q, the bracketed declarations

$$\mathbf{var}\, x; Q; \mathbf{end}\, x$$

provide a means of removing these variables from the alphabet. The converse operation is defined on a program R which does not have x or x' in its alphabet, and produces a result which does. Since R certainly does not update x, it is reasonable to assume that the final value x' is the same as the initial value x.

Definition 2.9.5 (Alphabet extension)

Let $x, x' \notin \alpha R$

$$\begin{aligned} R_{+x} &=_{df} R \wedge (x' = x) \\ \alpha R_{+x} &=_{df} \alpha R + \{x, x'\} \end{aligned}$$ □

The following laws state how declaration and alphabet extension cancel each other.

L9 If R does not mention x, then

$$\mathbf{var}\, x; R_{+x}; P; \mathbf{end}\, x = R; \mathbf{var}\, x; P; \mathbf{end}\, x$$

$$\mathbf{var}\, x; P; R_{+x}; \mathbf{end}\, x = \mathbf{var}\, x; P; \mathbf{end}\, x; R$$

Alphabet extension enables sequential composition to be applied to operands with non-matching alphabets. For example, if x is in αQ but x, x' are not in αP, then $(P; Q)$ is illegal, but $(P_{+x}; Q)$ can be written instead. This alphabet extension is unavoidable if P is a call of a procedure whose body was written outside the scope of the variable x, as described in Section 9.1. In fact, it is convenient just to write the simpler illegal form, on the understanding that it should be made legal by extending the alphabets of one or both of the operands in the minimal possible way.

The most important application of alphabet extension is for the declaration of local variables within recursively defined programs, for example

$$R =_{df} (\mu X \bullet \mathbf{var}\, x; \ldots; X; \ldots; \mathbf{end}\, x)$$

Because x is a local variable of R, the alphabet of R by Definition 2.9.1 excludes x and x'. But by the fixed point property of recursion

$$R = \mathbf{var}\, x; \ldots; R; \ldots; \mathbf{end}\, x$$

Unfortunately, the right hand side of this equation is now syntactically incorrect, because the alphabet of the internal R does not match its context.

The solution is to extend the alphabet of the internal call to match the alphabet of its context

$$R =_{df} \mu X \bullet \mathbf{var}\, x; \ldots; X_{+x}; \ldots; \mathbf{end}\, x$$

Apart from restoring syntactic correctness, the definition of alphabet extension also states that the value of the local variable x remains unchanged, even though the recursive call declares and manipulates a local variable with the same name. The constancy of the local variable at each level of recursion is essential to the correctness of recursively defined programs, and to reasoning about it.

The rule of alphabet extension for a recursive call places a strong duty on an implementation: it must reserve separate storage for local variables at each level of recursion. The storage may be released again on exit from that level of recursion, thereby permitting an efficient implementation of storage allocation by means of a stack.

Exercise 2.9.6

Prove that

$$\begin{aligned} (\mathbf{var}\, x)\, \mathbf{wp}\, r &= \forall x \bullet r \\ (\mathbf{end}\, x)\, \mathbf{wp}\, r &= r, \qquad \text{if } x \text{ is not free in } r \end{aligned}$$

□