Chapter 3

Designs

In this chapter, we work towards a more precise characterisation of the class of relations that are most useful in program design, namely those that are expressible (or at least implementable) in the limited notations of a particular programming language. As usual, we follow the standard practice of mathematics, which is to classify the basic concepts by their most important properties. For example, among the functions of real numbers, it is useful to single out those that are integrable, or continuous, or differentiable. A similar classification of the basic concept of a relation is essential to our goal of unifying theories of programming.

A subclass of formulae may be defined in a variety of ways. Sometimes it is done by a syntactic property, for example that a function can be expressed in a certain normal form using only a limited vocabulary of operators. Sometimes the definition requires satisfaction of a particular collection of algebraic laws. And sometimes the definition is by a general mathematical property: for example, a polynomial is a function whose higher derivatives all vanish. But the most useful definitions are those that are given in many different forms, together with a proof that all of them are equivalent.

The main goal of this chapter is to solve the paradox of non-termination presented in Section 2.6. We need therefore to define a subclass of relation P which can be proved to satisfy the zero laws

$$\mathbf{true}; P \;=\; \mathbf{true} \;=\; P; \mathbf{true}$$

Clearly this class must exclude the miraculous predicate **false**, which fails to satisfy these laws; indeed

$$\mathbf{true}; \mathbf{false} \;=\; \mathbf{false} \;=\; \mathbf{false}; \mathbf{true}$$

The easiest way to define the required subclass is to use the laws themselves as the defining property. Unfortunately the class of relations that satisfy the zero

laws themselves is too large, and does not have the right closure properties. It is necessary to define a slightly more restricted class of predicates by means of a slightly stronger collection of laws known as healthiness conditions. There are four laws in all, which will be numbered **H1**, **H2**, **H3** and **H4**.

Relations satisfying **H1** already satisfy the left zero law for **true**. Section 3.2 introduces the conditions **H3** and **H4**, and shows how the right zero law is also satisfied. This finally solves the outstanding problem raised in Section 2.6. Relations satisfying the first two laws can be split into two parts: an *assumption*, which a designer can assume will be satisfied before the program starts; and a *commitment*, which the program has to meet when it terminates. Such pairs of predicates are called *designs*; they are amenable to a calculus of refinement – essentially the same as that described by [14, 129, 130], or that used in the Vienna Development Method [101]. Section 3.1 shows that the definitions of these earlier calculi can be proved as theorems in the simpler calculus of relations. In this way, we achieve a unification of single-predicate theories of programming, like those based on B [7] or Z [171], with the familiar double-predicate theories of refinement; both of them can now be used interchangeably or in combination whenever this is most convenient.

The formalisation of the four healthiness conditions depends on a more explicit analysis of the phenomena of program initiation and termination, and this is what leads to a solution to the original problem of non-termination. We therefore introduce into the alphabet of our predicates a pair of Boolean variables to denote the relevant observations.

Definition 3.0.1 (ok and ok')

ok records the observation that the program has been started.

ok' records the observation that the program has terminated. Here, termination means proper normal termination, without error messages etc. □

If ok' is false, the program has not terminated and the final values of the program variables are unobservable: the predicate describing the program should make no prediction about these values. Similarly, if ok is false, the program has never started and even the initial values are unobservable. These considerations underlie the validity of the desired zero laws for sequential composition.

The variables ok and ok' are not global variables held in the store of any program, and it is assumed that they will never be mentioned in any expression or assignment of the program text. Furthermore, they will not be mentioned in any of the predicates featuring as assumptions or commitments; these are restricted to just the program variables, either in their dashed or undashed forms. However, the variables ok and ok' are *included* in the list of variables that are existentially quantified in the definition of sequential composition, and they are *included* in the list of universally quantified variables that are abbreviated by square brackets.

3.1 The refinement calculus

The purpose of the refinement calculus is to assist in the design of a complex software product. As indicated in Section 1.5, the complexity is mastered by splitting the overall task into well-defined separate subtasks, together with a proof (given in advance) that the assembly of components that fulfil the separate subtasks will meet the original overall goal. A design task is generally described by a pair of predicates: an *assumption* P which the designer can rely on when the program is initiated, and a *commitment* Q which must be true when the program terminates. The preconditions and postconditions of Section 2.8 are special cases of assumptions and commitments, but now we relax the restriction that forbids mention of dashed variables. The main achievement of the refinement calculus is to show how the assumptions made in one part of the design can be discharged by commitments made in other parts. Any outstanding assumptions are transmitted to a more global environment, or eventually to the user of the product. It is just this careful accounting of assumptions and commitments that enables a large team of engineers to collaborate successfully in the implementation of a large product.

There is one assumption that every program design must rely on, namely that the program will be started; that is that ok will be true. And there is one commitment that every design must make, namely that the program will terminate; that is that ok' will be true. If the assumption is violated, no constraint whatsoever is placed on the behaviour of the program: it may even fail to terminate. These insights permit a precise interpretation of the meaning of an assumption P and a commitment Q as parts of a single predicate describing the overall behaviour of the program. This predicate is

$$(ok \wedge P) \Rightarrow (ok' \wedge Q)$$

or in words "if the program starts in a state satisfying P, it will terminate, and on termination Q will be true".

The basic concept of a design in the refinement calculus deserves a notation of its own.

Definition 3.1.1 (Design)

Let P and Q be predicates not containing ok or ok'.

$$(P \vdash Q) =_{df} (ok \wedge P) \Rightarrow (ok' \wedge Q)$$

A *design* is a relation whose predicate is (or could be) expressed in this form. $\mathbf{D}$ will stand for the set of designs. □

In the interpretation of programs and specifications as single predicates, correctness (Section 1.5) is identified with implication. In the refinement calculus, the

corresponding ordering is known as refinement. The following theorem shows that the two orderings are the same. The notation $P[e, f/x, y]$ denotes the result of simultaneously substituting e for x and f for y in P.

Theorem 3.1.2

$$[(P_1 \vdash Q_1) \Rightarrow (P_2 \vdash Q_2)] \quad \textbf{iff} \quad [P_2 \Rightarrow P_1] \text{ and } [(P_2 \wedge Q_1) \Rightarrow Q_2]$$

Proof

$$\begin{aligned}
& [(P_1 \vdash Q_1) \Rightarrow (P_2 \vdash Q_2)] && \{\text{predicate calculus}\} \\
\equiv\ & [(P_1 \vdash Q_1)[true, false/ok, ok'] \Rightarrow (P_2 \vdash Q_2)[true, false/ok, ok']] \wedge \\
& [(P_1 \vdash Q_1)[true, true/ok, ok'] \Rightarrow (P_2 \vdash Q_2)[true, true/ok, ok']] \wedge \\
& [(P_1 \vdash Q_1)[false/ok] \Rightarrow (P_2 \vdash Q_2)[false/ok]] && \{\text{Def. 3.1.1}\} \\
\equiv\ & [\neg P_1 \Rightarrow \neg P_2] \wedge [(P_1 \Rightarrow Q_1) \Rightarrow (P_2 \Rightarrow Q_2)] && \{\text{predicate calculus}\} \\
\equiv\ & [P_2 \Rightarrow P_1] \wedge [(P_2 \wedge Q_1) \Rightarrow Q_2] && \square
\end{aligned}$$

The message of this theorem is that $(P_1 \vdash Q_1)$ is stronger because it has a weaker assumption P_1, and so it can be used more widely; furthermore, in all circumstances where $(P_2 \vdash Q_2)$ can be used, $(P_1 \vdash Q_1)$ has a stronger commitment, so its behaviour can be more readily predicted and controlled.

Equivalence of predicate pairs is defined in the normal way by mutual implication

$$\begin{aligned}
& [(P_1 \vdash Q_1) \equiv (P_2 \vdash Q_2)] \quad \textbf{iff} \\
& [(P_1 \vdash Q_1) \Rightarrow (P_2 \vdash Q_2)] \text{ and } [(P_2 \vdash Q_2) \Rightarrow (P_1 \vdash Q_1)]
\end{aligned}$$

It follows that all equivalent predicate pairs actually denote the same predicate. This gives a degree of freedom in the expression of the commitment, which can be strengthened or weakened in accordance with the equivalences

$$[(P \vdash Q) \equiv (P \vdash P \wedge Q)] \quad \text{and} \quad [(P \vdash Q) \equiv (P \vdash P \Rightarrow Q)]$$

In fact, these examples show that $P \wedge Q$ is the strongest and $P \Rightarrow Q$ is the weakest commitment predicate for which the equivalence holds. Any other commitment R that preserves equivalence must lie between them.

$$[(P \vdash Q) \equiv (P \vdash R)] \quad \textbf{iff} \quad [(P \wedge Q) \Rightarrow R] \text{ and } [R \Rightarrow (P \Rightarrow Q)]$$

In the extreme case, we have two alternative characterisations of **true**, namely

$$\textbf{false} \vdash \textbf{false} \quad \text{and} \quad \textbf{false} \vdash \textbf{true}$$

The definition of design already solves the first part of the paradox of Section 2.6; the left zero law is valid for all designs.

L1 $\mathbf{true}; (P \vdash Q) = \mathbf{true}$ (**true**-; left zero)

Proof

$$\begin{aligned} & \mathbf{true}; (P \vdash Q) && \{\text{def of ; and } \vdash\} \\ = \ & \exists ok^0, \ldots \bullet \mathbf{true} \wedge (ok^0 \wedge P \Rightarrow ok' \wedge Q) && \{\text{let } ok^0 = false\} \\ = \ & \mathbf{true} && \square \end{aligned}$$

All that remains is to show that every program can be expressed as a design. Unfortunately, this is not so. The problem arises right at the beginning, with our original definition of assignment. A new definition is needed, which recognises the role of *ok* as a precondition.

Definition 3.1.3 (Assignment)

$$x := e \ =_{df} \ (\mathbf{true} \vdash x' = e \wedge y' = y \wedge \ldots \wedge z' = z) \qquad \square$$

This definition can easily be generalised to solve the postponed problem of undefined expressions in assignments. For each expression e of a reasonable programming language, it is possible to calculate a condition $\mathcal{D}e$ which is true in just those circumstances in which e can be successfully evaluated [88, 131]. For example,

$$\begin{aligned} \mathcal{D}17 &= \mathcal{D}x = true \\ \mathcal{D}(e+f) &= \mathcal{D}e \wedge \mathcal{D}f \\ \mathcal{D}(e/f) &= \mathcal{D}e \wedge \mathcal{D}f \wedge (f \neq 0) \end{aligned}$$

Successful execution of an assignment relies on the assumption that the expression will be successfully evaluated, so we formulate our new definition of assignment

$$x := e \ =_{df} \ (\mathcal{D}e \vdash x' = e \wedge y' = y \wedge \ldots \wedge z' = z)$$

Expressed in words, this definition states that

either the program never starts ($ok = false$) and nothing can be said about its initial and final values,

or the initial values of the variables are such that evaluation of e fails ($\neg \mathcal{D}e$), and nothing can be said about the final values,

or the program terminates ($ok' = true$), and the value of x' is e, and the final values of all the other variables are the same as their initial values.

The definition of the conditional also needs to be modified to take into account the possibility that evaluation of the condition is undefined

$$P \lhd b \rhd Q \;=_{df}\; (\mathcal{D}b \Rightarrow (b \wedge P \vee \neg b \wedge Q))$$

However, in future we will maintain the simplifying assumption that all program expressions are everywhere defined. We return to the topic of $\mathcal{D}$ in Section 9.3.

The change in the definition of assignment requires us to give a new proof of the relevant algebraic laws.

L2 $(v := e\,;\, v := f(v)) \;=\; (v := f(e))$

L3 $v := e; (P \lhd b(v) \rhd Q) \;=\; (v := e; P) \lhd b(e) \rhd (v := e; Q)$

Proof of **L2**

$$\begin{array}{lll} & (v := e\,;\, v := f(v)) & \{\text{Def. 3.1.3}\} \\ = & (\neg ok \vee ok' \wedge (v' = e)); & \\ & (\neg ok \vee ok' \wedge (v' = f(v))) & \{2.4\mathbf{L6} \text{ and } \mathbf{L7}\} \\ = & \neg ok \vee (v' = e); (ok' \wedge (v' = f(v))) & \{\text{def of ;}\} \\ = & \neg ok \vee ok' \wedge (v' = f(e)) & \{\text{Def. 3.1.3}\} \\ = & v := f(e) & \Box \end{array}$$

The identity element $I\!I$ was defined as a special case of assignment $(v := v)$, and therefore needs a new definition

$$I\!I \;=_{df}\; (\mathbf{true} \vdash x' = x \wedge y' = y \wedge \ldots \wedge z' = z)$$

Fortunately, $I\!I$ is the left unit of sequential composition on designs.

L4 $I\!I\,;\, (P \vdash Q) \;=\; (P \vdash Q)$ ($I\!I-$; left unit)

The right unit law, however, is not valid for arbitrary designs; a solution to this problem is postponed to the next section.

The redefinition of assignment (Definition 3.1.3) is fortunately the only change that needs to be made to the definitions of Chapter 2. The normal combinators of the programming language have exactly the same meaning as operators on the single predicates as they have on the double predicates of the refinement calculus.

Theorem 3.1.4

(1) $(P_1 \vdash Q_1) \sqcap (P_2 \vdash Q_2) \;=\; (P_1 \wedge P_2 \vdash Q_1 \vee Q_2)$

(2) $(P_1 \vdash Q_1) \lhd b \rhd (P_2 \vdash Q_2) \;=\; (P_1 \lhd b \rhd P_2 \vdash Q_1 \lhd b \rhd Q_2)$

(3) $(P_1 \vdash Q_1); (P_2 \vdash Q_2) \;=\; (\neg(\neg P_1; \mathbf{true}) \wedge \neg(Q_1; \neg P_2) \vdash Q_1; Q_2)$

Proof of (3)
$$
\begin{aligned}
& (P_1 \vdash Q_1)\,;\,(P_2 \vdash Q_2) && \{\text{def of ;}\}\\
= \;& (P_1 \vdash Q_1)[\mathit{false}/ok']\,;\,(P_2 \vdash Q_2)[\mathit{false}/ok] \;\vee \\
& (P_1 \vdash Q_1)[\mathit{true}/ok']\,;\,(P_2 \vdash Q_2)[\mathit{true}/ok] && \{\text{Def. 3.1.1}\}\\
= \;& (\neg ok \vee \neg P_1)\,;\,\mathbf{true} \vee \\
& (\neg ok \vee \neg P_1 \vee Q_1)\,;\,(P_2 \vdash Q_2)[\mathit{true}/ok] && \{\text{2.4}\mathbf{L6}\text{ and }\mathbf{L7}\}\\
= \;& (\neg ok \vee \neg P_1)\,;\,(\mathbf{true} \vee (P_2 \vdash Q_2)[\mathit{true}/ok]) \vee \\
& Q_1\,;\,(P_2 \vdash Q_2)[\mathit{true}/ok] && \{\text{2.4}\mathbf{L6}\text{ and }\mathbf{L7}\}\\
= \;& \neg ok \vee (\neg P_1)\,;\,\mathbf{true} \vee \\
& (Q_1\,;\neg P_2) \vee ok' \wedge (Q_1\,;Q_2) && \{\text{Def. 3.1.1}\}\\
= \;& (\neg(\neg P_1\,;\mathbf{true}) \wedge \neg(Q_1\,;\neg P_2) \vdash Q_1\,;Q_2) && \square
\end{aligned}
$$

This theorem shows that all the combinators of the programming language map designs to designs. Since the primitive assignments have been redefined as designs, it follows that all predicates expressible as programs without recursion are also designs. This result will now be extended to recursive programs as well.

The law for disjunction (Theorem 3.1.4(1)) generalises to the union of arbitrary sets, and a similar law holds for arbitrary intersections.

Theorem 3.1.5

(1) $\prod_i (P_i \vdash Q_i) \;=\; (\bigwedge_i P_i) \vdash (\bigvee_i Q_i)$

(2) $\bigsqcup_i (P_i \vdash Q_i) \;=\; (\bigvee_i P_i) \vdash (\bigwedge_i (P_i \Rightarrow Q_i))$ $\square$

This means that designs form a complete lattice under implication ordering. Like all complete lattices, it contains a bottom element $\bot_{\mathbf{D}}$, which is $(\mathbf{false} \vdash \mathbf{true})$. It also has a top element

$$\top_{\mathbf{D}} \;=_{df}\; (\mathbf{true} \vdash \mathbf{false}) \;=\; \neg ok$$

This exactly describes a program that can never be started.

The really important property of a complete lattice is that it contains the weakest fixed point of any monotonic function. We have shown that all programming operators map designs to designs, and since the ordering of designs is the same as that of relations, the operators remain monotonic. It is this that justifies recursion in expressing designs and in developing their implementations, because it ensures that the result of the recursion will still be a design. That completes a demonstration that all programs are expressible as designs. As a result, they all satisfy the left zero law **L1**.

This appeal to Tarski's theorem gives an abstract proof of the validity of μ as a design notation. The following theorem gives an explicit way of calculating the assumption and commitment of a recursively defined design. As shown in Theorems 3.1.4 and 3.1.5, any monotonic function of designs, composed solely by lattice and programming operators, can be analysed as a pair of functions applied separately to the assumption and the commitment, for example

$$(F(P, Q) \vdash G(P, Q))$$

Here, F is monotonic in P and antimonotonic in Q, whereas for G it is the other way round. The weakest fixed point is given by a mutually recursive formula.

Theorem 3.1.6

$$\mu(X, Y) \bullet (F(X, Y) \vdash G(X, Y)) \;=\; (P(Q) \vdash Q)$$

where $P(Y) \;=\; \nu X \bullet F(X, Y)$

and $Q \;=\; \mu Y \bullet (P(Y) \Rightarrow G(P(Y), Y))$

Proof Here we only show that $(P(Q) \vdash Q)$ is a fixed point of the recursive equation

$$(X \vdash Y) \;=\; (F(X, Y) \vdash G(X, Y))$$

and leave to our readers the proof that the fixed point is the weakest.

$$\begin{array}{lll} & F(P(Q), Q) \vdash G(P(Q), Q) & \{\text{fixed point, def of } P\} \\ = & P(Q) \vdash G(P(Q), Q) & \{(P \vdash Q) \equiv (P \vdash (P \Rightarrow Q))\} \\ = & P(Q) \vdash (P(Q) \Rightarrow G(P(Q), Q)) & \{\text{fixed point, def of } Q\} \\ = & P(Q) \vdash Q & \Box \end{array}$$

Definition 3.1.3 and Theorems 3.1.4 to 3.1.6 of this section show that all programs of our language can be expressed solely in terms of predicate pairs, without ever translating them into single predicates by Definition 3.1.1. In a presentation of the refinement calculus for intending practitioners, these theorems are often presented as *definitions* of the notations of the programming language. There is no need then to introduce the variables ok and ok'. This overcomes a common philosophical objection to a variable like ok', whose value when false will never be observed. But this objection can be countered: similar encodings, like points at infinity, are common in mathematics and science, and they can be justified if they simplify the subsequent definitions, calculations and proofs. For exploration of unified theories, simplicity is paramount, though for practical application, the more complicated two-predicate definitions are more helpful. This section has shown how to get the best of both worlds.

Exercises 3.1.7

(1) Prove that $\top_{\mathbf{D}}; (P \vdash Q) = \top_{\mathbf{D}}$.

(2) Prove that $(x := e); \mathbf{true} = \mathbf{true}; (x := e) = \mathbf{true}$. □

3.2 Healthiness conditions

The previous section has made a start on defining an interesting subclass of predicates, namely those that can be written in the form

$$(ok \wedge P) \Rightarrow (ok' \wedge Q)$$

where P and Q do not contain ok or ok'. A trivial consequence is that all designs D satisfy

$$D = (ok \Rightarrow D)$$

From this, the left zero law follows trivially. A slightly less trivial consequence is satisfaction of the left unit law (3.1**L4**)

$$D = I\!I; D$$

In this section we explore these and additional conditions that can be placed on designs, to ensure that they satisfy additional desirable laws, such as the right unit law and the right zero law. By far the easiest way of doing this is to use the laws themselves to define the desired subclasses.

Definition 3.2.1 (Four healthiness conditions)

A predicate R is said to be **H1**, **H2**, **H3** and/or **H4** according to which of the following laws it satisfies.

H1 $R = (ok \Rightarrow R)$

H2 $[R[false/ok'] \Rightarrow R[true/ok']]$

H3 $R = R; I\!I$

H4 $R; \mathbf{true} = \mathbf{true}$ □

The trouble with such abstract definitions is that it is difficult to see what they are actually saying about programs or about the observations that can be made of program behaviour. Fortunately, each of the laws can be given an intuitive explanation, obtained often by expanding the definition of the operators. For example, **H1** is the simplest: it requires that the predicate R makes no prediction about the final values (or even the initial values) of the program variables until at least the program has started. That is reasonable, because these values are actually

impossible to observe: the understood condition for making the observation does not hold. The healthiness condition **H2** states formally that the predicate R is upward closed in the variable ok': as ok' changes from false to true, R cannot change from true to false. The semantic significance of the condition is not great: if R is a specification that under certain conditions allows failure to terminate, then R also allows an implementation which terminates under the same conditions. In other words, no specification can satisfy **H2** if it actually *requires* non-termination, so **H2** is a formal mathematical encoding of the fact that non-termination is something that is never wanted. Theorems 3.2.4 and 3.2.5 will give a similar interpretation of the semantic significance of **H3** and **H4**. The next two theorems describe the exact correspondence between **H1** and two of the laws that are definitely needed in any reasonable algebra for programming.

Theorem 3.2.2 (Algebraic characterisation of **H1**)

A predicate is **H1 iff** it satisfies the left zero and left unit laws.

Proof of ($\Leftarrow$)

$$
\begin{array}{lll}
 & R & \{\text{assumption : left unit law}\} \\
= & \mathit{II}; R & \{\text{def of } \mathit{II} \text{ and 2.4}\mathbf{L6}\} \\
= & \neg ok; R \vee \mathit{II}; R & \{\neg ok; \mathbf{true} = \neg ok \text{ and left unit}\} \\
= & \neg ok; \mathbf{true}; R \vee R & \{\text{assumption : left zero law}\} \\
= & \neg ok; \mathbf{true} \vee R & \{\neg ok; \mathbf{true} = \neg ok\} \\
= & \neg ok \vee R &
\end{array}
$$

($\Rightarrow$)

$$
\begin{array}{lll}
 & \mathbf{true}; R & \{R \text{ is } \mathbf{H1} \text{ and 2.4}\mathbf{L6}\} \\
= & \mathbf{true}; \neg ok \vee \mathbf{true}; R & \{\mathbf{true}; \neg ok = \mathbf{true}\} \\
= & \mathbf{true} &
\end{array}
$$

$$
\begin{array}{lll}
 & \mathit{II}; R & \{\text{def of } \mathit{II} \text{ and 2.4}\mathbf{L6}\} \\
= & \neg ok; R \vee (ok' \wedge v' = v); R & \{\neg ok; \mathbf{true} = \neg ok\} \\
= & \neg ok; \mathbf{true}; R \vee ok \wedge R & \{\mathbf{true}; R = \mathbf{true}\} \\
= & \neg ok; \mathbf{true} \vee ok \wedge R & \{\neg ok; \mathbf{true} = \neg ok,\ R \text{ is } \mathbf{H1}\} \\
= & R & \square
\end{array}
$$

The next theorem states the exact correspondence between the first two healthiness conditions and the syntactic definition (Definition 3.1.1) of a design as a pair of predicates.

Theorem 3.2.3 (Healthiness of designs)

A predicate is **H1** and **H2 iff** it is a design.

Proof of ($\Rightarrow$)

$$
\begin{array}{lll}
 & R & \{R \text{ satisfies } \mathbf{H1}\} \\
= & \neg ok \vee R & \{\text{predicate calculus}\} \\
= & \neg ok \vee (\neg ok' \wedge R[false/ok']) \vee & \\
 & (ok' \wedge R[true/ok']) & \{R \text{ satisfies } \mathbf{H2}\} \\
= & \neg ok \vee R[false/ok'] \vee R[true/ok'] \wedge ok' & \{\text{Def. 3.1.1}\} \\
= & \neg R[false/ok'] \vdash R[true/ok'] &
\end{array}
$$

Finally we need to prove that designs satisfy **H2**.

$$
\begin{array}{lll}
 & (P \vdash Q)[false/ok'] & \{\text{Def. 3.1.1}\} \\
= & \neg ok \vee \neg P & \{\text{predicate calculus}\} \\
\Rightarrow & \neg ok \vee \neg P \vee Q & \{\text{Def. 3.1.1}\} \\
= & (P \vdash Q)[true/ok'] & \Box
\end{array}
$$

The general definition of the assumption in a design allows it to contain dashed variables as well as undashed variables. This is a freedom which it would be better to forego, because there is no way in which such an assumption could be discharged by other components in the program, whether they are executed previously or subsequently. And a dashed variable in an assumption makes the implementation too easy. All that is needed is to find some final value for the dashed variables that makes the assumption false, and then the specification will be trivially satisfied – but probably not in the desired way.

None of these problems arise if the assumption is a precondition, containing only undashed variables. Then the responsibility for making the assumption true can be discharged by the preceding segment of program. This sensible restriction, observed in all current program calculi, corresponds exactly to the third healthiness condition.

Theorem 3.2.4 (Assumption and precondition)

A design $P \vdash Q$ is **H3 iff** its assumption P can be expressed as a condition.

Proof

$$
\begin{array}{lll}
 & (P \vdash Q) = (P \vdash Q); I\!I & \{\text{Theorem 3.1.4}\} \\
\equiv & (P \vdash Q) = (\neg(\neg P; \mathbf{true}) \vdash Q) & \{\text{Theorem 3.1.2}\} \\
\equiv & \neg P = (\neg P); \mathbf{true} & \{\text{predicate calculus}\} \\
\equiv & P = P; \mathbf{true} & \Box
\end{array}
$$

A significant benefit of **H3** is that it permits a simplification of Theorem 3.1.4(3), replacing $\neg(\neg P_1; \mathbf{true})$ simply by P_1.

If the precondition of a design $P \vdash Q$ is satisfied, the eventual program is required to terminate and deliver final values for the program variables, and these

must satisfy the predicate Q. But that will be logically impossible if there are no final values which satisfy Q. This paradox is precluded by **H4**, which states that for *any* initial values of the undashed variables that satisfy P, there exist final values for the dashed variables that satisfy Q.

Theorem 3.2.5 (Feasibility)

$P \vdash Q$ satisfies **H4 iff** $[\exists ok', x', \ldots, z' \bullet (P \vdash Q)]$.

Proof Expand Definition 2.2.1 of ;. □

It is the condition **H4** that excludes the miraculous predicate $\top_{\mathbf{D}}$. **H4** is called a *feasibility* condition; all programs will be proved to satisfy it, and furthermore if a design fails to satisfy it, there is no program that could ever implement that design.

Exercises 3.2.6

(1) Prove that sequential composition, non-deterministic and conditional choices preserve the healthiness conditions.

(2) A design $b(v) \vdash Q(v, v')$ is *predeterministic* if

$$[(b(v) \wedge Q(v, v_1) \wedge Q(v, v_2)) \Rightarrow (v_1 = v_2)]$$

Prove that if both R and S are predeterministic, so are $R; S$ and $R \lhd b \rhd S$.

(3) Define a condition b to be *stable* if

$$b = b \wedge ok$$

Restrict R to healthy predicates and b to stable conditions, and prove that

$$R \textbf{ wp } b \text{ is stable}$$

and that it obeys Dijkstra's healthiness condition

$$R \textbf{ wp } false = false \qquad \text{(absence of miracle)} \ \square$$