Appendix B

Answers to some exercises

Those adopting the book for teaching may obtain a complete set of answers from the author.

Chapter 1

Answer 1.1 (p.13) The number of refinement steps has nothing to do with closeness to code. *Refinement* is a relation between the meanings of programs; *code* is a description of the way in which programs can be written. Here is a refinement '*from* code':

$$\textbf{abort} \quad \sqsubseteq \quad y\colon \left[0 \le x \le 9\ ,\ y^2 = x\right].$$

Answer 1.2 (p.13) The new specification is the following:

$$y\colon \left[x \le 9\ ,\ (x < 0 \Rightarrow y = 0) \wedge (x \ge 0 \Rightarrow y^2 = x)\right].$$

Note that the original could abort if $x < 0$ initially.

Answer 1.4 (p.13) The valid ones are 1, 4, 5 and 7. (But you need *strengthen postcondition* 5.1 to show number 7.)

Answer 1.7 (p.14) On the left, the client cannot assume that x will not change; on the right, he can. But what is wrong with the following counter argument?

> On the right, the client cannot assume that x will change; on the left, he can.

Answer 1.8 (p.14)

$$
\begin{array}{ll}
& x \leq 9 \Rightarrow (\exists\, y : \mathbb{R} \cdot (x < 0 \Rightarrow y = 0) \wedge (x \geq 0 \Rightarrow y^2 = x)) \\
\Lleftarrow & \text{“Exercise 2.9”} \\
& (\exists\, y : \mathbb{R} \cdot (x < 0 \Rightarrow y = 0) \wedge (x \geq 0 \Rightarrow y^2 = x)) \\
\Lleftarrow & \text{“Exercise 2.11”} \\
& (x < 0 \Rightarrow (\exists\, y : \mathbb{R} \cdot y = 0)) \wedge (x \geq 0 \Rightarrow (\exists\, y : \mathbb{R} \cdot y^2 = x)) \\
\equiv & \mathsf{true} \wedge (x \geq 0 \Rightarrow (\exists\, y : \mathbb{R} \cdot y^2 = x)) \\
\equiv & \text{“property of } \mathbb{R}\text{”} \\
& \mathsf{true}\ .
\end{array}
$$

Answer 1.10 (p.14) From *feasibility* 1.4, we must prove that

$$x \geq 0 \quad \Rrightarrow \quad \left(\exists\, y : \mathbb{R} \cdot y^2 = x \wedge y > 0\right).$$

But that is not true, since the right-hand side is equivalent to $x > 0$.

Informally, the program, when $x = 0$, must establish $y = 0$ (because $0^2 = 0$) and $y > 0$ (in the postcondition) simultaneously.

Answer 1.11 (p.14) It is feasible (false implies anything); it is never guaranteed to terminate. Hence it is **abort**. That it can change w is suggested by the remark on p.12 concerning the behaviour of **abort**. See also Exercise 8.8.

Answer 1.12 (p.14) Executing a false assumption causes the program to abort, and that does change the program (unless it would have aborted anyway). But a false assumption can be placed at a point which is never executed; there, it has no effect (and little value).

Answer 1.13 (p.14) Assumptions may be weakened; the program is refined because it assumes less.

Answer 1.14 (p.14)

$$
\begin{array}{ll}
& \{pre'\}\ \{pre\} \\
= & \text{“}\textit{assumption}\ 1.6\text{”} \\
& \{pre'\};\ : [pre\ ,\ \mathsf{true}] \\
= & \text{“}\textit{absorb assumption}\ 1.8\text{”} \\
& : [pre' \wedge pre\ ,\ \mathsf{true}] \\
= & \text{“}\textit{assumption}\ 1.6\text{”} \\
& \{pre' \wedge pre\}\ .
\end{array}
$$

Answer 1.17 (p.15) Remember that anything is refined (trivially) by itself.

Answer 1.18 (p.15) The law *strengthen postcondition* 1.1 requires $post \Rrightarrow \mathsf{true}$; the law *weaken precondition* 1.2 requires $\mathsf{false} \Rrightarrow pre$. Both hold.

Chapter 2

Answer 2.3 (p.25) The propositional formulae are 1 and 7. Number 2 is a variable; number 3 is an English word. Number 6 is a statement about two simple formulae.

Answer 2.5 (p.26)

1. $(\forall\, i : \mathbb{Z} \bullet \mathsf{even}\, i \vee \mathsf{odd}\, i)$
2. $(\forall\, m : \mathbb{N} \bullet \mathsf{odd}\, m \Rightarrow (\exists\, n : \mathbb{N} \bullet \mathsf{even}\, n \wedge m = n + 1))$
3. $(\exists\, i : \mathbb{Z} \bullet \mathsf{even}\, i \wedge \neg\, (\exists\, n : \mathbb{N} \bullet \mathsf{odd}\, n \wedge i = n + 1))$
4. $(\forall\, n : \mathbb{N} \bullet 0 \leq n)$
5. $\neg\, (\exists\, i : \mathbb{Z} \bullet (\forall\, j : \mathbb{Z} \bullet i \leq j))$
6. $(\forall\, r : \mathbb{R} \bullet r > 0 \Rightarrow (\exists\, s : \mathbb{R} \bullet 0 < s < r))$

Answer 2.6 (p.26) $(\exists\, y \bullet (\forall\, x \bullet \mathcal{A} \Rightarrow x = y))$.

Answer 2.8 (p.26) We have for the first formula

$\mathcal{A}$	$\mathcal{B}$	$\mathcal{A}$	$\Rightarrow$	$(\mathcal{B}$	$\Rightarrow$	$\mathcal{A})$
true	true	true	true	true	true	true
true	false	true	true	false	true	true
false	true	false	true	true	false	false
false	false	false	true	false	true	false
			$\uparrow$			

The indicated column is all true.

The other formulae are done similarly.

Answer 2.9 (p.26) Exercise 2.8 showed that $\mathcal{A} \Rightarrow (\mathcal{B} \Rightarrow \mathcal{A})$ is true in all states; that's what $\mathcal{A} \Rrightarrow \mathcal{B} \Rightarrow \mathcal{A}$ means.

Answer 2.10 (p.26)

$$
\begin{array}{ll}
 & (\exists\, x \bullet (\mathcal{A} \Rightarrow \mathcal{B}) \wedge (\neg \mathcal{A} \Rightarrow \mathcal{C})) \\
\equiv & \text{“Predicate law A.38”} \\
 & (\exists\, x \bullet \mathcal{A} \wedge \mathcal{B} \vee \neg \mathcal{A} \wedge \mathcal{C}) \\
\equiv & \text{“Predicate law A.66”} \\
 & (\exists\, x \bullet \mathcal{A} \wedge \mathcal{B}) \vee (\exists\, x \bullet \neg \mathcal{A} \wedge \mathcal{C}) \;\; .
\end{array}
$$

Answer 2.12 (p.26)

$$
\begin{array}{ll}
 & (\exists\, a \bullet (\forall\, b \bullet \mathcal{A})) \\
\equiv & \text{“Predicate law A.74”} \\
 & (\forall\, b \bullet (\exists\, a \bullet (\forall\, b \bullet \mathcal{A})))
\end{array}
$$

$\Rrightarrow$ "Predicate law A.86"
$(\forall\, b \bullet (\exists\, a \bullet \mathcal{A}))$.

The converse is not true. (Try it.)

Chapter 3

Answer 3.3 (p.34)

$$
\begin{array}{lll}
 & x\colon [x = X \ ,\ x = X^4] & \\
\sqsubseteq & \text{“}\textit{sequential composition}\ 3.3\text{”} & \\
 & x\colon [x = X \ ,\ x = X^2]\,; & \text{(i)} \\
 & x\colon [x = X^2 \ ,\ x = X^4] & \text{(ii)} \\
\text{(i)}\ \sqsubseteq & x := x^2 & \\
\text{(ii)}\ \sqsubseteq & x := x^2\ . &
\end{array}
$$

Answer 3.6 (p.34)

$$
\begin{array}{lll}
 & x, y, t\colon [x = X \wedge y = Y \ ,\ x = Y \wedge y = X] & \\
\sqsubseteq & \text{“}\textit{sequential composition}\ 3.3\text{”} & \\
 & x, y, t\colon [x = X \wedge y = Y \ ,\ t = Y \wedge y = X]\,; & \\
 & x, y, t\colon [t = Y \wedge y = X \ ,\ x = Y \wedge y = X] & \\
\sqsubseteq & \text{“}\textit{assignment}\ 1.3\text{”} & \\
 & x, y, t\colon [x = X \wedge y = Y \ ,\ t = Y \wedge y = X]\,; & \lhd \\
 & x := t & \\
\sqsubseteq & \text{“}\textit{sequential composition}\ 3.3\text{”} & \\
 & x, y, t\colon [x = X \wedge y = Y \ ,\ x = X \wedge t = Y]\,; & \\
 & x, y, t\colon [x = X \wedge t = Y \ ,\ t = Y \wedge y = X] & \\
\sqsubseteq & \text{“}\textit{assignment}\ 1.3\text{”} & \\
 & t := y; & \\
 & y := x\ . &
\end{array}
$$

Answer 3.7 (p.35)

$$
\begin{array}{ll}
 & x, y, t := y, x, ? \\
\sqsubseteq & \text{“}\textit{open assignment}\ 3.1\text{”} \\
 & x, y, t := y, x, y \\
= & \text{“aim for leading } t := y\text{”} \\
 & x, y, t := t[t \backslash y], x, y \\
= & \text{“}\textit{leading assignment}\ 3.6\text{”} \\
 & t := y; \\
 & x, y := t, x
\end{array}
$$

$$\begin{array}{ll} = & \text{“special case of Law 3.6; } t \text{ contains no } y\text{”} \\ & t := y; \\ & y := x; \\ & x := t \ . \end{array}$$

Answer 3.8 (p.35)

$$\begin{array}{lll} & w, x\colon [\mathit{pre}\ ,\ \mathit{post}] & \\ \sqsubseteq & \text{“\textit{sequential composition} 3.3”} & \\ & w, x\colon [\mathit{pre}\ ,\ \mathit{post}[x \backslash E]]\,; & \\ & w, x\colon [\mathit{post}[x \backslash E]\ ,\ \mathit{post}] & \triangleleft \\ \sqsubseteq & x := E\ . & \end{array}$$

Chapter 4

Answer 4.1 (p.39)

$$\begin{array}{llr} & x := \mathsf{abs}\, y & \\ \sqsubseteq & \mathbf{if}\ y \geq 0 \rightarrow x\colon [y \geq 0\ ,\ x = \mathsf{abs}\, y] & \text{(i)} \\ & [\!]\ \ y \leq 0 \rightarrow x\colon [y \leq 0\ ,\ x = \mathsf{abs}\, y] & \text{(ii)} \\ & \mathbf{fi} & \\ \text{(i)}\ \sqsubseteq & x := y & \\ \text{(ii)}\ \sqsubseteq & x := -y\ . & \end{array}$$

Answer 4.4 (p.39) Write *prog* as $w\colon [\mathit{pre}\ ,\ \mathit{post}]$, and use *absorb assumption* 1.8 everywhere. Then use *weaken precondition* 1.2 and the fact that $\mathcal{A} \Rrightarrow \mathcal{B}$ implies $\mathcal{A} \Rrightarrow \mathcal{A} \wedge \mathcal{B}$.

Chapter 5

Answer 5.2 (p.47) Let I abbreviate the invariant $n \neq 0 \wedge (\mathsf{pt}\, N \Leftrightarrow \mathsf{pt}\, n)$. The development then continues

$$\begin{array}{lll} \sqsubseteq & n\colon [I\ ,\ I \wedge \neg(2 \mid n)] & \\ \sqsubseteq & \text{“invariant } I\text{, variant } n\text{”} & \\ & \mathbf{do}\ 2 \mid n \rightarrow & \\ & \qquad n\colon [2 \mid n\ ,\ n \neq 0 \wedge (\mathsf{pt}\, N \Leftrightarrow \mathsf{pt}\, n)\ ,\ 0 \leq n < n_0] & \triangleleft \\ & \mathbf{od} & \end{array}$$

$\sqsubseteq\ n := n \div 2\ .$

It is the same code as before.

Answer 5.3 (p.47) The laws used are *sequential composition* 3.3 and *strengthen postcondition* 1.1. Continuing,

$$
\begin{array}{ll}
\text{(i)} & \sqsubseteq\ \text{“}\textit{assignment}\ 1.3\text{”} \\
& f := 1 \\
\text{(ii)} & \sqsubseteq\ \text{“invariant } f \times n! = F \text{, variant } n\text{”} \\
& \mathbf{do}\ n \neq 0 \rightarrow \\
& \qquad n, f\colon [n \neq 0\ ,\ f \times n! = F\ ,\ 0 \le n < n_0] \qquad \triangleleft \\
& \mathbf{od} \\
& \sqsubseteq\ \text{“}\textit{following assignment}\ 3.5\text{”} \\
& n, f\colon [n \neq 0 \land f \times n! = F\ , \\
& \quad f \times (n-1)! = F \land 0 \le n - 1 < n_0]\,; \qquad \triangleleft \\
& n := n - 1 \\
& \sqsubseteq\ \text{“}\textit{contract frame}\ 5.4\text{”} \\
& f\colon [n \neq 0 \land f \times n! = F\ ,\ f \times (n-1)! = F] \\
& \sqsubseteq\ \text{“}\textit{weaken precondition}\ 1.2\text{”} \\
& f\colon [f \times n \times (n-1)! = F\ ,\ f \times (n-1)! = F] \\
& \sqsubseteq\ \text{“}\textit{assignment}\ 1.3\text{”} \\
& f := f \times n\ .
\end{array}
$$

Answer 5.4 (p.48) First we strengthen the postcondition, then we weaken the precondition. The conditions of *strengthen postcondition* 5.1 require

$$
\begin{array}{ll}
& (0 \le x \le 9)[y \backslash y_0] \land (0 \le x \Rightarrow y^2 = x) \\
\Rrightarrow & y^2 = x\ .
\end{array}
$$

The law *weaken precondition* 1.2 requires the trivial $0 \le x \le 9 \Rrightarrow \mathsf{true}$. Together they give the derivation

$$
\begin{array}{l}
y\colon [0 \le x \le 9\ ,\ y^2 = x] \\
\sqsubseteq\ \text{“}\textit{strengthen postcondition}\ 5.1\text{”} \\
y\colon [0 \le x \le 9\ ,\ 0 \le x \Rightarrow y^2 = x] \\
\sqsubseteq\ \text{“}\textit{weaken precondition}\ 1.2\text{”} \\
y\colon [y^2 = x]\ .
\end{array}
$$

Why is the order of the laws (5.1 before 1.2) important?

Answer 5.5 (p.48)

1. Use *assignment* 5.2: the condition is $y > x_0 \Rrightarrow y > x_0$.
2. Use *assignment* 5.2: the condition is $x_0 < 0 \Rrightarrow -x_0 > x_0$.

3. Use *assignment* 5.2: the condition is $\mathsf{true} \Rrightarrow y_0 = y_0 \wedge x_0 = x_0$.
4. Use *assignment* 5.2: the condition is $x_0 = X + 1 \Rrightarrow x_0 + 1 = X + 2$.
5. $$\begin{array}{ll} & x\colon [x = X + 1\ ,\ x = X + 2] \\ \sqsubseteq & \text{"}\textit{strengthen postcondition}\ 5.1\text{"},\ x_0 = X + 1 \wedge x = x_0 + 1 \ \Rrightarrow\ x = X + 2 \\ & x\colon [x = X + 1\ ,\ x = x_0 + 1] \\ \sqsubseteq & \text{"}\textit{weaken precondition}\ 1.2\text{"} \\ & x\colon [x = x_0 + 1]\ . \end{array}$$
6. $$\begin{array}{ll} & x\colon [x = x_0 + 2] \\ \sqsubseteq & \mathbf{con}\ X \bullet \\ & x\colon [x = x_0 + 1]\,; \\ & x\colon [x = X + 1\ ,\ x = X + 2] \qquad \triangleleft \\ \sqsubseteq & \text{"as in 5 above"} \\ & x\colon [x = x_0 + 1]\ . \end{array}$$

   This required the more advanced *sequential composition* 8.4. What did you use?

Answer 5.6 (p.48) The law would be as follows:

Law B.1 <u>initialised iteration</u> Provided *inv* contains no initial variables,

$$\begin{array}{ll} & w\colon [\mathit{pre}\ ,\ \mathit{inv} \wedge \neg G] \\ \sqsubseteq & w\colon [\mathit{pre}\ ,\ \mathit{inv}]\,; \\ & \mathbf{do}\ G \rightarrow w\colon [G \wedge \mathit{inv}\ ,\ \mathit{inv} \wedge (0 \le V < V_0)]\ \mathbf{od}\ . \end{array}$$

□

The expression V is the variant.

Answer 5.8 (p.49) Conjuncts (5.8) and (5.9) of the invariant are sufficient to ensure requirement (5.7) of a fair handing-out, taking into account that the iteration body is executed exactly C times.

For requirement (5.6) it would be enough to know that $s = 0$ on termination, since conjunct (5.10) of the invariant then suffices. Unfortunately, the negated guard gives us $c = 0$, not $s = 0$.

We choose a slightly different invariant therefore, to get around that problem. Since by the quoted properties of $\lfloor\ \rfloor$ we have

$$\lfloor S/C \rfloor \le \lfloor s/c \rfloor \equiv \lfloor S/C \rfloor \le s/c\ ,$$

and similarly for (5.9), we replace (5.8) and (5.9) by the single

$$c \times \lfloor S/C \rfloor \le s \le c \times \lceil S/C \rceil\ . \tag{B.1}$$

Then $c = 0$ (from termination of the iteration) together with (B.1), gives us $s = 0$, which is what we needed.

Now we must consider maintenance of the invariant. Conjunct (5.10) is maintained by the combination of 'hand out t' and $s := s - t$ in the body. For (B.1) we must show that the iteration body refines

$$c, s, t\colon [(B.1)\ ,\ (B.1)]\ . \tag{i}$$

Following assignment gives us

$$\begin{array}{ll} \text{(i)} \sqsubseteq & t\colon [(B.1)\ ,\ (c-1)\times\lfloor S/C\rfloor \le s-t \le (c-1)\times\lceil S/C\rceil\ ] \qquad \text{(ii)} \\ & c, s := c-1, s-t \end{array}$$

and, leaving aside the 'hand out', we could strengthen the postcondition to reach

$$\text{(ii)} \sqsubseteq\ t\colon [\ \lfloor s/c\rfloor \le t \le \lceil s/c\rceil\ ]$$

provided

$$\begin{array}{ll} & \lfloor s/c\rfloor \le t \le \lceil s/c\rceil \\ \wedge & c\times\lfloor S/C\rfloor \le s \le c\times\lceil S/C\rceil \\ \Rrightarrow & (c-1)\times\lfloor S/C\rfloor \le s-t \le (c-1)\times\lceil S/C\rceil\ . \end{array} \tag{B.2}$$

Claim (B.2) is really the core of the problem; it is the key mathematical fact, finally extracted from the details of programming. To finish off, consider the following:

$$\begin{array}{ll} & (c-1)\times\lfloor S/C\rfloor \le s-t \\ \Lleftarrow & \text{“}t \le \lceil s/c\rceil \text{ in antecedent”} \\ & (c-1)\times\lfloor S/C\rfloor \le s-\lceil s/c\rceil \\ \equiv & \lceil s/c\rceil \le s-(c-1)\times\lfloor S/C\rfloor \\ \equiv & \text{“property of } \lceil\ \rceil\text{”} \\ & s/c \le s-(c-1)\times\lfloor S/C\rfloor \\ \equiv & \text{“iteration guard ensures } c\neq 0\text{”} \\ & (c-1)\times\lfloor S/C\rfloor \le (c-1)\times s/c \\ \Lleftarrow & \text{“even when } c=1!\text{”} \\ & \lfloor S/C\rfloor \le s/c \\ \equiv & \text{“}c\neq 0 \text{ again”} \\ & c\times\lfloor S/C\rfloor \le s\ , \end{array}$$

which is given to us in the antecedent of Claim (B.2): it is part of the invariant. The other inequality is handled similarly.

This has been a simple problem, but not an easy one; and that is often the case. Finding an invariant, and then discovering an inexorable proof that it is the *correct* invariant, can take a very long time. In spite of that, one should aim to make the resulting reasoning so clear and polished that its *checking* is very short. Then — and only then — is the correctness of the program finally 'obvious'.

Chapter 6

Answer 6.3 (p.61) It follows from $n \in \mathbb{N} \wedge z = n \Rrightarrow z \geq 0$.

Answer 6.5 (p.61) We would use **var** $c : \mathbb{N}$; **and** $c = 3$. The remaining difference is that we can assign to such constants, and that is feasible as long as the value assigned is the value it already has. That cannot be checked automatically by a compiler, so conventional languages allow **const** declarations in code only by prohibiting assignments to them altogether.

Answer 6.6 (p.61) Yes. Use of the law *assignment* 1.3 requires $\mathsf{true} \wedge n \in \mathbb{N} \Rrightarrow (-1)^2 = 1$. The assignment is infeasible, however, and in Section 6.6 that was called ill-typed.

Answer 6.7 (p.61)

1. Infeasible: z might be negative.
2. Feasible: $\mathbb{N} \subseteq \mathbb{Z}$.
3. Feasible.
4. Infeasible: even if positive, an integer need not have an integral square root.
5. Feasible: If c is a complex root of unity, then c and $1/c$ are conjugates.

Answer 6.10 (p.62) Unfold the iteration *twice*, as shown on p.42. The first execution of the body establishes $E < e$; the second requires $E = e$, and so can abort (which it shouldn't).

Chapter 7

Answer 7.2 (p.66) See Figure 19.1.

Answer 7.4 (p.66) The formula $p < q$ is implied by the previous precondition. The 'increasing variant' was originally $(q - r) < (q_0 - r_0)$; contracting the frame gave $(q - r) < (q - r_0)$; then strengthening the postcondition gave $r_0 < r$.

Chapter 8

Answer 8.2 (p.72) It is unnecessary because if x is not in the frame, its initial and final values are equal. Here is the proof of equality:

$$w\colon [pre\ ,\ post]$$

$$
\begin{array}{ll}
= & \text{“}\mathit{expand\ frame}\ 8.3\text{”} \\
 & w, x \colon [\mathit{pre}\ ,\ \mathit{post} \wedge x = x_0] \\
= & w, x \colon [\mathit{pre}\ ,\ \mathit{post}[x_0 \backslash x] \wedge x = x_0] \\
= & \text{“}\mathit{expand\ frame}\ 8.3\text{”} \\
 & w \colon [\mathit{pre}\ ,\ \mathit{post}[x_0 \backslash x]]\ .
\end{array}
$$

Answer 8.4 (p.73) Note first that $pre[w \backslash w_0] \Rrightarrow (\exists\, w \cdot pre)$, from Predicate law A.87. Then we have

$$
\begin{array}{ll}
 & w \colon [\mathit{pre}\ ,\ \mathit{post}] \\
\sqsubseteq & \text{“}\mathit{strengthen\ postcondition}\ 5.1\text{”} \\
 & w \colon [\mathit{pre}\ ,\ (\exists\, w \cdot \mathit{pre}) \Rightarrow \mathit{post}] \\
\sqsubseteq & \text{“}\mathit{strengthen\ postcondition}\ 1.1\text{”} \\
 & w \colon [\mathit{pre}\ ,\ (\exists\, w \cdot \mathit{pre}) \wedge \mathit{post}] \\
\sqsubseteq & \text{“}\mathit{strengthen\ postcondition}\ 5.1\text{”} \\
 & w \colon [\mathit{pre}\ ,\ \mathit{post}]\ .
\end{array}
$$

Answer 8.5 (p.73) The result is this law:

Law B.2 <u>sequential composition</u>

$$
\begin{array}{ll}
 & w, x \colon [\mathit{pre}\ ,\ \mathit{post}] \\
\sqsubseteq & x \colon [\mathit{pre}\ ,\ \mathit{mid}]\ ; \\
 & w, x \colon [\mathit{mid}\ ,\ \mathit{post}]\ .
\end{array}
$$

The formula *mid* must not contain initial variables; and *post* must not contain x_0.
□

Law B.2 allows initial variables w_0 in post, since the first command $x \colon [\mathit{pre}\ ,\ \mathit{mid}]$ in the composition does not change them. In contrast, *sequential composition* 3.3 allows no initial variables at all.

Answer 8.7 (p.73)

$$
\begin{array}{lll}
 & w \colon [\mathsf{true}\ ,\ \mathsf{false}] & \\
\sqsubseteq & \text{“}\mathit{contract\ frame}\ 5.4\text{”} & \\
 & \colon [\mathsf{true}\ ,\ \mathsf{false}] & = \mathbf{magic} \\
\sqsubseteq & \text{“}\mathit{expand\ frame}\ 8.3\text{”} & \\
 & w \colon [\mathsf{true}\ ,\ \mathsf{false} \wedge w = w_0] & \\
\sqsubseteq & w \colon [\mathsf{true}\ ,\ \mathsf{false}]\ . &
\end{array}
$$

Chapter 9

Answer 9.2 (p.91)

1. $\{n : \mathbb{N} \bullet n^2\}$
2. $\left\{i, j, k : \mathbb{N} \bullet 2^i 3^j 4^k\right\}$
3. $\mathbb{N}^{++} - \{i, j : \mathbb{N}^{++} \bullet i \times j\}$, where $\mathbb{N}^{++} \;\widehat{=}\; \{k : \mathbb{N} \mid k \geq 2\}$
4. $\{c : \mathbb{C} \mid c^n = 1\}$

Answer 9.3 (p.91) Using the alternative definition, the promoted relation is transitive if the original relation is. But this law, for example, no longer holds:

$$s1 \odot s2 \wedge s1' \subseteq s1 \wedge s2' \subseteq s2 \quad \not\Rrightarrow \quad s1' \odot s2' \; .$$

We retain the original definition, therefore.

Answer 9.6 (p.91)

$$\begin{array}{rcl}
\{\mid \mathsf{true} \bullet x\} & = & \{x\} \\
\{\mid \mathsf{false} \bullet x\} & = & \{\} \\
\langle i : 0 \rightarrow n \bullet x\rangle & = & \underbrace{\langle x, x, \cdots, x\rangle}_{n \text{ times}} .
\end{array}$$

Answer 9.7 (p.91) The definition is $\prod q \;\widehat{=}\; (\times x : q)$. The empty product $\prod\langle\rangle$ is 1, because 1 is the identity of multiplication.

Answer 9.10 (p.91) It gives the size of the set. The same applies to bags and sequences.

Answer 9.16 (p.92) The placement of the well-definedness assumption allows abortion explicitly in any case in which $as[i]$ is 'undefined', and there are two cases: when the iteration is encountered for the first time; and when the guard is re-evaluated after an execution of the iteration body.

For the refinement, we have

$$\begin{array}{lll}
 & i\colon [a \in as \;,\; a = as[i]] & \\
 & \sqsubseteq \text{“}I \;\widehat{=}\; a \in as{\downarrow}i\text{”} & \\
 & i\colon [a \in as \;,\; I]\,; & \text{(i)} \\
 & i\colon [I \;,\; I \wedge a = as[i]] & \text{(ii)} \\
\text{(i)} & \sqsubseteq \text{“}w\colon [\mathit{pre} \;,\; \mathit{post}] \;\sqsubseteq\; w\colon [\mathit{pre} \;,\; \mathit{post}]\,; \{\mathit{post}\}\text{”} & \\
 & i := 0; & \\
 & \{I\} & \triangleleft \\
 & \sqsubseteq \text{“}\mathit{weaken\ precondition}\ 1.2\text{”} & \\
 & \{i < N\} &
\end{array}$$

$$
\begin{array}{lll}
\text{(ii)} & \sqsubseteq & \textbf{do } a \neq as[i] \rightarrow \\
& & \qquad i\colon [a \neq as[i]\ ,\ I\ ,\ i_0 < i \leq N] \qquad \triangleleft \\
& & \textbf{od} \\
& \sqsubseteq & \text{“as above”} \\
& & i\colon [a \neq as[i]\ ,\ I\ ,\ i_0 < i \leq N] \qquad \text{(iii)} \\
& & \{I\} \qquad \triangleleft \\
& \sqsubseteq & \{i < N\} \\
\text{(iii)} & \sqsubseteq & i := i + 1\ .
\end{array}
$$

Although some of the above manipulations of assumptions do not follow easily from the laws we have so far, they should be intuitively plausible: since $w\colon [pre\ ,\ post]$ establishes $post$, one can sequentially compose on the right with $\{post\}$, which behaves like **skip** in those circumstances. A 'nice' formal derivation of that step would be

$$
\begin{array}{ll}
& i\colon [a \in as\ ,\ a \in as{\downarrow}i] \\
\sqsubseteq & \{a \in as{\downarrow}0\}\ i := 0 \\
\sqsubseteq & \text{“}\textit{advance assumption}\text{ 22.2 backwards”} \\
& i := 0; \\
& \{a \in as{\downarrow}i\}\ ,
\end{array}
$$

using a law postponed until Chapter 22 (though not a very complicated one).

Evaluation of $as[i]$ when $i = N$ is 'now' acceptable because the assumptions $\{i < N\}$ abort in that case anyway, and so it matters not at all what the implementation might do with $as[N]$.

The general rule relation between invariants and possibly 'undefined' iteration guards is that the invariant should imply the well-definedness condition of the guard. Requiring those conditions explicitly as assumptions in the code has the effect of forcing the developer to use just such an invariant.

Answer 9.17 (p.93) The specification terminates unconditionally, but the program aborts at the first assumption when (for example) $N = 0$. The refinement by that fact alone is invalid. (Look for *simple* counter examples.)

Operationally, the code of Exercise 9.15 under the conditions of Exercise 9.16 is inappropriate because evaluation of the iteration guard could fail (and cause abortion) when $i = N$. See Exercise 13.3 for one of the well-known ways around that problem.

Chapter 10

Answer 10.1 (p.100) The main step is *leading assignment* 8.5, after writing true as $(k = 0)[k\backslash 0]$. But the refinement is operationally obvious anyway: after assigning 0 to k, the precondition $k = 0$ may be assumed.

Answer 10.2 (p.100)

1. If $P{\uparrow}l \not\leq P[l]$, then $P{\uparrow}l$ cannot be empty; hence l cannot be 0. Thus the greatest element $P[l \ominus 1]$ of $P{\uparrow}l$ (which is sorted) must exceed $P[l]$.
   The converse is immediate.
2. For any sequence q such that $\mathsf{up}\, q$, we have $x \leq q$ iff $x \leq q[0]$. Apply that to $P{\downarrow}l$, noting that $\mathsf{up}\, P\backslash\{l \ominus 1\}$ implies $\mathsf{up}\, P{\downarrow}l$.
3. From the other conjuncts, we have $P[l] < P[l \ominus 1] \leq P{\downarrow}(l+1)$.

Answer 10.4 (p.100) The guard $\neg J$ is $P{\uparrow}l \not\leq P[l]$, and that was simplified in Answer 10.2 to $l \neq 0 \wedge P[l \ominus 1] > P[l]$. We can perform that simplification here as well, because the invariant was in the precondition at the point the iteration was developed.

Answer 10.5 (p.100) It is $L[as \backslash as[l := t]]$; that is, it is as before if we imagine as with its l^{th} element replaced by t.

Since the changed program does not maintain the local invariant, it must be removed first.

Answer 10.6 (p.101) If the sequence contains repeated elements initially, no amount of sorting is going to make them go away: a strictly ascending sequence cannot result, no matter what code we use.

The step in error is the refinement of (iv) to a swap. Conjuncts J and K will have been modified to be strict —

$$
\begin{array}{lcl}
J & \widehat{=} & P{\uparrow}l < P[l] \\
K & \widehat{=} & P[l] < P{\uparrow}(l+1)
\end{array}
$$

— and the K part of the invariant is not maintained by the swap. (Check it!)

Answer 10.10 (p.101) There's no good reason why not. But one should avoid the impression that a strategy must be officially a 'law' in order to be valid, or useful.

Answer 10.11 (p.101) The meaning of the program is unaffected by the difference between $k-1$ and $k \ominus 1$, since the assignment is executed only when $k > 0$, and there the two alternatives are equal. The only disadvantage in using $k-1$ is that *if* variable k is typed as a natural number (instead of, say, as an integer), *and* automatic type checking is performed (the kind that can't 'notice' that $k > 0$ in the iteration body), *then* the statement will be rejected as ill-typed: a conservative, but at least a safe, judgement.

The disadvantage of using $k \ominus 1$ is that it would have distracted attention from the main issue, down iteration itself.

Chapter 11

Answer 11.1 (p.115)

1. $$(a := f+1)[\textbf{value } f \backslash a]$$

$$\begin{array}{ll} = & |[\ \textbf{var } l \bullet \\ & \quad l := a; \\ & \quad a := l+1 \\ &]| \\ = & a := a+1\ . \end{array}$$

2. $$(f := a+1)[\textbf{result } f \backslash a]$$

$$\begin{array}{ll} = & |[\ \textbf{var } l \bullet \\ & \quad l := a+1; \\ & \quad a := l \\ &]| \\ = & a := a+1\ . \end{array}$$

3. $$(f := a+1)[\textbf{value result } f \backslash a]$$

$$\begin{array}{ll} = & |[\ \textbf{var } l \bullet \\ & \quad l := a; \\ & \quad l := a+1; \\ & \quad a := l \\ &]| \\ = & a := a+1\ . \end{array}$$

4. $$(f := f+1)[\textbf{value result } f \backslash a]$$

$$\begin{array}{ll} = & |[\ \textbf{var } l \bullet \\ & \quad l := a; \\ & \quad l := l+1; \\ & \quad a := l \\ &]| \\ = & a := a+1\ . \end{array}$$

Answer 11.2 (p.115)

1. $$n := (n+1)!$$

$$\begin{array}{ll} = & |[\ \textbf{var } a, f \bullet \\ & \quad a := f; \\ & \quad f := a!; \\ & \quad n := f \\ &]| \\ \sqsubseteq & (f := a!)[\textbf{value } a; \textbf{result } f \backslash n+1, n]\ . \end{array}$$

2.

$$
\begin{array}{ll}
 & a\!:> a \\
= & |[\ \ \mathbf{var}\ l\bullet \\
 & \quad l := a; \\
 & \quad a\!:> l \\
 &]| \\
\sqsubseteq & a\!: [a > b]\,[\mathbf{value}\ b \backslash a]\ .
\end{array}
$$

3.

$$
\begin{array}{ll}
 & x\!: [x \neq 0\ ,\ x = 1/x_0] \\
\sqsubseteq & x\!: [(p \neq 0)[p\backslash x]\ ,\ (x = 1/p_0)[p_0\backslash x_0]] \\
\sqsubseteq & |[\ \ \mathbf{var}\ p, q\bullet \\
 & \quad p := x; \\
 & \quad q\!: [p \neq 0\ ,\ q \times p = 1]\,; \\
 & \quad x := q \\
 &]| \\
\sqsubseteq & q\!: [p \neq 0\ ,\ q \times p = 1]\,[\mathbf{value}\ p; \mathbf{result}\ q\backslash x, x]\ .
\end{array}
$$

Answer 11.3 (p.115)

$$
\begin{array}{ll}
 & x\!: [0 \leq x\ ,\ x^2 = x_0] \\
\sqsubseteq & |[\ \ \mathbf{var}\ a, b : \mathbb{R}\bullet \\
 & \quad a, b, x\!: [0 \leq x\ ,\ x^2 = x_0] \\
 &]| \\
\sqsubseteq & \text{“}\textit{following assignment}\ 3.5\text{”} \\
 & |[\ \ \mathbf{var}\ a, b : \mathbb{R}\bullet \\
 & \quad a, b, x\!: [0 \leq x\ ,\ b^2 = x_0]\,; \\
 & \quad x := b \\
 &]| \\
\sqsubseteq & \text{“}\textit{leading assignment}\ 8.5;\ \textit{contract frame}\ 5.4\text{”} \\
 & |[\ \ \mathbf{var}\ a, b : \mathbb{R}\bullet \\
 & \quad a := x; \\
 & \quad b\!: [0 \leq a\ ,\ b^2 = a_0]\,; \\
 & \quad x := b \\
 &]| \\
\sqsubseteq & \mathit{Sqrts}\ (x, x)\ .
\end{array}
$$

Answer 11.4 (p.115) Variable a has been captured. Here is the correct use of the Copy Rule:

$$
\begin{array}{ll}
 & |[\ \mathbf{procedure}\ \mathit{One}\ \ \widehat{=}\ \ a := 1 \bullet |[\ \mathbf{var}\ a \bullet \mathit{One}\ ]|\ ]| \\
\sqsubseteq & |[\ \mathbf{procedure}\ \mathit{One}\ \ \widehat{=}\ \ a := 1 \bullet |[\ \mathbf{var}\ b \bullet \mathit{One}\ ]|\ ]| \\
\sqsubseteq & |[\ \mathbf{var}\ b \bullet a := 1\ ]| \\
\sqsubseteq & a := 1\ .
\end{array}
$$

Answer 11.6 (p.116) Using a local invariant makes this easier; otherwise, just carry the bag-condition through. We have

$$
\begin{array}{rll}
& p, q, r\colon \left[\begin{array}{c} p \le q \le r \\ \llbracket p, q, r \rrbracket = \llbracket p_0, q_0, r_0 \rrbracket \end{array}\right] & \\
\sqsubseteq & \textbf{con } B;\ \textbf{ and } B = \llbracket p, q, r \rrbracket \bullet & \\
& p, q, r\colon [p \le q \le r] & \\
\sqsubseteq & \text{“\textit{sequential composition} 3.3”} & \\
& p, q, r\colon [p \le q]\,; & \text{(i)} \\
& p, q, r\colon [p \le q\ ,\ p \le r \wedge q \le r]\,; & \text{(ii)} \\
& p, q, r\colon [p \le r \wedge q \le r\ ,\ p \le q \le r] & \text{(iii)} \\
\text{(i)} \sqsubseteq & p, q := p \sqcap q, p \sqcup q & \\
\text{(ii)} \sqsubseteq & q, r := q \sqcap r, q \sqcup r & \\
\text{(iii)} \sqsubseteq & p, q := p \sqcap q, p \sqcup q\ . &
\end{array}
$$

The three commands are just an unfolded insertion sort; the invariant is maintained trivially.

Answer 11.7 (p.116) It is given in Section 6.2.3.

Answer 11.8 (p.116) The proof is straightforward.

The law as given is useful when developing a procedure for the first time: it generates the specification of the procedure body, which then may be further developed. The alternative suggested in this exercise is useful when one seeks to use an existing procedure whose specification is known already.

Chapter 12

Answer 12.2 (p.122)

$$
\begin{array}{ll}
& \neg\, \mathsf{hi}\, k \\
\equiv & (\exists\, j : l{\rightarrow}k \bullet k \rightsquigarrow j \wedge as[k] > as[j])\ .
\end{array}
$$

From J, however, we have $\mathsf{ph}(l{\rightarrow}h - \{k\})$, so the j whose existence is asserted above must be either $2k+1$ or $2k+2$. Thus we continue

$$
\equiv \quad \left\{ \begin{array}{ll} & 2k+1 < h \wedge as[k] > as[2k+1] \\ \vee & 2k+2 < h \wedge as[k] > as[2k+2]\ . \end{array} \right.
$$

Answer 12.4 (p.122) We assume that all variables range over strictly positive values. From the definition of $\lim_{N\rightarrow\infty}$, we have

$$(\forall\, c \bullet (\exists\, M \bullet (\forall\, N \bullet N \ge M \Rightarrow \mathsf{f}\, N / \mathsf{g}\, N \le c)))\ .$$

By Predicate law A.68 we conclude that

$$(\exists\, c, M \bullet (\forall\, N \bullet N \geq M \Rightarrow \mathsf{f}\, N \leq c \times \mathsf{g}\, N))\,,$$

which is just $\mathsf{f} \preceq \mathsf{g}$. It remains to show that $\mathsf{f} \not\succeq \mathsf{g}$, and we proceed

$$\begin{array}{ll}
 & \neg\,(\exists\, c, M \bullet (\forall\, N \bullet N \geq M \Rightarrow \mathsf{g}\, N \leq c \times \mathsf{f}\, N)) \\
\equiv & \text{"Predicate laws A.61, A.62"} \\
 & (\forall\, c, M \bullet (\exists\, N \bullet N \geq M \wedge \mathsf{f}\, N < 1/c \times \mathsf{g}\, N)) \\
\Leftarrow & \text{"take } d = 1/2c\text{"} \\
 & (\forall\, d, M \bullet (\exists\, N \bullet N \geq M \wedge \mathsf{f}\, N \leq d \times \mathsf{g}\, N)) \\
\Leftarrow & \text{"Lemma B.1 below"} \\
 & (\forall\, d \bullet (\exists\, M \bullet (\forall\, N \bullet N \geq M \Rightarrow \mathsf{f}\, N \leq d \times \mathsf{g}\, N))) \\
\equiv & \text{the formula above.}
\end{array}$$

In fact, Lemma B.1 is the interesting part of this exercise:

Lemma B.1 If $\mathcal{A}$ does not contain free M, then

$$(\exists\, M \bullet (\forall\, N \bullet N \geq M \Rightarrow \mathcal{A})) \Rrightarrow (\forall\, M \bullet (\exists\, N \bullet N \geq M \wedge \mathcal{A}))\,.$$

Proof:

$$\begin{array}{ll}
 & (\exists\, M \bullet (\forall\, N \bullet N \geq M \Rightarrow \mathcal{A})) \\
\equiv & \text{"}L\text{ fresh; Predicate law A.74"} \\
 & (\forall\, L \bullet (\exists\, M \bullet (\forall\, N \bullet N \geq M \Rightarrow \mathcal{A}))) \\
\Rrightarrow & \text{"Predicate law A.86"} \\
 & (\forall\, L \bullet (\exists\, M \bullet L \sqcup M \geq M \Rightarrow \mathcal{A}[N \backslash L \sqcup M])) \\
\equiv & \text{"}L \sqcup M \geq M\text{ is identically true"} \\
 & (\forall\, L \bullet (\exists\, M \bullet \mathcal{A}[N \backslash L \sqcup M])) \\
\equiv & \text{"}L \sqcup M \geq L\text{ is identically true"} \\
 & (\forall\, L \bullet (\exists\, M \bullet L \sqcup M \geq L \wedge \mathcal{A}[N \backslash L \sqcup M])) \\
\Rrightarrow & \text{"Predicate law A.87"} \\
 & (\forall\, L \bullet (\exists\, M, N \bullet N \geq L \wedge \mathcal{A})) \\
\equiv & \text{"Predicate laws A.75, A.84; } L, M \text{ not free in } \mathcal{A}\text{"} \\
 & (\forall\, M \bullet (\exists\, N \bullet N \geq M \wedge \mathcal{A}))\ \ .
\end{array}$$

□

Answer 12.8 (p.123) They do not obey the usual rules for equality: we can have $\mathsf{f} = O(\mathsf{h})$ and $\mathsf{g} = O(\mathsf{h})$, but $\mathsf{f} = \mathsf{g}$ does not follow. The intended meaning is $\mathsf{f} \in O(\mathsf{h})$, where $O(\mathsf{h}) \ \widehat{=}\ \{\mathsf{f} \mid \mathsf{f} \preceq \mathsf{h}\}$.

Answer 12.10 (p.123) Time complexity N is 'faster than' $2N$, though both are linear. Similarly, Quick Sort is faster than Heap Sort by some constant factor, on average. Incidentally, Quick Sort has worst-case complexity N^2, but in the worst case Heap Sort is still $N \log N$.

Chapter 13

Answer 13.1 (p.130) No. That es is finite is used when defining the variant to be $\#es$, since that must be an integer.

(For those with some knowledge of set theory: The variant can be any ordinal-valued function, even infinite. But in developing iteration or recursion we depend on a strict decrease of the variant:

$$\#(es - \{e\}) < \#es.$$

That is not true when $\#es$ is infinite; instead, the two expressions are equal.)

Answer 13.2 (p.130)

$$
\begin{array}{ll}
 & \textbf{procedure } \mathit{Fact}'(\textbf{value } m, k : \mathbb{N}) \ \widehat{=}\ f := m! \times k \\
\sqsubseteq & \textbf{variant } M \textbf{ is } m \bullet \\
 & \textbf{if } m = 0 \to f := k \\
 & [\!]\ \ m > 0 \to \{0 < m = M\}\ f := m! \times k \qquad \triangleleft \\
 & \textbf{fi} \\
\sqsubseteq & f\colon [0 \le m \ominus 1 < M\ ,\ f = (m \ominus 1)! \times m \times k] \\
\sqsubseteq & \mathit{Fact}'\ (m \ominus 1, m \times k)\ .
\end{array}
$$

Chapter 14

Answer 14.1 (p.135) See Figure B.1.

Answer 14.2 (p.135) The defining properties of the Gray code treat cases $n = 0$, $n = 1$, and $n \ge 2$; the code should do the same.

Answer 14.6 (p.135)

$$
\begin{array}{ll}
 & \alpha, \omega\colon [\omega = \mathsf{rv}\,\alpha_0] \\
\sqsubseteq & \textbf{rewrite}; \\
 & \alpha, \omega\colon [\omega = \omega_0 +\!\!+\ \mathsf{rv}\,\alpha_0] \qquad \triangleleft \\
\sqsubseteq & \textbf{re } R \textbf{ variant } N \textbf{ is } \#\alpha \bullet \\
 & \alpha, \omega\colon [\#\alpha = N\ ,\ \omega = \omega_0 +\!\!+\ \mathsf{rv}\,\alpha_0] \\
\sqsubseteq & \textbf{if} \quad \textbf{eof} \to \textbf{skip} \\
 & [\!]\ \ \neg\textbf{eof} \to \\
 & \qquad \alpha, \omega\colon [\alpha \ne \langle\rangle \wedge \#\alpha = N\ ,\ \omega = \omega_0 +\!\!+\ \mathsf{rv}\,\alpha_0] \qquad \triangleleft \\
 & \textbf{fi} \\
\sqsubseteq & \textbf{var } e : E \bullet \\
 & \alpha, \omega, e\colon [\#\,\mathsf{tl}\,\alpha < N\ ,\ \omega = \omega_0 +\!\!+\ \mathsf{rv}\,\mathsf{tl}\,\alpha_0 +\!\!+\ \langle \mathsf{hd}\,\alpha_0\rangle]
\end{array}
$$

n	$\mathsf{gc}\,n$	binary
0	0	0
1	1	1
2	1 1	1 0
3	1 0	1 1
4	1 1 0	1 0 0
5	1 1 1	1 0 1
6	1 0 1	1 1 0
7	1 0 0	1 1 1
8	1 1 0 0	1 0 0 0
9	1 1 0 1	1 0 0 1
10	1 1 1 1	1 0 1 0
11	1 1 1 0	1 0 1 1
12	1 0 1 0	1 1 0 0
13	1 0 1 1	1 1 0 1
14	1 0 0 1	1 1 1 0
15	1 0 0 0	1 1 1 1

Figure B.1 Gray codes

$$\begin{array}{ll}
\sqsubseteq & \textbf{input } e; \\
 & \alpha,\omega\colon [\#\alpha < N \ ,\ \omega = \omega_0 +\!\!+ \ \mathsf{rv}\,\alpha_0 +\!\!+ \langle e\rangle] \qquad \triangleleft \\
\sqsubseteq & \alpha,\omega\colon [\#\alpha < N \ ,\ \omega +\!\!+ \langle e\rangle = \omega_0 +\!\!+ \ \mathsf{rv}\ \alpha_0 +\!\!+ \langle e\rangle]\,; \qquad \triangleleft \\
 & \textbf{output } e \\
\sqsubseteq & R\ .
\end{array}$$

Chapter 15

Answer 15.2 (p.150)

1. $Buf \;\widehat{=}\; \mathsf{empty} \mid \mathsf{full}\,\mathbb{N}$
2. $Tree \;\widehat{=}\; \mathsf{tip}\,\mathbb{R} \mid \mathsf{node}\ Tree\ \ Tree$
3. violet | indigo | blue | green | yellow | orange | red

Answer 15.3 (p.150)

$$\textbf{if}\ nn\ \textbf{is}\ \mathsf{ok}\,n \rightarrow nn := \mathsf{ok}(n+1)\ \textbf{fi}$$

'increments' n is if is not undefined, and aborts otherwise.

Answer 15.4 (p.150) The specification is $d\colon [k \;\mathsf{in}\; db \Rightarrow \mathsf{full}\, k\;\; d \in db]$.

Answer 15.6 (p.150) The specification is

$$\begin{array}{lll} \textbf{if} & \neg(k\;\;\mathsf{in}\; db) & \rightarrow d := \mathsf{undefined} \\ [\!] & k \;\mathsf{in}\; db & \rightarrow d\colon [\mathsf{full}\, k\;\; d \in db] \\ \textbf{fi} & & \end{array}$$

Why is the overall specification feasible even though the body of the second branch, on its own, is not?

Answer 15.9 (p.151) The reasoning would continue

$$\begin{array}{ll} \equiv & \text{"Predicate law A.56"} \\ & paradox \in \mathsf{set}\; TooBig \wedge \mathsf{several}\, paradox \notin paradox \;\;, \end{array}$$

establishing a contradiction, since the first conjunct follows from the definition of *paradox*. Our only remaining assumption is that the definition of *TooBig* was meaningful, which we must now abandon.

Answer 15.14 (p.152) It is a strict partial order, and $\subseteq$ is a non-strict partial order. Whether it is well founded depends on the set over which it is defined: over **set** $\mathbb{N}$ it is not well founded; but over **finset** $\mathbb{N}$ it is.

Answer 15.16 (p.152)

$$\begin{array}{ll} & r, s := \mathsf{rv}\; s, ? \\ \sqsubseteq & \textbf{con}\; R \bullet \\ & r, s\colon [R = \mathsf{rv}\; s\;\;,\;\; r = R] \\ \sqsubseteq & r := \langle\rangle; \\ & r, s\colon [R = \mathsf{rv}\; s +\!\!+ r\;\;,\;\; R = \mathsf{rv}\; s +\!\!+ r \wedge s = \langle\rangle] \qquad \triangleleft \\ \sqsubseteq & \textbf{do}\; s\; \textbf{is}\; h{:}t \rightarrow r, s := h{:}r, t\; \textbf{od}\;. \end{array}$$

Answer 15.18 (p.153) This development requires two stacks of trees, say $ntl1$ and $ntl2$, with invariant 'the frontiers are equal so far':

$$\left(\exists\, F : \mathbf{seq}\, X \bullet \begin{array}{l} F1 = F +\!\!+ (+\!\!+ nt : ntl1 \bullet \mathsf{frontier}\, nt) \\ F2 = F +\!\!+ (+\!\!+ nt : ntl2 \bullet \mathsf{frontier}\, nt) \end{array} \right) \;,$$

where $F1$ and $F2$ are logical constants equal to the frontiers of $nt1$ and $nt2$ respectively.

The iteration has four alternatives, corresponding to the four possibilities for the tops of the two stacks when considered together; it should terminate when either or both of the stacks is empty.

This algorithm is a good example of one in which for space efficiency (two) explicit stacks are needed, rather than recursion. (Other options are coroutines, or a lazy functional language.)

Answer 15.19 (p.153) We give an argument for each of the two cases. First we have

$$
\begin{array}{ll}
 & \mathsf{sizelt}\ ntl \\
= & \text{“alternation guard”} \\
 & \mathsf{sizelt}(\mathsf{empty}{:}ntl' \\
= & \text{“definition } \mathsf{sizelt}\text{”} \\
 & \mathsf{sizet}\ \mathsf{empty} + \mathsf{sizelt}\ ntl' \\
= & \text{“definition } \mathsf{sizet}\text{”} \\
 & 1 + \mathsf{sizelt}\ ntl' \\
> & \mathsf{sizelt}\ ntl'\ .
\end{array}
$$

For the second we have

$$
\begin{array}{ll}
 & \mathsf{sizelt}\ ntl \\
= & \text{“alternation guard”} \\
 & \mathsf{sizelt}(\mathsf{node}\ n'\ \ nt1\ \ nt2{:}ntl' \\
= & \text{“definition } \mathsf{sizelt}\text{”} \\
 & \mathsf{sizet}(\mathsf{node}\ n'\ \ nt1\ \ nt2) + \mathsf{sizelt}\ ntl' \\
= & \text{“definition } \mathsf{sizet}\text{”} \\
 & 1 + \mathsf{sizet}\ nt1 + \mathsf{sizet}\ nt2 + \mathsf{sizelt}\ ntl' \\
= & \text{“definition } \mathsf{sizet}\text{”} \\
 & 1 + \mathsf{sizelt}(nt1{:}nt2{:}ntl') \\
> & \mathsf{sizelt}(nt1{:}nt2{:}ntl')\ .
\end{array}
$$

Chapter 16

Answer 16.2 (p.161) Change *Acquire* to read

$$t, u\colon [u \neq \mathbb{N}\ ,\ \ t \notin u_0 \wedge 2 \mid t \wedge u = u_0 \cup \{t\}]\ \ .$$

That refines the original specification, hence the new procedure refines the old, hence the new module refines the old.

But it is not feasible, because it is forced to return an even number even when u might contain only odd numbers. See Chapter 17 for a way to get around that — surely, beginning with an infinite supply of integers both even and odd, one should never run out!

module *Random*;
  var $n : \mathbb{N}$;

  procedure *Acquire* (**result** $t : \mathbb{N}$)
  $\widehat{=}$ $n := (A \times n) \ \mathsf{mod}\ B$;
    $t := n$;

  procedure *Return* (**value** $t : \mathbb{N}$)
  $\widehat{=}$ **skip**;

  initially $n = S$
end

The constants $A, B, S : \mathbb{N}$ determine the precise values returned.

Figure B.2 Random number module

Answer 16.4 (p.162) See Figure B.2; it does not refine *Tag*, because pseudo-random sequences eventually repeat.

Answer 16.5 (p.162) In Figure 16.7, yes: Part 3 of *refine module* 16.2 allows it. But in Figure 16.8, that change would affect the exported procedure $P1$ in a way not allowed by *refine module* 16.2. Hence in Figure 16.8, no.

Answer 16.6 (p.162) Originally *Out* assigned 0 to n, given the specific actual procedure *In* supplied; after the change, it assigned 1 to n instead. The new module *does* refine the old module, but the new behaviour does not refine the old behaviour.

The explanation is that the $n := 0$ behaviour, though guaranteed given the actual procedure *In*, represented a resolution of nondeterminism in the implementation which cannot be depended on by clients. Indeed the behaviour could be changed at any time by a junior programmer who knows no more than to link imported procedures to actual procedures that refine them, and that he need not ask permission to do so.

Chapter 17

Answer 17.4 (p.179) For the first, we have by *diminish specification* 17.11,

$$\begin{array}{ll} & x\colon [(\forall a_0 \cdot (\exists a \cdot x = a_0))] \\ \sqsubseteq & x\colon [(\forall a_0 \cdot x = a_0)] \ . \end{array} \qquad \text{(i)}$$

For the second, we have

$$\begin{array}{ll} & x\colon [(\forall\, a_0 \bullet (\exists\, a \bullet x = a))] \\ \sqsubseteq & x\colon [\mathsf{true}] \\ \sqsubseteq & \mathbf{choose}\ x\ . \end{array}$$

The first results in the infeasible (i): we cannot have x equal to *all* values of a_0! But a is not auxiliary in the given specification anyway, since the final value of x depends on the initial value of a. So we should expect infeasibility.

The second refines to code.

Answer 17.5 (p.179) The invariant is $N = l + m \times n$.

The augmented program is

$$\begin{array}{l} l, m, n, l', m' := 0, 1, N, 0, M; \\ \mathbf{do}\ n \neq 0 \rightarrow \\ \quad \mathbf{if}\ \mathsf{even}\, n \rightarrow m, n, m' := 2 \times m, n \div 2, 2 \times m' \\ \quad [\!]\ \ \mathsf{odd}\, n \ \ \rightarrow l, n, l' := l + m, n - 1, l' + m' \\ \quad \mathbf{fi} \\ \mathbf{od}\ . \end{array}$$

Since the original program established $l = N$, the augmented program will establish $l' = M \times N$ (as well).

To determine which variables are *auxiliary*, as far as the calculation of l' is concerned, we do the opposite: collect the essential variables. Variable n is essential, because it occurs in guards, and it is given that l' is essential. From the assignment $l' := l' + m'$ we find that we need m' as well, but the process stops there: thus l and m are auxiliary. After removing them, then renaming (of l' to l and m' to m), we get

$$\begin{array}{l} n, l, m := N, 0, M; \\ \mathbf{do}\ n \neq 0 \rightarrow \\ \quad \mathbf{if}\ \mathsf{even}\, n \rightarrow n, m := n \div 2, 2 \times m \\ \quad [\!]\ \ \mathsf{odd}\, n \ \ \rightarrow n, l := n - 1, l + m \\ \quad \mathbf{fi} \\ \mathbf{od}\ . \end{array}$$

Answer 17.8 (p.180) If the constant A were a matrix, division by $A - 1$ would no longer be a simple matter.

Answer 17.9 (p.181) We have by *diminish specification* 17.12

$$\begin{array}{ll} & x\colon [(\exists\, a \bullet pre)\ \ ,\ \ (\forall\, a \bullet pre_0 \Rightarrow post)] \\ \sqsubseteq & \text{“Predicate laws A.74, A.75”} \\ & x\colon [pre\ \ ,\ \ pre_0 \Rightarrow post] \end{array}$$

$\sqsubseteq$ "*strengthen postcondition* 5.1"
$w\colon [pre\ ,\ post]$.

Thus such specifications are unaffected.

Answer 17.10 (p.181) Here are the two laws:

Law B.3 augment assumption The assumption $\{pre\}$ *becomes* $\{pre \wedge CI\}$.

Proof:

$\{pre\}$
$= \ : [pre\ ,\ \mathsf{true}]$.

Then by *augment specification* 17.6 that *becomes*

$c\colon [pre \wedge CI\ ,\ CI]$
$\sqsubseteq \ : [pre \wedge CI\ ,\ CI]$
$\sqsubseteq$ "*strengthen postcondition* 5.1"
$: [pre \wedge CI\ ,\ \mathsf{true}]$
$= \{pre \wedge CI\}$.

□

Law B.4 diminish assumption The assumption $\{pre\}$ is diminished to $\{(\exists\, a : A \bullet pre)\}$.

Proof:

$= \ : [pre\ ,\ \mathsf{true}]$

Then by *diminish specification* 17.12 that becomes

$c\colon [(\exists\, a : A \bullet pre)\ ,\ (\forall\, a \bullet pre_0 \Rightarrow \mathsf{true})]$
$\sqsubseteq \ : [(\exists\, a : A \bullet pre)\ ,\ \mathsf{true}]$
$= \{(\exists\, a : A \bullet pre)\}$.

□

Answer 17.11 (p.181) Here are the two laws:

Law B.5 augment coercion The coercion $[post]$ *becomes* $[CI \Rightarrow post]$.
Proof:

$[post]$
$= \ : [post]$
becomes "*augment specification* 17.6"
$: [CI\ ,\ CI \wedge post]$

$$\begin{array}{l} \sqsubseteq\ :[CI \Rightarrow post] \\ =\ [CI \Rightarrow post]\ . \end{array}$$

□

Law B.6 diminish coercion The coercion $[post]$ *becomes* $[(\forall\, a : A \bullet post)]$.
Proof:

$$\begin{array}{ll} & \sqsubseteq\ [post] \\ & =\ :[post] \\ \textit{becomes} & \text{“}\textit{diminish specification}\ 17.12\text{”} \\ & \quad c\!: [(\exists\, a : A \bullet \mathsf{true})\ ,\ (\forall\, a \bullet \mathsf{true} \Rightarrow post)] \\ & \sqsubseteq\ c\!: [(\forall\, a : A \bullet post)] \\ & =\ [(\forall\, a : A \bullet post)]\ . \end{array}$$

□

Answer 17.13 (p.181) The law *augment guard* 17.9 gives the new guards $n \neq 0$ and $n = 0$ directly; the law *alternation guards* 4.3 is then unnecessary! The other data refinements are as before.

Answer 17.14 (p.181) The effect of the augmentation is to add $u \in \mathbf{finset}\,\mathbb{N}$ as an invariant of every command: in particular,

$$\begin{array}{ll} & t, u\!: [u \neq \mathbb{N}\ ,\ t \notin u_0 \wedge u = u_0 \cup \{t\}] \\ \textit{becomes} & t, u\!: [u \neq \mathbb{N}\ ,\ u \in \mathbf{finset}\,\mathbb{N}\ ,\ t \notin u_0 \wedge u = u_0 \cup \{t\}]\ . \\ \sqsubseteq & \text{“}u \cup \{t\} \text{ is finite if } u \text{ is”} \\ & t, u\!: [u \in \mathbf{finset}\,\mathbb{N}\ ,\ t \notin u_0 \wedge u = u_0 \cup \{t\}]\ . \end{array}$$

The effect on Exercise 16.2 is to allow us to include $u \in \mathbf{finset}\,\mathbb{N}$ in the precondition before we strengthen the postcondition with $2 \mid t$. Feasibility is maintained.

Answer 17.15 (p.181) After adding v we have Figure B.3; removing u gives Figure B.4. See also Exercise 17.16.

Answer 17.18 (p.182) The data-type invariant (such as imposed by **and**) of the module is strengthened; the augmentation laws simply ensure that it is maintained.

Answer 17.20 (p.182) The concrete iteration **do** $aq[n] \neq a \rightarrow n := n + 1$ **od** would fail to terminate if a were not in aq. It *is* reasonable, since the abstract command could abort in that case.

Answer 17.22 (p.182) There is no hard-and-fast answer to questions like that: it depends on what your ‘compiler’ (or reader) will accept. In this case, one might

module *Tag*
  var u, v : **set** $\mathbb{N}$;

  procedure *Acquire* (**result** $t : \mathbb{N}$)
  $\widehat{=}\ u, t, v\colon [u \neq \mathbb{N}\ ,\ u \subseteq v \wedge v \in \mathbf{finset}\,\mathbb{N}\ ,\ t \notin u_0 \wedge u = u_0 \cup \{t\}]\,;$

  procedure *Return* (**value** $t : \mathbb{N}$)
  $\widehat{=}\ u, v := u - \{t\}, v;$

  initially $u = \{\} \wedge u \subseteq v \wedge v \in \mathbf{finset}\,\mathbb{N}$
end

Figure B.3 Augmented module (Exercise 17.15)

module *Tag*
  var v : **set** $\mathbb{N}$;

  procedure *Acquire* (**result** $t : \mathbb{N}$)
  $\widehat{=}\ t, v\colon [v \neq \mathbb{N}\ ,\ t \notin v_0 \wedge v = v_0 \cup \{t\}]\,;$

  procedure *Return* (**value** $t : \mathbb{N}$)
  $\widehat{=}\ v := v;$

  initially $v = \{\}$
end

Figure B.4 Diminished module (Exercise 17.15)

be thinking of a further data refinement to a fixed-length sequence and an integer 'end-of-sequence' pointer. The data refinement of `fr` then is trivial.

Answer 17.24 (p.182) The refinements are

1. $a := aq[a]$;
2. $a, n := as[n-1], n-1$; and
3. $\{n < N\}\ aq[n], n := a, n+1$.

Answer 17.25 (p.182) Binary search is a suitable finishing off of the derivation. (See Exercise 10.7).

Answer 17.26 (p.182)

$$
\begin{array}{ll}
 & w\colon [pre\ ,\ post] \\
\sqsubseteq & \text{“}\textit{expand frame}\ 8.3\text{”} \\
 & w, a\colon [pre\ ,\ post \wedge a = a_0] \\
\multicolumn{2}{l}{\textit{becomes}\ \text{“}\textit{data-refine specification}\ 17.15\text{”}} \\
 & w, c\colon [pre[a \backslash \mathsf{af}\ c]\ ,\ \mathsf{dti}\ c\ ,\ post[a_0, a \backslash \mathsf{af}\ c_0, \mathsf{af}\ c] \wedge \mathsf{af}\ c = \mathsf{af}\ c_0] \\
\sqsubseteq & \text{“}\textit{contract frame}\ 5.4\text{”} \\
 & w\colon [pre[a \backslash \mathsf{af}\ c]\ ,\ \mathsf{dti}\ c\ ,\ post[a_0, a \backslash \mathsf{af}\ c_0, \mathsf{af}\ c]]\ .
\end{array}
$$

The effect is exactly as before, so that *data-refine specification* 17.15 can be used even when a does not occur in the abstract frame. Note however that the c_0 in the postcondition can be replaced by c, since c is not in the concrete frame.

Of more interest however is the use of *contract frame* 5.4, where a proper refinement occurs. The conjunct $\mathsf{af}\ c = \mathsf{af}\ c_0$, effectively strengthened to $c = c_0$ by *contract frame* 5.4, allows considerable freedom in adjusting the concrete representation. (See Exercise 17.27).

Answer 17.27 (p.182) Implementing $a \in as$, where the concrete representation is an *unordered* sequence aq, might result in bringing a to the front of the sequence aq so that it would be found more quickly on a subsequent search.

Answer 17.28 (p.183) Assuming a declaration $adb : K \nrightarrow D$, the specification would be

$$
\begin{array}{l}
\textbf{if}\ k \notin \mathsf{dom}\ adb \rightarrow d := \mathsf{undefined} \\
[]\ \ k \in \mathsf{dom}\ adb \rightarrow d := \mathsf{ok}\ adb\ k \\
\textbf{fi}\ .
\end{array}
$$

The coupling invariant is $adb = \{k : K;\ d : d \mid \mathsf{full}\ k\ \ d \in db\}$. Note that it is functional.

Answer 17.29 (p.183) If the array were fixed-size, then there would be an upper limit on the number of trees it could contain; with the given algorithm the size needed would depend on the depth of the original tree. Thus the specification would indeed need modification, including a precondition limiting the depth of the tree nt.

That taken care of, the data-refinement is simple. The abstract variable is ntl, and the concrete say would be $nta : \mathbf{seq}_N\ Tree;\ h : \mathbb{N}$ with coupling invariant $ntl = nta{\uparrow}h \wedge h \leq N$. The resulting code is shown in Figure B.5.

That leaves the records and pointers. Since the original tree nt is not modified, one could represent it and its subtrees by pointers to their root nodes: the array nta would then become a fixed-length sequence of pointers to records.

Answer 17.35 (p.183) These are the refinement steps:

$|[$ **var** $nta : \mathbf{seq}_N\ Tree;\ h : \mathbb{N}\bullet$
$\quad nta[0], h := \langle nt \rangle, 1;$
$\quad$ **do** $h \neq 0 \rightarrow$
$\qquad$ **if** $nta[h-1]$ is
$\qquad\quad$ empty $\rightarrow h := h - 1$
$\qquad [\!]$ (node $n'\, nt1\ \ nt2$) $\rightarrow$
$\qquad\quad n, nta[h-1], nta[h], h := n + n', nt1, nt2, h + 1$
$\qquad$ **fi**

$\quad$ **od**
$]|$

Figure B.5 Iterative tree-summing with fixed-size array

$\quad x := 1$
$\sqsubseteq$ "*simple specification* 8.1"
$\quad x \colon [x = 1]$
$\sqsubseteq\ x \colon [(x = 1 \vee x = -1) \wedge x \geq 0]$
$\sqsubseteq$ "*absorb coercion* 17.2 (backwards)"
$\quad x \colon [x = 1 \vee x = -1]\,;$ $\quad\triangleleft$
$\quad [x \geq 0]$ $\quad$ (i)
$\sqsubseteq$ "*alternation* 4.1, *assignment* 1.3"
$\quad$ **if** true $\rightarrow x := 1$
$\quad [\!]$ true $\rightarrow x := -1$
$\quad$ **fi** .

The coercion (i) would test for $x \geq 0$, and force *backtracking* to the earlier non-deterministic choice if the test failed. 'Eventually' the correct branch $x := 1$ would be taken, and the coercion would then behave like **skip**.

Chapter 18

Answer 18.1 (p.196) If $\neg$ em $as{\uparrow}i$, then for all x we have

$$(as{\uparrow}i).x \leq i/2\ .$$

Hence for all x not equal to $as[i]$, we have

$$(as{\uparrow}(i+1)).x \leq (i+1)/2\ .$$

Thus the only possible majority in $as{\uparrow}(i+1)$ is $as[i]$ itself.

Answer 18.4 (p.196) We have these equalities:

$$
\begin{array}{ll}
& |[\ \mathbf{var}\ x \bullet prog\ ]| \\
= & |[\ \mathbf{var}\ x;\ \mathbf{procedure}\ P\ \widehat{=}\ prog \bullet \\
& \quad P \\
&]| \\
= & |[\ \ \mathbf{module}\ M \\
& \qquad \mathbf{var}\ x; \\
& \qquad \mathbf{procedure}\ P\ \widehat{=}\ prog \\
& \quad \mathbf{end} \bullet \\
& \\
& \quad P \\
&]|\ .
\end{array}
$$

The transformation is carried out on the interior of module M, and the above process reversed.

Answer 18.5 (p.196) For p.187, we need two steps. First, the invariant $c = as{\uparrow}i.x$ is conjoined to the guards. Then we must show that

$$
\begin{array}{lll}
1. & & (\neg\, \mathsf{em}\ as{\uparrow}i \wedge c = as{\uparrow}i.x) \\
& \vee & (\mathsf{sm}\ x\ \ as{\uparrow}i \wedge c = as{\uparrow}i.x) \\
& & \\
& \Rrightarrow & \mathsf{true} \\
2. & & ((\neg\, \mathsf{em}\ as{\uparrow}i \wedge c = as{\uparrow}i.x) \\
& \vee & (\mathsf{sm}\ x\ \ as{\uparrow}i \wedge c = as{\uparrow}i.x)) \\
& \wedge & c \leq i/2 \\
& & \\
& \Rrightarrow & \neg\, \mathsf{em}\ as{\uparrow}i \wedge c = as{\uparrow}i.x \\
3. & & ((\neg\, \mathsf{em}\ as{\uparrow}i \wedge c = as{\uparrow}i.x) \\
& \vee & (\mathsf{sm}\ x\ \ as{\uparrow}i \wedge c = as{\uparrow}i.x)) \\
& \wedge & c > i/2 \\
& & \\
& \Rrightarrow & \mathsf{sm}\ x\ \ as{\uparrow}i \wedge c = as{\uparrow}i.x\ .
\end{array}
$$

For p.189, the procedure is similar.

Chapter 20

Answer 20.1 (p.207) The refinement is valid because, in seeking to minimise overall waste wt, we cannot do better than to minimise wt for what remains.

That is not a property of all waste functions, but it is of wt. Is it a property of this one?

$$\mathsf{wt1}\ pss\ \ \widehat{=}\ \ (+ls : \mathsf{fr}\ pss \bullet M - \textstyle\sum ls)$$

What kind of paragraph results from minimising wt1?

The refinement of (ii) is proved by *strengthen postcondition*, then (trivially) *weaken precondition*. What we must show is of the form

$$(\forall\, qss \cdot \mathcal{A} \Rightarrow \mathcal{B}) \quad \Rrightarrow \quad (\forall\, qss \cdot \mathcal{A} \Rightarrow \mathcal{C})\,,$$

and that follows from just $\mathcal{B} \Rrightarrow \mathcal{C}$ by propositional reasoning and distribution of $\Rrightarrow$ through $\forall\, qss$. The precondition of (ii) contains I, which by instantiating qss to $\langle ws[i{\rightarrow}j]\rangle +\!\!+\; qss$ (Predicate law A.86) gives

$$\begin{array}{ll} & \mathsf{ep}(ws{\downarrow}i)\ (\langle ws[i{\rightarrow}j]\rangle +\!\!+\; qss) \\ \Rightarrow & \mathsf{ep}\, ws\ (pss +\!\!+\; \langle ws[i{\rightarrow}j]\rangle +\!\!+\; qss)\ . \end{array}$$

That is exactly what is required.

Answer 20.3 (p.207) It is removed when the *shortest* next line is taken.

Answer 20.4 (p.207) Procedure *GetWord* is used to fill the sequence ws, in each case assigning $w+1$ to account for a following blank. The line width M is increased by 1 to allow for the blank following the last word on a line.

In the code of the second half, pss is removed. The command $j, s := j+1, s+ws[j]$ is followed by *PutWord* and $pss := pss +\!\!+\; \langle ws[i{\rightarrow}j]\rangle$ is replaced by *PutLine*.

Chapter 21

Answer 21.1 (p.216) The time is linear in the size of hs.

For a given call of *Hist*, let I be the value it receives through its formal parameter i, and let J be the value it returns through its formal parameter J. Note first that $I < J$: the procedure is guaranteed to return J greater than the I it received.

Then consider the number of further calls made by *Hist* (to itself): it is no more than $J - I - 1$, and so any call to *Hist* accounts for at most $J - I$ calls in all (including the initial one).

Since for the initial call we have $I = -1$, and the J returned cannot exceed N, we have limited the total number of calls to $N + 1$.

Answer 21.2 (p.216) Consider successively longer 'row prefixes' of the rectangle, maintaining the invariant that the largest true rectangle of the prefix is known. Re-establishing the invariant for one further row is done by finding the largest rectangle under the histogram formed by 'true columns' above the new row.

The heights of columns in the histogram above successive rows can form part of the invariant also, leading to overall time complexity linear in the area of the array.

Answer 21.3 (p.216) This problem is much easier than Exercise 21.2, though the complexity is still linear in the area of the array. One considers 'lower-right corners' of possible squares element-by-element along rows. Knowing the height of a 'true column' above the current element, the length of the 'true suffix' of the part of the current row ending in that element, and finally the size of the largest true square for the element immediately left and above it, is sufficient to re-establish the invariant that the largest true square is known for all elements considered so far.

The difference between squares and rectangles is that if one square has less area than another, then it can be nested within the other. The same is not true of rectangles.

Chapter 22

Answer 22.1 (p.244) Following the convention that all procedures have the identity of their user as the first parameter, the definition would be

$$\begin{array}{l}\textbf{procedure } \mathit{Cancel}\ (\textbf{value } \mathit{me} : \mathit{Usr};\ \mathit{id} : \mathit{Id}) \\ \mathrel{\widehat{=}}\ \mathit{sent} := \mathit{sent} \rhd \{\mathit{id}\}\ .\end{array}$$

'Unauthorised' removal is possible by any user knowing the *id* of the message, and in practice that would be someone who has already received it. (The sender knows the *id*, but is 'authorised'.)

Including *me* always as a first parameter allows subsequent design changes without altering the interface — in this case, we might change the specification to record messages' senders, and alter *Cancel* to take that into account. Would it be a refinement?

Answer 22.3 (p.244) The unfortunate programmer saw in the specification of *Send* that the new identifier was first chosen outside the set $\mathsf{dom}\, msgs$, and then immediately added to it by $msgs[id] := msg$. Noticing that *msgs* was not assigned to elsewhere in the module, he reasoned correctly that Procedure *Send* never returned the same *id* twice. Rather than write a generator of unique identifiers himself, therefore, he simply used the one he had found.

The change represented by Figure 22.2 invalidated his second observation, for *msgs* is assigned to by the replacement *Read*, and in fact made smaller.

Answer 22.8 (p.246) Remove *msgs*, *sent* and *recd* as abstract variables, and introduce the concrete variable *used*; the coupling invariant is

$$\mathsf{ran}\, sent \subseteq used'\ .$$

Yes, it is a problem. If we do not address *time* explicitly, then we do not have the vocabulary with which to express 'prompt' delivery. And one extreme of not being prompt is not to deliver at all.

Beyond *any* mathematical specification are 'extra-mathematical' requirements that must be specified some other way. If informal requirements are not good enough in any particular case, then more sophisticated mathematical techniques must be used; where the mathematics stops, and informality begins, is a matter for taste and good judgement.