Chapter 18

Case study: Majority voting

Although our goal in program development is to reach code, there are reasons one might want to go further: to increase efficiency is one; and a second reason is to translate from one programming language into another. In this chapter, an example more extended than usual, we show two successive program developments, both successful. But the first, too inefficient (quadratic complexity), provides the motivation for extra trouble and ingenuity pursued in the second (linear complexity).

From the second attempt, however, we go much further; a series of carefully chosen transformations in the spirit of Chapter 17 leads on to a program of unexpected simplicity.

18.1 Refinement of code

One of our early examples of alternation was the following program fragment, illustrating both nondeterminism and nontermination:

$$\begin{array}{l} \textbf{if } 2 \mid x \rightarrow x := x \div 2 \\ [\!]\ \ 3 \mid x \rightarrow x := x \div 3 \\ \textbf{fi}\ . \end{array}$$

In spite of its being code, we may nevertheless need to refine it further, given the demands of a particular programming language. As we saw in Exercise 4.5, one possibility is

$$\begin{array}{ll} \sqsubseteq & \text{“}\textit{alternation guards}\ 4.3\text{”} \\ & \textbf{if } 2 \mid x \quad\quad\ \rightarrow x := x \div 2 \\ & [\!]\ \ \neg(2 \mid x) \rightarrow x := x \div 3 \\ & \textbf{fi} \\ = & \text{“transliteration into } C\text{”} \end{array}$$

```
if (x%2 == 0) x=x/2; else x=x/3;
```

Note that the above is a proper refinement: the final program does not equal the original. If $x = 6$, the original program sets it either to 2 or to 3; but the refined program sets it to 2. If $x = 7$, the original program can abort; the refined program must terminate, setting it to 2.

The final phase of this chapter will be concerned with replacing code by code, sometimes by simple refinement as above, sometimes by transformation (as within modules). To begin, however, we set out the problem and follow a straightforward and innocent approach to its development.

18.2 Winning an election

The *strict majority* of a bag of values is that value occurring with frequency strictly more than half the size of the bag. If the values represented votes, the strict majority value would identify the candidate, if any, that had an absolute majority.

Not every bag has a strict majority (just as not every election has an absolute winner). For example, the strict majority in $[\![A, B, A, C, A]\!]$ is A, but neither the empty bag nor $[\![A, B, B, A]\!]$ has a strict majority.

To be more specific, we define three predicates: first sm for *strict majority*; then em for *exists majority*; and finally cm for *conditional majority*. It will be convenient in the code to use a sequence rather than a bag, and so we suppose a sequence $as : \mathbf{seq}\ T$ and value $x : T$, defining

$$
\begin{array}{rcl}
\mathsf{sm}\ x\ as & \widehat{=} & as.x > \#as/2 \\
\mathsf{em}\ as & \widehat{=} & (\exists\, x : T \cdot \mathsf{sm}\ x\ as) \\
\mathsf{cm}\ x\ as & \widehat{=} & \mathsf{em}\ as \Rightarrow \mathsf{sm}\ x\ as\ .
\end{array}
$$

(Recall from p.81 that $as.x$ is the number of occurrences of x in as, whether as is a set or a bag.)

The task of our program will be to find a strict majority if there is one, terminating whether there is one or not. (Thus if as contains no strict majority, the program may set x at random, but still must terminate.) Here is our abstract program:

$$
\begin{array}{l}
\mathbf{var}\ as : \mathbf{seq}_N\ T;\ \ x :\ \ T \cdot \\
x\!: [\mathsf{cm}\ x\ \ as]\ \ .
\end{array}
$$

18.3 A straightforward attempt yields quadratic code

18.3.1 A simple invariant

We begin with our usual strategy, to establish the postcondition over longer and longer prefixes:

$$
\begin{array}{l}
|[\ \ \mathbf{var}\ i : \mathbb{N}\bullet \\
\\
\quad i := 0; \\
\quad \mathbf{do}\ i \neq N \rightarrow \\
\qquad \mathbf{if}\ \neg\, \mathsf{em}\ as{\uparrow}i \rightarrow x := as[i] \\
\qquad [\!]\ \ \mathsf{sm}\ x\ \ as{\uparrow}i \rightarrow \mathbf{skip} \\
\qquad \mathbf{fi}; \\
\qquad i := i + 1 \\
\quad \mathbf{od} \\
]|
\end{array}
$$

Figure 18.1 Summary of first refinements

$$
\begin{array}{lll}
\sqsubseteq & \text{“iterate up”}\ \mathbf{var}\ i : \mathbb{N}\bullet & \\
& i := 0; & \\
& \mathbf{do}\ i \neq N \rightarrow & \\
& \quad x\colon [\mathsf{cm}\ x\ \ as{\uparrow}i\ ,\ \ \mathsf{cm}\ x\ \ as{\uparrow}(i+1)]\,; & \triangleleft \\
& \quad i := i + 1 & \\
& \mathbf{od} & \\
\sqsubseteq & \mathbf{if}\ \neg\, \mathsf{em}\ as{\uparrow}i \rightarrow x\colon [\neg\, \mathsf{em}\ as{\uparrow}i\ ,\ \ \mathsf{cm}\ x\ \ as{\uparrow}(i+1)] & \text{(i)} \\
& [\!]\ \ \mathsf{sm}\ x\ \ as{\uparrow}i \rightarrow x\colon [\mathsf{sm}\ x\ \ as{\uparrow}i\ ,\ \ \mathsf{cm}\ x\ \ as{\uparrow}(i+1)] & \text{(ii)} \\
& \mathbf{fi}\ . &
\end{array}
$$

Note how the alternation exploits the disjunction inherent in $\mathsf{cm}\ x\ \ as{\uparrow}i$.

Now if there is no majority in $as{\uparrow}i$, then the only possible majority in $as{\uparrow}(i+1)$ is $as[i]$ itself. Hence

$$\text{(i)}\ \ \sqsubseteq\ \ x := as[i]\ .$$

On the other hand, if x is the majority in $as{\uparrow}i$, then either it is the majority in $as{\uparrow}(i+1)$ as well (certainly if $as[i] = x$) or there is no majority in $as{\uparrow}(i+1)$ at all (but only if $as[i] \neq x$). Hence

$$\text{(ii)}\ \ \sqsubseteq\ \ \mathbf{skip}\ .$$

The program so far is collected in Figure 18.1. Only the guards are left to do.

18.3.2 State transformation of local blocks

In Chapter 17, we showed how to transform the state of modules. The same techniques apply to local blocks.

$$
\begin{array}{l}
|[\ \ \mathbf{var}\ i, c : \mathbb{N};\\
\quad \mathbf{initially}\ c = as{\uparrow}i.x\ \bullet\\
\\
\quad i, c := 0, 0;\\
\quad \mathbf{do}\ i \neq N \rightarrow\\
\qquad \mathbf{if}\ c \leq i/2 \rightarrow x, c := as[i], (as{\uparrow}i).as[i] \qquad\qquad \text{(iii)}\\
\qquad [\!]\ \ c > i/2 \rightarrow \mathbf{skip}\\
\qquad \mathbf{fi};\\
\qquad \mathbf{if}\ as[i] = x \rightarrow i, c := i+1, c+1\\
\qquad [\!]\ \ as[i] \neq x \rightarrow i := i+1\\
\qquad \mathbf{fi}\\
\quad \mathbf{od}\\
]|
\end{array}
$$

Figure 18.2 Introduction of count c

Recall Figures 16.2 and 16.3. A local block can be made into a module, then transformed, then made back into a local block again. An initialisation may suddenly appear in the transformed block, of course. To simplify that, we introduce this abbreviation:

Abbreviation 18.1 local block initialisation

$$
\begin{array}{ll}
 & |[\ \mathbf{var}\ l : T;\ \ \mathbf{initially}\ inv \bullet prog\ ]| \\
\widehat{=} & |[\ \mathbf{var}\ l : T \bullet l\!: [inv];\ prog\ ]|\ .
\end{array}
$$

□

(Abbreviation 18.1 also simplifies the translation between modules and local blocks.)

All the transformation laws of Chapter 17 carry over to local blocks. In fact, even more is possible for local blocks: the coupling invariant can refer to global variables as well (compare p.164 'any formula over the *local and exported* variables of the module'). But we do not need that here.

Now we apply the above to Figure 18.1. We add a variable $c : \mathbb{N}$ which counts the occurrences of x in the prefix $as{\uparrow}i$ examined so far: the coupling invariant is $c = as{\uparrow}i.x$. That gives the program of Figure 18.2. Though we include the initialisation, Abbreviation 18.1 shows it to be unnecessary: it is subsumed by the command $i, c := 0, 0$.

The law *alternation guards* 4.3 has allowed the two guards to be simplified dramatically, to $c \leq i/2$ and $c > i/2$. (See Exercise 18.5.) That leaves only the refinement of (iii):

$$
\begin{array}{lll}
\text{(iii)} & \sqsubseteq & x := as[i];\\
 & & c := as{\uparrow}i.x \qquad\qquad \triangleleft
\end{array}
$$

$$
\begin{array}{l}
\sqsubseteq \text{ “invariant } c = as{\uparrow}j.x \wedge j \le i\text{”; } \mathbf{var}\ j : \mathbb{N}\bullet \\
\quad j, c := 0, 0; \\
\quad \mathbf{do}\ j \ne i \to \\
\quad\quad \mathbf{if}\ x = as[j] \to c, j := c+1, j+1 \\
\quad\quad [\!]\ \ x \ne as[j] \to j := j+1 \\
\quad\quad \mathbf{fi} \\
\quad \mathbf{od}.
\end{array}
$$

We have reached code. But its time complexity is quadratic: due to the iteration appearing as a the refinement of (iii), the entire prefix $as{\uparrow}i$ is re-examined to compute c for the newly chosen x.

18.4 A second attempt is faster

18.4.1 How to do better

The troublesome (iii) occurs in a command guarded by $\neg\,\mathsf{em}\ as{\uparrow}i$, which we did not exploit. Can that reduce the time complexity? A crucial property of sm is that for any sequences as, as' and value x,

$$\mathsf{sm}\ x\ \ (as +\!\!+\ as') \quad \Rrightarrow \quad \mathsf{sm}\ x\ \ as \vee \mathsf{sm}\ x\ \ as'\ . \tag{18.1}$$

If x is a majority in a concatenation, then it must be a majority in one part or the other. Hence if x is a majority in as overall (that is, if $\mathsf{sm}\ x\ \ as$), but $as{\uparrow}i$ has no majority (and $\neg\,\mathsf{em}\ as{\uparrow}i$), then x must be a majority in the remainder $as{\downarrow}i$ (thus $\mathsf{sm}\ x\ \ as{\downarrow}i$). Thus under the given conditions, we can forget the prefix altogether!

A convenient consequence of the above is given in this lemma:

Lemma 18.1 For sequences as, as',

$$\neg\,\mathsf{em}\ as \wedge \mathsf{cm}\ x\ \ as' \quad \Rrightarrow \quad \mathsf{cm}\ x\ \ (as +\!\!+\ as')\ .$$

Proof: Note that $\neg\,\mathsf{em}\ as \Rrightarrow \neg\,\mathsf{sm}\ x\ \ as$. Now we consider the two cases in $\mathsf{cm}\ x\ \ as'$.

First, if $\neg\,\mathsf{em}\ as'$, then by Property (18.1) we have $\neg\,\mathsf{em}(as +\!\!+\ as')$, hence it follows that $\mathsf{cm}\ x\ \ (as +\!\!+\ as')$.

Second, if $\mathsf{sm}\ x\ \ as'$ then for all $y \ne x$ we have $\neg\,\mathsf{sm}\ y\ \ as'$. Again by Property (18.1), for all $y \ne x$ that gives $\neg\,\mathsf{sm}\ y\ \ (as +\!\!+\ as')$, hence finally $\mathsf{cm}\ x\ \ (as +\!\!+\ as')$.
□

Lemma 18.1 leads us to this new development:

$$
\begin{array}{l}
\quad x\colon [\mathsf{cm}\ x\ \ as] \\
\sqsubseteq \text{ “Lemma 18.1”; } \mathbf{var}\ j : \mathbb{N}\bullet \\
\quad x, j\colon [\neg\,\mathsf{em}\ as[0{\to}j] \wedge \mathsf{cm}\ x\ \ as[j{\to}N]\ ]
\end{array}
$$

$$
\begin{array}{l}
|[\ \ \mathbf{var}\ i,j,c:\mathbb{N}\bullet \\
\\
\qquad i,j,c:=0,0,0; \\
\qquad \mathbf{do}\ i\neq N\rightarrow \\
\qquad\qquad \mathbf{if}\ c\leq(i-j)/2\rightarrow j,x,c:=i,as[i],0 \\
\qquad\qquad [\!]\ \ c>(i-j)/2\rightarrow \mathbf{skip} \\
\qquad\qquad \mathbf{fi}; \\
\qquad\qquad \mathbf{if}\ as[i]=x\rightarrow i,c:=i+1,c+1 \\
\qquad\qquad [\!]\ \ as[i]\neq x\rightarrow i:=i+1 \\
\qquad\qquad \mathbf{fi} \\
\qquad \mathbf{od} \\
]|
\end{array}
$$

Figure 18.3 Second attempt: linear code

$$
\begin{array}{ll}
\sqsubseteq & \left\{\begin{array}{l} I\ \widehat{=}\ \neg\,\mathsf{em}\ as[0\rightarrow j]\wedge\mathsf{cm}\ x\ \ as[j\rightarrow i] \\ \mathbf{var}\ i:\mathbb{N}\bullet \end{array}\right. \\
& x,j,i\colon[I\wedge i=N] \\
\\
\sqsubseteq & \text{“invariant } I\wedge j\leq i\leq N\text{”} \\
& i,j:=0,0; \\
& \mathbf{do}\ i\neq N\rightarrow \\
& \qquad \mathbf{if}\ \neg\,\mathsf{em}\ as[0\rightarrow j]\wedge\neg\,\mathsf{em}\ as[j\rightarrow i]\rightarrow \\
& \qquad\qquad x,j\colon[\neg\,\mathsf{em}\ as[0\rightarrow j]\wedge\neg\,\mathsf{em}\ as[j\rightarrow i]\ ,\ I[i\backslash i+1]] \qquad\qquad \text{(iv)} \\
& \qquad [\!]\ \ \neg\,\mathsf{em}\ as[0\rightarrow j]\wedge\mathsf{sm}\ x\ \ as[j\rightarrow i]\rightarrow \\
& \qquad\qquad x,j\colon[\neg\,\mathsf{em}\ as[0\rightarrow j]\wedge\mathsf{sm}\ x\ \ as[j\rightarrow i]\ ,\ I[i\backslash i+1]] \qquad\qquad \text{(v)} \\
& \qquad \mathbf{fi}; \\
& \qquad i:=i+1 \\
& \mathbf{od}
\end{array}
$$

$$
\begin{array}{ll}
\text{(iv)}\ \sqsubseteq & j,x:=i,as[i] \\
\\
\text{(v)}\ \sqsubseteq & \mathbf{skip}\ .
\end{array}
$$

Again we introduce c, this time with the coupling invariant $c=as[j\rightarrow i].x$; the result is Figure 18.3. Note that again *alternation guards* 4.3 is used to simplify the guards. (See Exercise 18.5.) Now the code has linear time complexity; but — as we see below — it can be simplified dramatically.

$$
\begin{array}{l}
|[\ \ \mathbf{var}\ i, j, c : \mathbb{N};\ d : \mathbb{Z} \cdot \\
\\
\quad i, j, c, d := 0, 0, 0, 0; \\
\quad \mathbf{do}\ i \neq N \rightarrow \\
\qquad \mathbf{if}\ d \leq 0 \rightarrow j, x, c, d := i, as[i], 0, 0 \\
\qquad [\!]\ \ d > 0 \rightarrow \mathbf{skip} \\
\qquad \mathbf{fi}; \\
\qquad \mathbf{if}\ as[i] = x \rightarrow i, c, d := i+1, c+1, d+1 \\
\qquad [\!]\ \ as[i] \neq x \rightarrow i, d := i+1, d-1 \\
\qquad \mathbf{fi} \\
\quad \mathbf{od} \\
]|
\end{array}
$$

Figure 18.4 Add variable d

18.5 Transformation of code

18.5.1 Representing two variables by one

The guards in Figure 18.3 can be further simplified by a state transformation. We introduce a single variable $d : \mathbb{Z}$ using the coupling invariant $d = 2c - (i - j)$; then we remove c, j. First, the guards become $d \leq 0$ and $d > 0$, and the resulting program is Figure 18.4. (The superfluous initialisation is omitted.) Then, removing the auxiliary c, j gives Figure 18.5.

18.5.2 Laws of distribution

Inspection of Figure 18.5 reveals that the two alternations are not independent: the $d \leq 0$ branch of the first cannot be followed by the $as[i] \neq x$ branch of the second. With the following distribution law we can exploit that:

Law 18.2 <u>left-distribution of composition over alternation</u>

$$
\begin{array}{ll}
 & \mathbf{if}\ ([\!]\ i \cdot G_i \rightarrow \mathit{branch}_i)\ \mathbf{fi};\ \ \mathit{prog} \\
= & \mathbf{if}\ ([\!]\ i \cdot G_i \rightarrow \mathit{branch}_i;\ \ \mathit{prog})\ \mathbf{fi}\ .
\end{array}
$$

□

First, we distribute $i := i + 1$ out of the second alternation; then we distribute the second alternation into the first. The result is Figure 18.6.

In the first branch, we now have assignments before an alternation. We can use the following law to simplify that:

$$
\begin{array}{l}
|[\ \ \mathbf{var}\ i:\mathbb{N};\ \ d:\mathbb{Z}\bullet \\
\\
\quad i,d:=0,0; \\
\quad \mathbf{do}\ i\neq N\rightarrow \\
\qquad \mathbf{if}\ d\leq 0\rightarrow x,d:=as[i],0 \\
\qquad [\!]\ \ d>0\rightarrow \mathbf{skip} \\
\qquad \mathbf{fi}; \\
\qquad \mathbf{if}\ as[i]=x\rightarrow i,d:=i+1,d+1 \\
\qquad [\!]\ \ as[i]\neq x\rightarrow i,d:=i+1,d-1 \\
\qquad \mathbf{fi} \\
\quad \mathbf{od} \\
]|
\end{array}
$$

Figure 18.5 Remove auxiliary c, j

$$
\begin{array}{l}
\vdots \\
\mathbf{if}\ d\leq 0\rightarrow \\
\qquad x,d:=as[i],0; \\
\qquad \mathbf{if}\ as[i]=x\rightarrow d:=d+1 \\
\qquad [\!]\ \ as[i]\neq x\rightarrow d:=d-1 \\
\qquad \mathbf{fi} \\
[\!]\ \ d>0\rightarrow \\
\qquad \mathbf{if}\ as[i]=x\rightarrow d:=d+1 \\
\qquad [\!]\ \ as[i]\neq x\rightarrow d:=d-1 \\
\qquad \mathbf{fi} \\
\mathbf{fi}; \\
i:=i+1 \\
\vdots
\end{array}
$$

Figure 18.6 Merge alternations

Law 18.3 <u>right-distribution of assignment over alternation</u>

$$
\begin{array}{ll}
 & x:=E;\ \ \mathbf{if}\ ([\!]\ i\bullet G_i\rightarrow branch_i)\ \mathbf{fi} \\
= & \mathbf{if}\ ([\!]\ i\bullet G_i[x\backslash E]\rightarrow x:=E;\ \ branch_i)\ \mathbf{fi}\ .
\end{array}
$$

□

The result is Figure 18.7.

$$
\begin{array}{l}
\vdots \\
\textbf{if } d \le 0 \rightarrow \\
\qquad \textbf{if } \mathsf{true} \rightarrow x, d := as[i], 0;\;\; d := d+1 \\
\qquad [\!]\;\; \mathsf{false} \rightarrow x, d := as[i], 0;\;\; d := d-1 \\
\qquad \textbf{fi} \\
[\!]\;\; d > 0 \rightarrow \\
\qquad \textbf{if } as[i] = x \rightarrow d := d+1 \\
\qquad [\!]\;\; as[i] \neq x \rightarrow d := d-1 \\
\qquad \textbf{fi} \\
\textbf{fi} \\
\vdots
\end{array}
$$

Figure 18.7 Distribute assignment over alternation

18.5.3 Laws of alternation

The true and false guards of Figure 18.7 are handled with these laws; the result is Figure 18.8, in which we have merged the assignments as well.

Law 18.4 remove false guard

$$
\begin{array}{ll}
& \textbf{if } ([\!]\; i \cdot G_i \rightarrow branch_i) \\
& [\!]\;\; \mathsf{false} \rightarrow branch \\
& \textbf{fi} \\
= & \textbf{if } ([\!]\; i \cdot G_i \rightarrow branch_i) \textbf{ fi} \;.
\end{array}
$$

□

Law 18.5 remove alternation

$$\textbf{if } \mathsf{true} \rightarrow branch \textbf{ fi} \quad = \quad branch \;.$$

□

Now we flatten the alternations with the following law. The result is Figure 18.9.

Law 18.6 flatten nested alternations

$$
\begin{array}{ll}
& \textbf{if } ([\!]\; i \cdot G_i \rightarrow \textbf{if } ([\!]\; j \cdot H_j \rightarrow branch_{ij}) \textbf{ fi}) \textbf{ fi} \\
= & \textbf{if } ([\!]\; i, j \cdot G_i \wedge H_j \rightarrow branch_{ij}) \textbf{ fi} \;.
\end{array}
$$

□

$$
\begin{array}{l}
\vdots \\
\textbf{if } d \le 0 \rightarrow x, d := as[i], 1 \\
[\!]\ \ d > 0 \rightarrow \\
\qquad \textbf{if } as[i] = x \rightarrow d := d + 1 \\
\qquad [\!]\ \ as[i] \ne x \rightarrow d := d - 1 \\
\qquad \textbf{fi} \\
\textbf{fi} \\
\vdots
\end{array}
$$

Figure 18.8 Simplify alternation

$$
\begin{array}{l}
\vdots \\
\textbf{if } d \le 0 \rightarrow x, d := as[i], 1 \\
[\!]\ \ d > 0 \wedge as[i] = x \rightarrow d := d + 1 \\
[\!]\ \ d > 0 \wedge as[i] \ne x \rightarrow d := d - 1 \\
\textbf{fi} \\
\vdots
\end{array}
$$

Figure 18.9 Flatten nested alternations

18.5.4 Introducing invariants

Inspection of Figure 18.9, recalling its surrounding text, suggests that $d \ge 0$ is invariant. It is true initially, and is maintained by every assignment in the program. That takes us to Figure 18.10. Note that the type of d is now $\mathbb{N}$.

$$
\begin{array}{l}
\vdots \\
\textbf{if } d = 0 \rightarrow x, d := as[i], 1 \\
[\!]\ \ d \ne 0 \wedge as[i] = x \rightarrow d := d + 1 \\
[\!]\ \ d \ne 0 \wedge as[i] \ne x \rightarrow d := d \ominus 1 \\
\textbf{fi} \\
\vdots
\end{array}
$$

Figure 18.10 Introduce invariant $d \ge 0$

$$
\begin{array}{l}
\vdots \\
\textbf{if } d = 0 \to x, d := as[i], d+1 \\
[\!]\ \ d \neq 0 \wedge as[i] = x \to x, d := as[i], d+1 \\
[\!]\ \ d \neq 0 \wedge as[i] \neq x \to d := d \ominus 1 \\
\textbf{fi} \\
\vdots
\end{array}
$$

Figure 18.11 Exploit guards

$$
\begin{array}{l}
\vdots \\
\textbf{if } d = 0 \vee as[i] = x \to x, d := as[i], d+1 \\
[\!]\ \ d \neq 0 \wedge as[i] \neq x \to d := d \ominus 1 \\
\textbf{fi} \\
\vdots
\end{array}
$$

Figure 18.12 Collapse branches

In fact, invariant introduction is a special case of the add variable transformation: we introduce *no* variables, but have a coupling invariant nevertheless. The law *augment assignment* 17.8 reduces to checking that assignments preserve the invariant (the list c is empty); the law *augment guard* 17.9 allows the invariant to simplify the guards.

If we exploit the guards, we can reach Figure 18.11, in which we have made two branches identical. The following law then takes Figure 18.11 to Figure 18.12:

Law 18.7 <u>collapse identical branches</u>

$$
\begin{array}{ll}
 & \textbf{if } ([\!]\ i \bullet G_i \to \mathit{branch}_i) \\
 & [\!]\ \ G \to \mathit{branch} \\
 & [\!]\ \ G' \to \mathit{branch} \\
 & \textbf{fi} \\
= & \textbf{if } ([\!]\ i \bullet G_i \to \mathit{branch}_i) \\
 & [\!]\ \ G \vee G' \to \mathit{branch} \\
 & \textbf{fi}\ .
\end{array}
$$

□

```
|[ var i, d : ℕ•

   i, d := 0, 0;
   do i ≠ N →
       if d = 0 ∨ as[i] = x
          then x, d := as[i], d + 1
          else d := d ⊖ 1
       fi;
       i := i + 1
   od
]|
```

Figure 18.13 Simplified code

18.6 Simplified code

With Figure 18.12 we reach the end of the development. The code is collected in Figure 18.13, where we use the conventional **if** $\cdots$ **then** $\cdots$ **else** $\cdots$ **end**.

Curiously, we have iterative code but have 'lost' the invariant. Where has it gone?

The last invariant quoted was on p.189:

$$\begin{array}{r} j \le i \le N \\ \neg\,\mathsf{em}\ as[0\to j] \\ \mathsf{cm}\ x\ \ as[j\to i]\ . \end{array}$$

Introducing c adds a conjunct to that; introducing d adds another:

$$\begin{array}{r} j \le i \le N \\ \neg\,\mathsf{em}\ as[0\to j] \\ \mathsf{cm}\ x\ \ as[j\to i] \\ c = as[j\to i].x \\ d = 2c - (i-j)\ . \end{array}$$

Removing c, j removes them from the invariant, leaving this:

$$\left(\exists\, c, j : \mathbb{N} \cdot \begin{array}{r} j \le i \le N \\ \neg\,\mathsf{em}\ as[0\to j] \\ \mathsf{cm}\ x\ \ as[j\to i] \\ c = as[j\to i].x \\ d = 2c - (i-j) \end{array}\right) . \tag{18.2}$$

Since an invariant is unaffected by refinements to the iteration *body*, formula (18.2) is the invariant for the final program. (See Exercise 18.6.)

18.7 Exercises

Ex. 18.1 ♡ (From p.186) Prove that

$$\neg\,\mathsf{em}\ as{\uparrow}i \quad\Rrightarrow\quad \mathsf{cm}\ as[i]\ as{\uparrow}(i+1)\ .$$

Ex. 18.2 (From p.186) Prove that

$$\mathsf{sm}\ x\ as{\uparrow}i \quad\Rrightarrow\quad \mathsf{cm}\ x\ as{\uparrow}(i+1)\ .$$

Ex. 18.3 Why isn't **initially** false a good idea in a local block? *Hint*: Recall *local block initialisation* 18.1.

Ex. 18.4 ♡ (From p.187) Work through the details of showing that transformation is valid for local blocks. *Hint*: Introduce a module with a single parameterless procedure, called once.

Ex. 18.5 ♡ Check the claims made about simplifying guards (pp. 187 and 189).

Ex. 18.6 Show that the monstrous formula (18.2) entails

$$\begin{array}{ll} & i \le N \\ & as[0{\to}i].x \le (i+d)/2 \\ \text{for all } y \ne x & as[0{\to}i].y \le (i-d)/2\ . \end{array}$$

Using that as an invariant, develop the code of Figure 18.13 directly.

Ex. 18.7 Use *alternation guards* 4.3, *remove false guard* 18.4, and *remove alternation* 18.5 to prove this law:

Law 18.8 <u>select true guard</u>

if $(\,[\!]\ i \cdot G_i \to branch_i)$
$[\!]$ true $\to branch$
fi

$\sqsubseteq\ branch\ .$

□