Chapter 5

Iteration

5.1 Operational description

Iterations (sometimes called '*while* loops') implement repetition: typically a command is executed repeatedly while a certain condition holds. In their general form iterations are, like alternations, built from guarded commands. We write them as follows:

$$
\begin{array}{ll}
\textbf{do} & G_0 \rightarrow prog_0 \\
[\!] & G_1 \rightarrow prog_1 \\
 & \vdots \\
[\!] & G_n \rightarrow prog_n \\
\textbf{od} & .
\end{array}
$$

We can also write $\textbf{do}\ ([\!]i \cdot G_i \rightarrow prog_i)\ \textbf{od}$.

The repetition occurs in this way. In the initial state, none, one, or several guards will be true. If none is true, the command terminates successfully and the state is unchanged. (Note the difference from alternation, which aborts in that case.)

If one or several guards are true, just one is chosen and its corresponding command is executed. Then the process is repeated, beginning with a re-evaluation of all the guards.

It is possible (but usually undesirable) for iterations to repeat forever. From the above, we see that they terminate only when all guards are false; thus as an extreme example (of the opposite)

$$\textbf{do}\ \mathsf{true} \rightarrow \textbf{skip}\ \textbf{od}$$

never terminates. By convention, a never-ending iteration is equivalent to **abort**.

Successful iterations finally make all their guards false, and that may occur from some initial states but not others. Consider the following program, which for natural number n establishes $n = 1$ finally whenever n is a power of 2 initially:

$$\textbf{do}\ 2 \mid n \rightarrow n := n \div 2\ \textbf{od}\ . \tag{5.1}$$

If n is not a power of two initially, then we are not assured of $n = 1$ finally: starting with $n = 12$ for example would lead via $n = 6$ to $n = 3$, where the iteration would terminate successfully because 2 does not divide 3 exactly. And if $n = 0$ initially, there is no finally: variable n is set and reset to 0 forever.

Even from our informal view, we see that an iteration can be *unfolded* without affecting its meaning:

$$
\begin{array}{ll}
 & \textbf{do}\ G \rightarrow \mathit{prog}\ \textbf{od} \\
= & \textbf{if}\ G\ \textbf{then} \\
 & \quad \mathit{prog}; \\
 & \quad \textbf{do}\ G \rightarrow \mathit{prog}\ \textbf{od} \\
 & \textbf{fi}\ .
\end{array}
$$

Each unfolding makes one more repetition explicit. Thus we can unfold again...

$$
\begin{array}{ll}
= & \textbf{if}\ G\ \textbf{then} \\
 & \quad \mathit{prog}; \\
 & \quad \textbf{if}\ G\ \textbf{then} \\
 & \quad\quad \mathit{prog}; \\
 & \quad\quad \textbf{do}\ G \rightarrow \mathit{prog}\ \textbf{od} \\
 & \quad \textbf{fi} \\
 & \textbf{fi}\ .
\end{array}
$$

...any number of times, and that shows that an iteration can be regarded as equivalent to an unbounded nesting of alternations.

5.2 Refinement law: informally

Rather than unfolding as above, we rely instead on a refinement law that abstracts from the number of repetitions: it is all captured in a single formula, the invariant. An *invariant* for an iteration is a formula which, if true initially, is true also after every repetition including the final one. Overall, the iteration *maintains* the invariant, and establishes additionally the negation of the guards — provided it terminates at all. And that is true no matter how many repetitions occur (even 0).

For an example we return to Program (5.1), defining a predicate pt to mean 'is a power of 2':

$$\mathsf{pt}\, n \ \widehat{=}\ \left(\exists\, k : \mathbb{N} \cdot n = 2^k\right).$$

The formula $\mathsf{pt}\,n$ is an invariant for the iteration: if it is true initially, and a repetition occurs (because $2 \mid n$ is true also), then after execution of the iteration body $n := n \div 2$ it is true still. Another way of writing that is

$$n\colon [2 \mid n \wedge \mathsf{pt}\,n \;,\; \mathsf{pt}\,n] \quad \sqsubseteq \quad n := n \div 2 \;. \tag{5.2}$$

In general, a formula *inv* is an invariant of **do** $G \rightarrow prog$ **od** if for some frame w

$$w\colon [G \wedge inv \;,\; inv] \quad \sqsubseteq \quad prog \;. \tag{5.3}$$

That is, if the guard G holds, then the iteration body *prog* preserves the invariant *inv*.

The utility of an invariant is that, assuming termination, its preservation as in (5.3) is sufficient to establish

$$w\colon [inv \;,\; inv \wedge \neg G] \quad \sqsubseteq \quad \mathbf{do}\ G \rightarrow prog\ \mathbf{od} \;. \tag{5.4}$$

And in (5.4) we have the essence of our refinement law, which will allow us to replace a specification by an iteration. As an example, we derive (5.1) from the informal specification given earlier:

$$
\begin{array}{ll}
 & n\colon [\mathsf{pt}\,n \;,\; n = 1] \\
\sqsubseteq & \text{“1 is the only power of 2 not divisible by 2”} \\
 & n\colon [\mathsf{pt}\,n \;,\; \mathsf{pt}\,n \wedge \neg(2 \mid n)] \\
\sqsubseteq & \text{“(5.4), justified by (5.2) in this case”} \\
 & \mathbf{do}\ 2 \mid n \rightarrow n := n \div 2\ \mathbf{od} \;.
\end{array}
$$

Our refinement rule ‘if (5.3) then (5.4)’ is not yet complete, however: in fact if used in that form it would produce ‘refinements’ that were invalid. That is because (an extreme case) we could reason

$$
\begin{array}{rl}
\text{because} & w\colon [\mathsf{true} \wedge \mathsf{true} \;,\; \mathsf{true}] \quad \sqsubseteq \quad \mathbf{skip} \\
\text{we ‘conclude’} & w\colon [\mathsf{true} \;,\; \mathsf{true} \wedge \mathsf{false}] \quad \sqsubseteq \quad \mathbf{do}\ \mathsf{true} \rightarrow \mathbf{skip}\ \mathbf{od} \;.
\end{array}
$$

Put more starkly, we would be claiming that

$$
\begin{array}{rl}
\text{because} & \mathbf{choose}\ w \quad \sqsubseteq \quad \mathbf{skip} \\
\text{we ‘conclude’} & \mathbf{magic} \quad \sqsubseteq \quad \mathbf{do}\ \mathsf{true} \rightarrow \mathbf{skip}\ \mathbf{od} \;.
\end{array}
$$

That is clearly nonsense: the premise is true, but the conclusion is false.

What we are missing is the idea of termination: that the iteration body not only maintains the invariant but cannot be executed indefinitely. For that we need a variant function.

5.3 Termination of iterations: variants

In the example, termination is guaranteed informally by the following observation:

> Each repetition of $n := n \div 2$, when $2 \mid n$, strictly decreases the integer n; yet n cannot become negative.

We say therefore that the variant of the iteration is n. In general, some integer-valued[1] variant expression is chosen that is strictly decreased by each repetition, but never below some fixed lower bound (often chosen to be 0).

5.3.1 Specifying decrease of the variant

We have no difficulty writing a command that decreases an integer-valued variable n — the assignment $n := n - 1$ is one of many that would do — but to *specify* that n is strictly decreased is another story. Writing

$$n\colon [n = n - 1] \ ,$$

or more generally $n\colon [n < n]$, would *not* do: both of them are equivalent to **magic**.

We adopt a convention that 0-subscripted variables in a postcondition refer to the *initial* (rather than to the final) values of the variables. With that, the above discussion could be summarised

$$n\colon [n < n_0] \quad \sqsubseteq \quad n := n - 1 \ . \tag{5.5}$$

That is, the assignment $n := n - 1$ (the more refined side) is just one of many ways of strictly decreasing n (expressed by the specification on the left-hand side).

We will meet initial variables in more generality shortly. For now we re-examine just three of our earlier laws, generalising them to take initial variables into account. They become

Law 5.1 strengthen postcondition If $pre[w \backslash w_0] \wedge post' \Rrightarrow post$, then

$$w\colon [pre \ , \ post] \quad \sqsubseteq \quad w\colon [pre \ , \ post'] \ .$$

□

Law 5.1 strictly generalises our earlier *strengthen postcondition* 1.1 in that the precondition of the left-hand side can now be taken into account.

Law 5.2 assignment If $(w = w_0) \wedge (x = x_0) \wedge pre \Rrightarrow post[w \backslash E]$, then

$$w, x\colon [pre \ , \ post] \quad \sqsubseteq \quad w := E \ .$$

□

[1] More general variants are possible; they range over well-founded sets.

Law 5.2 generalises our earlier *assignment* 1.3. Notice that the substitution $[w\backslash E]$ affects only 'final' (*un*subscripted) variables in the postcondition.

Law 5.3 skip command If $(w = w_0) \wedge pre \Rrightarrow post$, then

$$w\colon [pre \;,\; post] \quad \sqsubseteq \quad \mathbf{skip} \;.$$

□

Law 5.3 generalises *skip command* 3.2, taking advantage of the fact that in a **skip** command the initial and final variables have the same value.

Each of the above reduces to its earlier version if initial variables are not present, and the earlier versions remain valid even when initial variables are present — but they are not as powerful as the newer versions.

Using our new *assignment* 5.2 we can show Refinement (5.5) in just one step:

$$\begin{array}{ll} & n\colon [n < n_0] \\ \sqsubseteq & \text{“}n = n_0 \wedge \mathsf{true} \;\Rrightarrow\; (n < n_0)[n\backslash n-1]\text{”} \\ & n := n - 1 \;. \end{array}$$

Simplified, the proviso would be simply $n = n_0 \Rrightarrow n - 1 < n_0$.

5.3.2 Initial variables and the frame

We should also mention at this point an interaction between initial variables (values before any change) and the frame (listing which variables may be changed). If a variable x is not in the frame, then x and x_0 may be used interchangeably in the postcondition since the initial and final values of x are equal in that case. Put another way, if we remove x from the frame, then we can replace x_0 by x in the postcondition. That is summarised in the following law:

Law 5.4 contract frame

$$w, x\colon [pre \;,\; post] \quad \sqsubseteq \quad w\colon [pre \;,\; post[x_0\backslash x]] \;.$$

□

Note that x_0 need not occur in *post* for Law 5.4 to apply — and that means, as a special case, that simply removing a variable from the frame is a refinement too. (Recall Exercise 1.7.)

That concludes our look at initial variables; we return to them in Chapter 8.

5.4 The refinement rule for iteration

With initial variables available for specifying the decrease of the variant, we can now give the full refinement rule. It is

Law 5.5 iteration Let *inv*, the *invariant*, be any formula; let V, the *variant*, be any integer-valued expression. Then if GG is the disjunction of the guards,

$$\begin{array}{ll} & w\colon [inv \ , \ inv \wedge \neg GG] \\ \sqsubseteq & \mathbf{do}\ ([\!]\, i \cdot G_i \rightarrow w\colon [inv \wedge G_i \ , \ inv \wedge (0 \leq V < V_0)])\ \mathbf{od}\ . \end{array}$$

Neither *inv* nor G_i may contain initial variables. The expression V_0 is $V[w \backslash w_0]$.
□

In the interests of keeping the amount we must write to a minimum, we introduce an abbreviation that avoids writing *inv* twice in the iteration body. When a formula appears in both the pre- and the postcondition of a specification (as *inv* does in the iteration body above), it can instead be written once in between, as a third formula:

Abbreviation 5.6 specification invariant Provided *inv* contains no initial variables,

$$w\colon [pre \ , \ inv \ , \ post] \ \widehat{=} \ w\colon [pre \wedge inv \ , \ inv \wedge post] \ .$$

□

The iteration body in Law 5.5 can now be written (more briefly as) just

$$w\colon [G_i \ , \ inv \ , \ 0 \leq V < V_0] \ .$$

Figure 5.1 gives a complete development of Program (5.1), including the variant. In the simplified proviso we have used $n = n_0$ on the left-hand side to replace n_0 by n on the right.

5.5 The iteration 'checklist'

In planning one's approach to developing an iteration, it is sometimes useful to consider the following characteristics of iterations that are built in to the iteration law:

1. The invariant holds initially.
2. The invariant and negated guard are sufficient to establish the desired result.
3. The iteration body maintains the invariant provided the guard holds as well.
4. The variant is strictly decreased by execution of the iteration body, provided the invariant and guard hold.
5. The variant cannot be decreased below 0 by the iteration body, provided the invariant and guard hold.

$$
\begin{array}{ll}
& n\colon [\mathsf{pt}\, n\ ,\ n = 1] \\
\sqsubseteq & n\colon [\mathsf{pt}\, n\ ,\ \mathsf{pt}\, n \wedge \neg(2 \mid n)] \\
\sqsubseteq & \mathbf{do}\ 2 \mid n \rightarrow \\
& \qquad n\colon [2 \mid n\ ,\ \mathsf{pt}\, n\ ,\ 0 \le n < n_0] \qquad \triangleleft \\
& \mathbf{od} \\
\sqsubseteq & \text{“}2 \mid n \wedge \mathsf{pt}\, n \ \Rrightarrow\ \mathsf{pt}(n \div 2) \wedge 0 \le n \div 2 < n\text{”} \\
& n := n \div 2\ .
\end{array}
$$

Figure 5.1 Example iteration development

Characteristic 1 is found in the precondition of the specification on the left-hand side: by '$\cdots [inv, \cdots$' we express that the invariant must hold initially. Similarly, Characteristic 2 is found in the postcondition: writing '$\cdots, inv \wedge \neg GG] \cdots$' states what is required of the iteration.

In '$\cdots [G_i \wedge inv, inv \wedge \cdots$' of the iteration body is found Characteristic 3; Characteristic 4 is expressed by '$\cdots [G_i \wedge inv, \cdots V < V_0]$', and finally Characteristic 5 is given by '$\cdots [G_i \wedge inv, \cdots 0 \le V \cdots$'.

When choosing an invariant it is sometimes helpful to run through the checklist[2] informally, before setting out the development in full — one is then (literally) carrying out a feasibility study.

5.6 Exercises

Ex. 5.1 Give a single assignment command that refines

$$n\colon [\mathsf{pt}\, n\ ,\ n = 1]\ .$$

Ex. 5.2 ♡ *Checking for powers of two* Use the invariant $n \ne 0 \wedge (\mathsf{pt}\, N \Leftrightarrow \mathsf{pt}\, n)$ to complete the following development, in which N is used to hold the original value of n:

$$
\begin{array}{ll}
& n\colon [n \ne 0 \wedge n = N\ ,\ n = 1 \Leftrightarrow \mathsf{pt}\, N] \\
\sqsubseteq & n\colon [n \ne 0 \wedge (\mathsf{pt}\, N \Leftrightarrow \mathsf{pt}\, n)\ ,\ (\mathsf{pt}\, N \Leftrightarrow \mathsf{pt}\, n) \wedge \neg(2 \mid n)] \\
\sqsubseteq & \cdots
\end{array}
$$

Hint: You have seen the code before.

Ex. 5.3 ♡ Consider this factorial program, in which we assume f and n are integers. What laws are used for the first refinement step (shown)? (The constant F is used to refer to the desired factorial value.)

[2] It is borrowed from [Gri81].

$$\begin{array}{ll} & f, n\colon [F = n!\ ,\ f = F] \\ \sqsubseteq & f\colon [F = n!\ ,\ f \times n! = F]\,; \qquad \text{(i)} \\ & f, n\colon [f \times n! = F\ ,\ f \times n! = F \wedge n = 0]\ . \qquad \text{(ii)} \end{array}$$

Complete the refinement to code.

Ex. 5.4 ♡ The law *strengthen postcondition* 5.1 is stronger than *strengthen postcondition* 1.1 because it uses information from the precondition. Assuming $x, y : \mathbb{R}$, use it to show that

$$\begin{array}{ll} & y\colon [0 \le x \le 9\ ,\ y^2 = x] \\ \sqsubseteq & y\colon [0 \le x \Rightarrow y^2 = x]\ . \end{array}$$

Explain the fact that the law is called 'strengthen postcondition', yet above the new postcondition is weaker than the old.

Ex. 5.5 ♡ Assuming $x, y : \mathbb{R}$, prove each of the following:

1. $x\colon [y > x\ ,\ x > x_0] \sqsubseteq x := y$
2. $x\colon [x < 0\ ,\ x > x_0] \sqsubseteq x := -x$
3. $x, y\colon [x = y_0 \wedge y = x_0] \sqsubseteq x, y := y, x$
4. $x\colon [x = X + 1\ ,\ x = X + 2] \sqsubseteq x := x + 1$
5. $x\colon [x = X + 1\ ,\ x = X + 2] \sqsubseteq x\colon [x = x_0 + 1]$
6. $$\begin{array}{ll} & x\colon [x = x_0 + 2] \\ \sqsubseteq & x\colon [x = x_0 + 1]\,; \\ & x\colon [x = x_0 + 1]\ . \end{array}$$

Ex. 5.6 ♡ Design an *initialised iteration* law of this form:

$$\begin{array}{ll} & w\colon [\mathit{pre}\ ,\ \mathit{inv} \wedge \neg G] \\ \sqsubseteq & \mathit{prog}1; \\ & \mathbf{do}\ G \rightarrow \mathit{prog}2\ \mathbf{od}\ . \end{array}$$

You should supply *prog*1 and *prog*2, as specifications, and you may assume that *inv* and G contain no initial variables.

Ex. 5.7 *Logarithm* Here is a specification of a logarithm-finding program, in which l, n and N are integers:

$$l, n\colon \left[1 \le n = N\ ,\ 2^l \le N < 2^{l+1}\right]\ .$$

Variable N holds the initial value of n. Develop the specification to code using the invariant

$$n \times 2^l \le N < (n + 1) \times 2^l \wedge 1 \le n\ .$$

Ex. 5.8 ♡ *Handing out sweets* Suppose S sweets are to be handed out to C children. If C divides S exactly, then each child should receive S/C (whole!) sweets. But if the division is not exact, then some will receive $\lfloor S/C \rfloor$ and others $\lceil S/C \rceil$, where $\lfloor\ \rfloor$ and $\lceil\ \rceil$ are the *floor* and *ceiling* functions that take a real number to the closest integer no more than and no less than it respectively.

Here is a program for handing out the sweets, using natural number variables s, c and t:

$$
\begin{array}{l}
s, c := S, C; \\
\textbf{do } c \neq 0 \rightarrow \\
\quad t\!: [\ \lfloor s/c \rfloor \leq t \leq \lceil s/c \rceil\ ]; \\
\quad \text{`hand out } t \text{ sweets to the next child'}; \\
\quad s, c := s - t, c - 1 \\
\textbf{od}\ .
\end{array}
$$

We say the handing out is *fair* if

$$\text{all the sweets are handed out; and} \tag{5.6}$$

$$\text{each child receives between } \lfloor S/C \rfloor \text{ and } \lceil S/C \rceil \text{ sweets.} \tag{5.7}$$

Does the program implement a fair handing out?

Hint: Use *informal* invariant-based reasoning, following the checklist of Section 5.5; consider an invariant resembling[3]

$$\lfloor S/C \rfloor \leq \lfloor s/c \rfloor \tag{5.8}$$

$$\lceil s/c \rceil \leq \lceil S/C \rceil \tag{5.9}$$

$$S = s + \text{`the number of sweets handed out already'}\ . \tag{5.10}$$

Useful facts about $\lfloor\ \rfloor$ and $\lceil\ \rceil$ are that for all integers i and reals r

$$
\begin{array}{rrcl}
 & i \leq \lfloor r \rfloor & \equiv & i \leq r \\
\text{and} & \lceil r \rceil \leq i & \equiv & r \leq i\ .
\end{array}
$$

[3] You may have to alter it slightly...