Chapter 4

Alternation

4.1 Operational description

Alternations (sometimes called '*if* statements') can informally be said to implement a case analysis: based on the initial state, one of several possible commands is selected for execution. They are built from *guarded commands*, each comprising a guard and an associated program called the *command*. A *guard* is a formula which selects those states to which its associated command applies. The guarded command itself is written

$$G \rightarrow prog \ ,$$

and it is pronounced 'G then *prog*'; the guard is G and the command is *prog*.

An *alternation* is a collection of guarded commands grouped together. They are separated by the symbol $[\!]$ (pronounced 'else') and enclosed in the alternation brackets **if** and **fi**. Here is an example:

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

We also write the above as **if** $([\!]\, i \cdot G_i \rightarrow prog_i)$ **fi**, with the limits 0 and n understood from context.

The case analysis occurs in this way. In the initial state, none, one, or several of the guards G_i will be true. If exactly one is true, its corresponding command is executed.

If several guards are true, then one of them is selected and its corresponding command is executed. If the alternation has been properly developed, it will not matter which of the several guards is chosen — and one can make no assumptions about which it will be.

If no guard is true, the alternation aborts: in that case it can do *anything*, and one may assume that 'anything' means 'something bad'. (Recall the discussion of **abort** in Section 1.9.) That is not usually the programmer's intention (unless it was the client's too); a proper development will avoid it.

Consider this alternation for calculating the maximum m of two integers a and b:

$$\begin{array}{l} \textbf{if } a \geq b \rightarrow m := a \\ [\!]\;\; b \geq a \rightarrow m := b \\ \textbf{fi} \; . \end{array}$$

There are two cases — they are $a \geq b$ and $b \geq a$ — and they overlap. Together, they cover all possibilities, thus avoiding **abort**. Whenever $a > b$, the first command is executed; whenever $b > a$, the second is executed. If $a = b$, then *either* can be executed, and of course in this alternation it makes no difference which. (Since the problem is symmetric, it is especially appropriate to have a symmetric solution.)

As a second example, consider

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

The formula '$2 \mid x$' means '2 divides x exactly'. If x is initially 2, then finally it will be 1; if initially 3, then the result is the same (but by different means!) If initially x is 6, then finally it could be either 2 or 3, and we cannot predict which. And if initially 7, finally it could be 17... or 289, or nothing at all (because of nontermination).

4.2 Refinement law

Since we deal with the general case, any number of guarded commands, we introduce a notation for the disjunction of all their guards. The name GG abbreviates the formula

$$G_0 \vee G_1 \vee \cdots \vee G_n$$

in the following refinement law:

Law 4.1 alternation If $pre \Rrightarrow GG$, then

$$\begin{array}{ll} & w\colon [pre \; , \; post] \\ \sqsubseteq & \textbf{if } ([\!]\; i \cdot G_i \rightarrow w\colon [G_i \wedge pre \; , \; post]) \textbf{ fi} \; . \end{array}$$

□

The precondition of the specification ensures that at least one guard is true; and each command assumes additionally in its precondition the truth of its guard.

Now let us reconsider the maximum-finding program. Using the binary operator $\sqcup$ for *maximum*, we begin

$$
\begin{array}{ll}
 & m := a \sqcup b \\
= & \text{“}\textit{simple specification } 1.7\text{”} \\
 & m\colon [m = a \sqcup b] \\
\sqsubseteq & \text{“}\textit{alternation } 4.1\text{”} \\
 & \mathbf{if}\ a \geq b \rightarrow m\colon [a \geq b\ ,\ m = a \sqcup b] \qquad \text{(i)} \\
 & [\!]\ \ b \geq a \rightarrow m\colon [b \geq a\ ,\ m = a \sqcup b] \qquad \text{(ii)} \\
 & \mathbf{fi}\ .
\end{array}
$$

Note that each command has in its precondition the corresponding guard.

The numbering (i) and (ii) is to allow those commands to be refined separately below. As with $\triangleleft$ (a special case), the context is carried forward, and the code can be collected at the end. Continuing, we use *assignment* 1.3 to refine each command:

$$
\begin{array}{ll}
\text{(i)} \sqsubseteq & m := a \\
\text{(ii)} \sqsubseteq & m := b\ \ .
\end{array}
$$

The resulting program is as shown earlier.

It is deliberate that alternation has no provision for defaults; each case must be explicitly mentioned. If in some case there is ‘nothing to do’ — because the initial state will serve as the final state — then **skip** is the appropriate command. Consider this development:

$$
\begin{array}{ll}
 & m := a \sqcup b \sqcup c \\
\sqsubseteq & \text{“}\textit{simple specification } 1.7;\ \textit{sequential composition } 3.3\text{”} \\
 & m := a; \\
 & m\colon [m = a\ ,\ m = a \sqcup b \sqcup c] \qquad \triangleleft \\
\sqsubseteq & \text{“}\textit{sequential composition } 3.3\text{”} \\
 & m\colon [m = a\ ,\ m = a \sqcup b]\,; \qquad \text{(i)} \\
 & m\colon [m = a \sqcup b\ ,\ m = a \sqcup b \sqcup c] \qquad \text{(ii)} \\
\text{(i)} \sqsubseteq & \text{“}\textit{alternation } 4.1\text{”} \\
 & \mathbf{if}\ m \leq b \rightarrow m\colon [m = a \wedge m \leq b\ ,\ m = a \sqcup b] \qquad \text{(iii)} \\
 & [\!]\ \ m \geq b \rightarrow m\colon [m = a \wedge m \geq b\ ,\ m = a \sqcup b] \qquad \text{(iv)} \\
 & \mathbf{fi} \\
\text{(iii)} \sqsubseteq & m := b \\
\text{(iv)} \sqsubseteq & \mathbf{skip} \\
\text{(ii)} \sqsubseteq & \mathbf{if}\ m \leq c \rightarrow m := c\ [\!]\ m \geq c \rightarrow \mathbf{skip}\ \mathbf{fi}\ .
\end{array}
$$

The resulting code, collected from the derivation above, is

$$\begin{array}{l} m := a; \\ \textbf{if } m \le b \to m := b \mathbin{[\!]} m \ge b \to \textbf{skip fi}; \\ \textbf{if } m \le c \to m := c \mathbin{[\!]} m \ge c \to \textbf{skip fi} . \end{array}$$

If we abbreviate **if** $G \to \mathit{prog} \mathbin{[\!]} \neg G \to$ **skip fi** by

$$\textbf{if } G \textbf{ then } \mathit{prog} \textbf{ fi} ,$$

then we can write the above

$$\begin{array}{l} m := a; \\ \textbf{if } m \le b \textbf{ then } m := b \textbf{ fi}; \\ \textbf{if } m \le c \textbf{ then } m := c \textbf{ fi} . \end{array}$$

4.3 Exercises

Ex. 4.1 ♡ Assuming that x and y are real numbers, refine

$$x := \mathsf{abs}\, y$$

to an alternation, thence to code.

Ex. 4.2 Assume that x and y are real numbers, and supposing that $\surd$ takes reals to reals and *always terminates*, refine this to code:

$$y\colon \left[x \ge 0 \Rightarrow y^2 = x\right] .$$

You may use $\surd$ in expressions.

Ex. 4.3 *Mortgage* Let c, s, b, m be respectively the cost of a house, savings in the bank, borrowing limit, and mortgage granted; they are all integers. Specify using maximum $\sqcup$ and minimum $\sqcap$ a program that determines m in terms of c, s, b. Then refine that program to code in which $\sqcup$ and $\sqcap$ do not appear.

Ex. 4.4 ♡ Prove *alternation* 4.1 from this law:

Law 4.2 <u>alternation</u>

$$\begin{array}{ll} & \{(\bigvee i \cdot G_i)\}\ \mathit{prog} \\ = & \textbf{if } ([\!]\ i \cdot G_i \to \{G_i\}\ \mathit{prog}) \textbf{ fi} . \end{array}$$

□

Ex. 4.5 Sometimes one wants to refine one alternation to another, simply re-arranging the guards. The following law can be used for that:

> *Law 4.3* <u>alternation guards</u> Let GG mean $G_0 \vee \cdots \vee G_n$, and HH similarly. Then provided
>
> 1. $GG \Rrightarrow HH$, and
> 2. $GG \Rrightarrow (H_i \Rightarrow G_i)$ for each i separately,
>
> this refinement is valid:
>
> $$\mathbf{if}\ ([\!]\ i \cdot G_i \rightarrow prog_i)\ \mathbf{fi} \quad \sqsubseteq \quad \mathbf{if}\ ([\!]\ i \cdot H_i \rightarrow prog_i)\ \mathbf{fi}\ .$$
>
> □

Use Law 4.3 to show that the second example of Section 4.1 can be refined to

$$\mathbf{if}\ 2 \mid x\ \mathbf{then}\ x := x \div 2\ \mathbf{else}\ x := x \div 3\ \mathbf{fi}\ .$$

where **if** G **then** *prog*1 **else** *prog*2 **fi** abbreviates

$$\begin{array}{ll} \mathbf{if} & G \rightarrow prog1 \\ [\!] & \neg G \rightarrow prog2 \\ \mathbf{fi}\ . & \end{array}$$

Does that mean that we can write our nondeterministic alternations in the programming language C?