Chapter 3

Assignment and sequential composition

3.1 Introduction

In Chapter 1 we saw that *code* is a sub-language in which we write programs that are executable by computer directly. All conventional computer programming languages are examples of code, because all of them were designed to be executed by computers. At least part of our language, however, was designed for program *development*: and so we must be explicit about which part of it is code, and which is not.

Our code will be written in a language that includes assignment, sequential composition, alternation, iteration and recursion. All of those have more or less their conventional meaning, but we do not explain them only in the conventional way.

Each executable construct is introduced in *two* ways: informally, as an operation on some computer; and formally, as a refinement of some specification. The first may aid the intuition, but is not essential for program development. Nor is it sufficient. The second way is essential for program development, however, because only that defines precisely how to reach code from specifications.

In this chapter we meet our first examples of code: assignment, and sequential composition. Others will follow in succeeding chapters.

3.2 Assignment

Informally, an *assignment* changes a single variable, leaving all others unchanged. An expression E is evaluated in the initial state; then in the final state, a variable w is newly mapped to that value, irrespective of its mapping in the initial state. (The initial value of the variable is lost.) Such assignments are written

$$w := E \ ,$$

$x := 1$	Assign 1 to x.
$x := y$	Assign the (initial) value of y to x.
$x := x + 1$	Assign the initial value of $x + 1$ to x — that is, increment x.
$x := y \times (x + y)$	A more complex expression.

Each of the above assignments takes any initial state that satisfies $x = 0 \wedge y = 1$ to a final state satisfying $x = 1 \wedge y = 1$, and changes no variable other than x.

Figure 3.1 Simple assignments

and they are read 'w gets E'. The expression E is built from any constants, variables, and operators available to us; later we will be more explicit about which they are. Figure 3.1 gives some examples of assignments.

A *multiple* assignment changes several variables at once. It is written

$$w_0, \cdots, w_n := E_0, \cdots, E_n \ .$$

That command assigns E_0 to w_0, $\cdots$, and E_n to w_n simultaneously. (Assigning E_0 to w_0 first, and then later E_n to w_n, does not in general have the same effect.) Figure 3.2 gives examples of multiple assignments. When discussing assignments in general we use the simple form $w := E$, and allow w and E to be lists if appropriate.

We have already met the refinement law for assignment as *assignment* 1.3 in Section 1.5.2 — think of that as a preview of this chapter. The law is based on the observation that in an assignment $w := E$, only w is changed; then *post* describes the final state provided that $post[w \backslash E]$ described the initial state.

Law *assignment* 1.3 shows that each of the assignments of Figure 3.1 refines the specification

$$x\colon [x = 0 \wedge y = 1 \ , \ x = 1 \wedge y = 1] \ ,$$

and similarly each of the multiple assignments of Figure 3.2 refines the specification

$$x, y\colon [x = 0 \wedge y = 1 \ , \ x = 1 \wedge y = 0] \ .$$

In Figure 3.3 there are more examples of refinement to assignment.

$x, y := 1, 0$	Assign 1 to x and 0 to y.
$x, y := y, x$	Swap x and y.
$x, y := x + y, x \times y$	Assign the initial value of $x + y$ to x and the initial value of $x \times y$ to y.

Each of the above assignments takes any initial state that satisfies $x = 0 \wedge y = 1$ to a final state satisfying $x = 1 \wedge y = 0$, and changes no variables other than x and y.

Figure 3.2 Multiple assignments

3.3 Open assignment

A slightly mysterious-looking form of (multiple) assignment, but one that we shall find useful later, assigns any value whatever: the command $w, x := E, ?$ assigns E to w but leaves open the value that will be assigned to x. For that reason we call it *open assignment*. (We assume as usual that w and x are disjoint lists of variables.)

Open assignment is used mainly in specifying desired behaviour — although it is code, in fact — because it is a convenient way of writing 'and x may be changed'. Thus the assignment $r, s := \sqrt{s}, ?$ is a command that sets r to the square root of the value found initially in s, and may change s in the process. Similarly, the command $x, y, t := y, x, ?$ swaps the values of x and y using (possibly) a temporary variable t along the way.

The refinement rule for open assignment simply allows the '?' to be replaced by any expression:

Law 3.1 <u>open assignment</u> For any expression F,

$$w, x := E, ? \quad \sqsubseteq \quad w, x := E, F \ .$$

□

The command **choose** x from Chapter 1 is a special case of open assignment, in which the list w is empty: we could just as well write $x := ?$.

3.4 The skip command

Another unusual command is **skip**, which does nothing: its final state is exactly the same as its initial state. (It was mentioned in Section 1.9.) It can also be regarded

$$x\colon [x = 1] \quad \sqsubseteq \quad x := 1$$

$$x, y\colon [x = X \wedge y = Y \ , \ x = Y \wedge y = X] \quad \sqsubseteq \quad x, y := y, x$$

$$x\colon [y \neq 0 \ , \ x = 1/y] \quad \sqsubseteq \quad x := 1/y$$

$$x\colon [\mathsf{false} \ , \ x = 0] \quad \sqsubseteq \quad x := 17$$

Figure 3.3 Refinement to assignment

as a degenerate assignment, in which the list of changed variables is empty. (It can even be considered to be an assignment of variables to themselves, as in $x := x$.) Its refinement law is as follows:

Law 3.2 skip command If *pre* $\Rrightarrow$ *post*, then

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

□

We shall see later that **skip** is a useful command in spite of its doing 'nothing'. (After all, the same holds for '0'.)

3.5 Sequential composition

So far, we have the atomic programs specification, assignment and **skip**. The latter two are code; the first is not. But they are all called *atomic* because they are not formed from still smaller programs: instead, they cannot be broken down any further. Viewing a large program from the bottom up, one first sees atomic programs put together to make *compound* programs. Then those compound programs are themselves put together to form larger compound programs, and so on.

Sequential composition, which we met in Section 1.6, is one way of putting programs together. Informally, the sequential composition of two programs *prog*1 and *prog*2 is a new program which 'first does *prog*1 and then does *prog*2'. It is written *prog*1 ; *prog*2, and operationally one thinks of 'control flowing from left to right'. Figure 3.4 gives examples of sequential compositions.

Sequential composition is an operator which, like addition of numbers for example, is written between its operands. We include it in this chapter on code because it is executable in the sense that if *prog*1 and *prog*2 are executable, then *prog*1; *prog*2 is as well — first *prog*1 is executed, then *prog*2.

In using a *law* for sequential composition, rather than informal reasoning, we are adopting a top-down view instead of bottom-up. A single specification is refined by

$x := 0;\ \ x := 1$	Assign 0 to x, then assign 1 to x.
$x := 1;\ \ y := x$	Assign 1 to x, then assign that (new) value of x to y.
$x := y;\ \ y := x$	Assign the initial value of y to x, then assign that (new) value to y.
$y := y;\ \ x := y$	Assign y to itself (no change), then assign that (same) value to x.

Each of these programs takes any initial state that satisfies $x = 0 \wedge y = 1$ to a final state satisfying $x = 1 \wedge y = 1$.

Figure 3.4 Sequential composition

the sequential composition of two others; and they, in turn, are refined by others still. Here is the law:

Law 3.3 sequential composition For any[1] formula *mid*,

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

□

The intuition operating here is that one way of reaching a final state satisfying *post* from an initial state satisfying *pre* is to proceed in two stages, via an intermediate state satisfying *mid*.

The intermediate *mid* can be any formula whatever: if it is strong (tending to false), then the first component in Law 3.3 is hard to refine subsequently, but the second is easy. If *mid* is weak (tending to true), then the reverse is the case. But any choice of *mid* is allowed, even true and false themselves. Figure 3.5 gives an example of Law 3.3; the resulting program hierarchy is shown in Figure 3.6.

For sequential composition with **skip** specifically, we have the following law:

Law 3.4 skip composition For any program *prog*,

$$prog;\ \mathbf{skip} \quad = \quad \mathbf{skip};\ prog \quad = \quad prog\ .$$

□

Note that the three programs are *equal*: each refines both of the others.

[1]Neither *mid* nor *post*, however, may contain the so-called ‘initial variables’ that are the subject of Chapter 8 to come. That does not at all concern us now, but must be remembered if ever referring to this law later, once they have been introduced. Law B.2 on page 275 is the most appropriate replacement for the more general case.

$$
\begin{array}{ll}
& x, y\colon [x = 0 \wedge y = 1] \\
\sqsubseteq & \text{“\textit{sequential composition} 3.3”} \\
& x, y\colon [x = 0]\,; \\
& x, y\colon [x = 0 \;,\; x = 0 \wedge y = 1]\,.
\end{array}
$$

We could refine each of those further with *assignment* 1.3, as follows:

$$
\begin{array}{rcl}
x, y\colon [x = 0] & \sqsubseteq & x := 0 \\
x, y\colon [x = 0 \;,\; x = 0 \wedge y = 1] & \sqsubseteq & y := 1\ .
\end{array}
$$

Figure 3.5 Example of *sequential composition* 3.3

3.6 Assignment and composition together

As our programming repertoire increases, we will find many other opportunities for laws that combine several constructions. One such is this special case of *assignment* 1.3 and *sequential composition* 3.3, which is useful when one ‘knows’ that a certain assignment is likely to be appropriate in the final code:

Law 3.5 following assignment For any term E,

$$
\begin{array}{ll}
& w, x\colon [pre \;,\; post] \\
\sqsubseteq & w, x\colon [pre \;,\; post[x\backslash E]]\,; \\
& x := E\ .
\end{array}
$$

□

What is left after applying Law 3.5 is a specification for the first half of the composition, which must be further developed; the assignment in the second half is code already.

Note that Law 3.5 allows *any* assignment in its second half provided the changed variables lie within the frame of the original specification. The required first half, on the other hand, is calculated by the law. (Ridiculous choices for the assignment in the second half probably lead to infeasible specifications in the first half — but that does not affect the validity of the step.) The laws *sequential composition* 8.4 and *leading assignment* 8.5 (both still to come) allow similar calculations.

3.7 Example: Swapping variables

We illustrate the laws so far by showing, in full, the development of the program which swaps x and y. (Note that the specification below allows the variable t to be changed as well.)

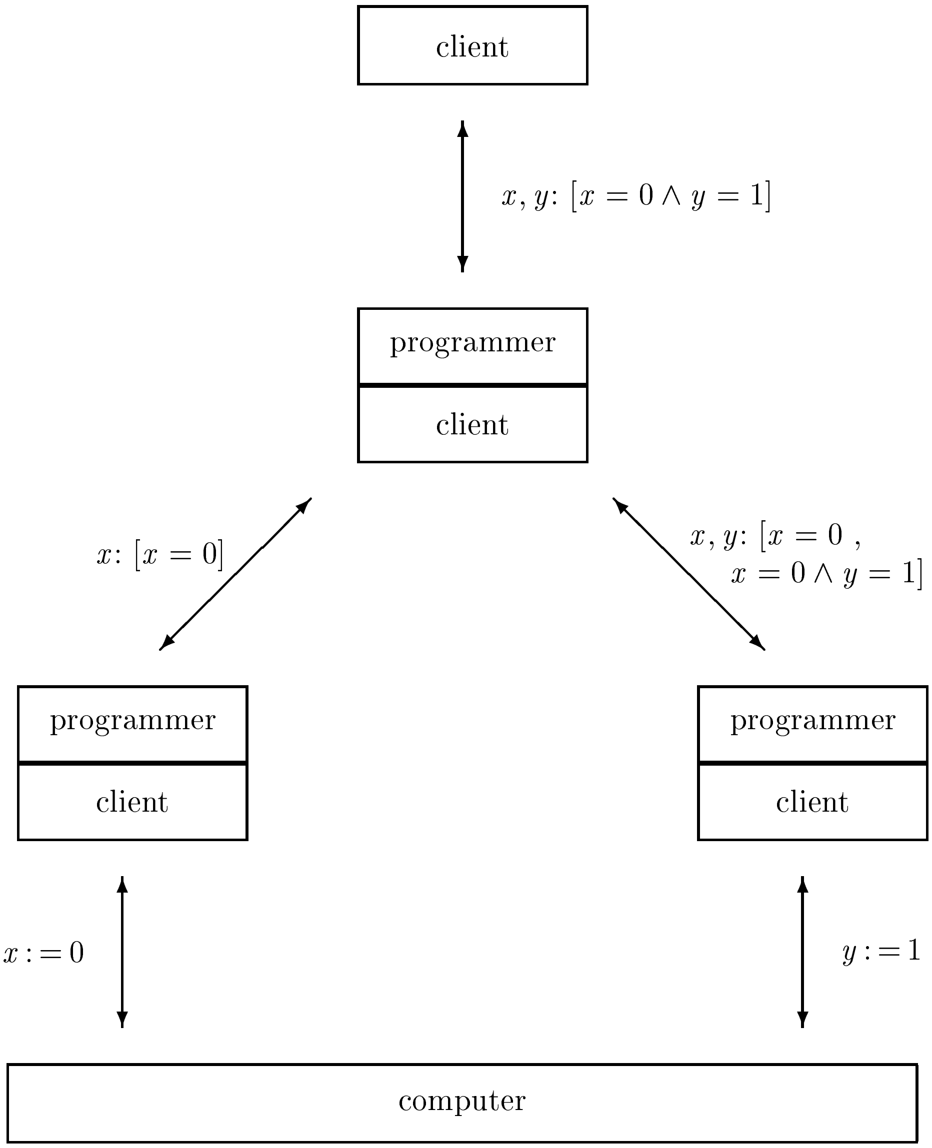

Figure 3.6 A programming hierarchy: recall Figure 1.1.

If the symbol $\triangleleft$ marks part of a program, then that part alone is next refined, with the rest of the program assumed to be carried forward around it. Decorations in quotes are hints, referring to the law(s) that justify the refinement step:

$$
\begin{array}{ll}
& x, y, t\colon [x = X \wedge y = Y\ ,\ x = Y \wedge y = X] \\
\sqsubseteq & \text{“}\textit{following assignment}\ 3.5\text{”} \\
& x, y, t\colon [x = X \wedge y = Y\ ,\ t = Y \wedge y = X]\,; \qquad \triangleleft \\
& x := t
\end{array}
$$

$$\begin{array}{ll} \sqsubseteq & \text{“}\textit{following assignment } 3.5\text{”} \\ & x, y, t\colon [x = X \wedge y = Y \ , \ t = Y \wedge x = X]\,; \qquad \triangleleft \\ & y := x \\ \sqsubseteq & \text{“}\textit{assignment } 1.3\text{”} \\ & t := y\ . \end{array}$$

Overall, the above development is summarised

$$\begin{array}{ll} & x, y, t\colon [x = X \wedge y = Y \ , \ x = Y \wedge y = X] \\ \sqsubseteq & t := y;\ \ y := x;\ \ x := t\ . \end{array}$$

It is the standard swap via a temporary variable.

3.8 Exercises

Ex. 3.1 Use *assignment* 1.3 to show that each of the assignments of Figure 3.1 refines the specification

$$x\colon [x = 0 \wedge y = 1 \ , \ x = 1 \wedge y = 1]\,.$$

Do the same for Figure 3.2, using the specification

$$x, y\colon [x = 0 \wedge y = 1 \ , \ x = 1 \wedge y = 0]\,.$$

Ex. 3.2 Prove the refinements of Figure 3.3.

Ex. 3.3 ♡ Fill in the details of this refinement, using *sequential composition* 3.3:

$$x\colon \left[x = X \ , \ x = X^4\right] \quad \sqsubseteq \quad x := x^2;\ \ x := x^2\ .$$

Ex. 3.4 Use *assignment* 1.3 to derive a law specifically for the assignment $w := w$. Comment on its similarity to *skip command* 3.2.

Ex. 3.5 Refine the following specification to a sequential composition of two assignments, neither of which uses the operation ‘raise to the fourth power’:

$$x, y\colon \left[x = z^2 \wedge y = z^4\right]\ .$$

Ex. 3.6 ♡ Redo the example of Section 3.7 without using *following assignment* 3.5: use *sequential composition* 3.3 directly.

Ex. 3.7 ♡ A *leading* assignment law for multiple assignments is the following:

Law 3.6 leading assignment For disjoint w and x,

$$w, x := E, F[w \backslash E] \quad = \quad w := E;\ x := F \ .$$

□

Note that as a special case we have

$$w, x := E, F \quad = \quad w := E;\ x := F \ ,$$

provided F contains no w.

Use Laws 3.1 and 3.6 to show that

$$x, y, t := y, x, ? \quad \sqsubseteq \quad t := y;\ y := x;\ x := t \ .$$

Ex. 3.8 ♡ Use *sequential composition* 3.3 to prove *following assignment* 3.5.