Chapter 11

Procedures and parameters

For any programming problem other than the very smallest, the size of the development will in itself require careful management if we are not to find that a rigorous approach is more trouble than it is worth.

One of the most basic ways of controlling size and structure in programming is the use of procedures: a procedure is defined in just one place, but used in many. For us, with our broader perspective allowing abstract programs as well as code, we will find procedures convenient for avoiding repeated developments: a specification whose body is a specification may be refined in just one place, but used in many.

There are many advantages of such economy. One is that the structure of the program is more clearly revealed, since sections of identical purpose are clearly labelled as such. Another advantage is that an implementation can save space (perhaps at the expense of some time) by keeping only one copy of the machine code.

But the most significant advantage for us is the one mentioned above. Without procedures, the refinement of a repeated specification must be carried out at its every occurrence; that at the very least involves a lot of copying. With procedures, refinement of the declaration alone gives 'automatic' refinement of all its calls.

The basic principle is that if a text is repeated many times in a program, the program may be improved by *naming* the text and then using the name many times instead. The association of a name with program text is *procedure declaration.* The use of the name to stand for the text is *procedure call.*

If a text is repeated not identically, but with systematic variation — say a renaming of its variables — then there still may be possibilities for reuse of a single declaration. We can use *substitution*, which allows a 'reference copy' of a program text to be adapted to slightly differing uses. When substitution is applied to procedures, we have what are usually called *parametrized procedures.*

In this chapter we look first at procedures, then substitution, and finally the two together: parametrized procedures.

11.1 Procedures without parameters

11.1.1 Declaring procedures

The basic declaration is extremely simple: a procedure is *declared* by associating a *name* with a program fragment, and the fragment is then called the *body* of the procedure. For declarations we use the syntax

$$\textbf{procedure}\ N \ \ \widehat{=}\ \ \mathit{prog}\ ,$$

where N is the name of the procedure, and *prog* is its body. Like the declaration of ordinary variables (for example **var** declarations), procedure declarations are made within a local block that indicates their scope. There are no restrictions on what may be declared a procedure, or where.

11.1.2 Why one bothers

Suppose we are given three integers p, q, r, and are to sort them into increasing order. We might try

$$\begin{array}{ll} & p, q, r\colon \left[\begin{array}{c} p \leq q \leq r \\ \llbracket p, q, r \rrbracket = \llbracket p_0, q_0, r_0 \rrbracket \end{array}\right] \\ \sqsubseteq & p, q := p \sqcap q, p \sqcup q; \\ & q, r := q \sqcap r, q \sqcup r; \\ & p, q := p \sqcap q, p \sqcup q\ , \end{array}$$

which is effectively Insertion Sort specialised to three elements. (Note the $\llbracket p, q, r \rrbracket = \llbracket p_0, q_0, r_0 \rrbracket$ corresponding to our more general $A = \mathsf{bag}\ as$ of Chapter 10.)

Because the first and third commands are the same, we introduce a procedure (in the spirit of this chapter) and continue the development:

$$\begin{array}{ll} \sqsubseteq & \textbf{procedure}\ \mathit{Sort}\ \widehat{=}\ p, q := p \sqcap q, p \sqcup q \bullet \qquad \text{(i)} \\ & \\ & \mathit{Sort}; \\ & q, r := q \sqcap r, q \sqcup r; \\ & \mathit{Sort}\ . \end{array}$$

As for **var** declarations, we can write the declaration as a decoration, and add the block brackets, later, when we collect the code.

Having defined the procedure, we can leave until later the development of its body. When finally that moment comes, and assuming for now that $\sqcup$ and $\sqcap$ are not code, we might conclude with

$$\text{(i)}\ \sqsubseteq\ \textbf{if}\ p \geq q\ \textbf{then}\ p, q := q, p\ \textbf{fi}\ .$$

$$
\begin{array}{l}
|[\ \ \textbf{procedure}\ Sort \\
\quad \hat{=}\ \ \textbf{if}\ p \geq q\ \textbf{then}\ p, q := q, p\ \textbf{fi} \bullet \\
\\
\quad Sort; \\
\quad q, r := q \sqcap r, q \sqcup r; \\
\quad Sort \\
]|
\end{array}
$$

Figure 11.1 Procedure call

$$
\begin{array}{l}
\textbf{if}\ p \geq q\ \textbf{then}\ p, q := q, p\ \textbf{fi}; \\
q, r := q \sqcap r, q \sqcup r; \\
\textbf{if}\ p \geq q\ \textbf{then}\ p, q := q, p\ \textbf{fi}
\end{array}
$$

Figure 11.2 Procedure removed from Figure 11.1.

The code resulting from those refinements overall is shown in Figure 11.1, where the middle assignment $q, r := \cdots$ stands tantalisingly untouched. We shall return to it.

All the above can be undone if we replace every occurrence of a procedure name by its text.[1] Applied to Figure 11.1, the result is Figure 11.2, where it is clear how the refinement of a procedure body (in one place) has in effect refined its (two) calls.

11.1.3 Variable capture

When declaring a procedure, or removing it, it is essential that the movement of its text — between point of call and point of declaration — does not move variables into or out of the blocks in which they are declared. The following will not do, for example:

$$
\begin{array}{ll}
 & |[\ \textbf{var}\ p \bullet p := p + 1\ ]| \\
\sqsubseteq ? & \textbf{procedure}\ Inc\ \hat{=}\ p := p + 1 \bullet \\
 & |[\ \textbf{var}\ p \bullet Inc\ ]|\ .
\end{array}
$$

[1]That technique, known as the *Copy Rule*, comes from the definition of the programming language *ALGOL-60*.

Moving $p := p + 1$ out of the block $|[$ **var** $p \cdots$ in which p is declared (out of 'scope') results in code that increments a different p altogether: the reference to p originally is within the local block, but in the 'refined' program, the reference to p is outside the local block. We see later that parametrization can avoid that kind of difficulty.

11.2 Substitution by value

Now we return to the problem of the second command in Figure 11.1, which looks so much like the other two. What we need is a way of altering *Sort* systematically, so that it affects q and r rather than p and q. Such alterations can be made by *substitution*, and we will examine three kinds: by value, by result, and by value-result.

Underlying all three substitutions is the notion of *simple* substitution of one variable for another. Section 11.8 explains why, simple though it is, it cannot be used directly in programming.

Our first kind of substitution, by value, is used in situations where the code we have involves some variable f, say, but we would like to replace the f by an expression A. For example, we may have a command

$$r := \sqrt{f} \tag{11.1}$$

that assigns to r the square root of f — but really we would like the square root of 2 instead. Substituting 2 by *value* for f in (11.1) will get us what we are after, because (as we shall soon see)

$$(r := \sqrt{f})[\mathbf{value}\ f : \mathbb{R}\backslash 2] \quad = \quad r := \sqrt{2}\ .$$

(Our choice of $\mathbb{R}$ for the type of the parameter is fairly arbitrary here: it has 2 as an element, and the operation $\sqrt{}$ is defined for it.)

In general, we speak of replacing the *formal* parameter f (a variable) by the *actual* parameter A — and *provided A itself contains no f*, we can define *prog*[**value** f : $T\backslash A$] to be

$$\begin{array}{l} |[\ \ \mathbf{var}\ f : T\bullet \\ \quad f := A; \\ \quad \mathit{prog} \\ ]|\ . \end{array}$$

The type T of f is used, just as any other type declaration would be, for subsequent refinement steps involving f within the procedure body.

If A does contain f (which is very common), then we must go to just a little more trouble, to avoid capturing the f in A with the **var** f declaration. We use a simple substitution to change the f in *prog* to some new local variable l, say. In that case, we would have

$$\begin{array}{l} |[\ \ \mathbf{var}\ l : T \bullet \\ \quad l := A; \\ \quad prog[f \backslash l] \\ ]|\ . \end{array}$$

The $[f \backslash l]$ is a simple substitution, replacing *all* occurrences of f by the fresh local variable l. ('All' means occurrences on the left of assignments as well as on the right, and 0-subscripted (initial) variables as well.)

If for example *prog* is $s := n^2$, but we need the square of $n+1$ instead, we might use

$$(s := n^2)[\mathbf{value}\ n : \mathbb{N} \backslash n + 1]$$

which, by the above, is

$$\begin{array}{l} |[\ \ \mathbf{var}\ l : \mathbb{N} \bullet \\ \quad l := n + 1; \\ \quad s := l^2 \\ ]|\ , \end{array}$$

and that, in turn, equals $s := (n+1)^2$.

Note that both 'obvious' alternatives to using the fresh variable would be wrong; the first,

$$\begin{array}{l} |[\ \ \mathbf{var}\ n : \mathbb{N} \bullet \\ \quad n := n + 1; \\ \quad s := n^2 \\ ]|\ , \end{array}$$

improperly captures n so that the resulting code finds the square of one more than an uninitialised local variable. The other alternative, just

$$\begin{array}{l} n := n + 1; \\ s := n^2\ , \end{array}$$

indeed finds the square of $n+1$, but changes n in the process — something that the desired $s := (n+1)^2$ does not do.

11.3 Procedures with parameters

Our principal use of substitution will be when calling procedures whose bodies are not quite what we want. Suppose we have two procedures, for example, one for finding square roots, and one for finding squares:

$$\begin{array}{lll} \mathbf{procedure}\ \ Sqrt & \widehat{=} & r := \sqrt{f} \\ \mathbf{procedure}\ \ Sqr & \widehat{=} & s := n^2\ . \end{array}$$

Then, as we have seen, $Sqrt[\mathbf{value}\ f : \mathbb{R} \backslash 2]$ assigns to r the square root of 2, and $Sqr[\mathbf{value}\ n : \mathbb{N} \backslash n+1]$ assigns to s the square of $n+1$. Since we are likely to use those procedures more than once (why else declare them?), and we do not want to write $[\mathbf{value}\ f : \mathbb{R} \backslash \cdots$ or $[\mathbf{value}\ s : \mathbb{N} \backslash \cdots$ each time, we use the following syntax instead. The 'substituted for' part of the substitution is written with the procedure body, as in

$$\begin{array}{l}
\mathbf{procedure}\ Sqrt\ (\mathbf{value}\ f : \mathbb{R})\ \ \widehat{=}\ \ r := \sqrt{f} \\
\vdots \\
Sqrt\ (2)\ ,
\end{array}$$

and

$$\begin{array}{l}
\mathbf{procedure}\ Sqr\ (\mathbf{value}\ n : \mathbb{N})\ \ \widehat{=}\ \ s := n^2 \\
\vdots \\
Sqr\ (n+1)\ .
\end{array}$$

(It is no coincidence that these now look like conventional procedure calls.)

To find out now what exactly is meant above by $Sqr\ (n+1)$, for example, we could reason

$$\begin{array}{ll}
 & Sqr\ (n+1) \\
= & \text{“parameter declaration with procedure”} \\
 & Sqr[\mathbf{value}\ n : \mathbb{N} \backslash n+1] \\
= & \text{“body of } Sqr\text{”} \\
 & (s := n^2)[\mathbf{value}\ n : \mathbb{N} \backslash n+1] \\
= & \text{“definition of value substitution”} \\
 & |[\ \ \mathbf{var}\ l : \mathbb{N} \bullet \\
 & \quad\ \ l := n+1; \\
 & \quad\ \ s := l^2 \\
 &]| \\
= & s := (n+1)^2\ .
\end{array}$$

That is the reasoning which forms the bedrock of our treatment of procedures and parameters — but in practice we need not always involve ourselves in quite so much tortuous detail.

11.4 Setting out refinements to procedure calls

Our examples so far have shown the effect of a substitution on a given procedure body. In practice, we will need to go in precisely the opposite direction, setting out our refinements in this fashion:

$$s := (n+1)^2 \qquad \text{(i)}$$

$$
\begin{array}{ll}
\sqsubseteq & |[\ \ \mathbf{var}\ l : \mathbb{N} \cdot \\
& \quad l := n + 1; \\
& \quad s := l^2 \\
&]| \\
\sqsubseteq & \text{“assuming declaration } Sqr\ (\mathbf{value}\ n : \mathbb{N})\ \widehat{=}\ s := n^2\text{”} \\
& Sqr\ (n+1)\ . \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(ii)}
\end{array}
$$

Command (i) is where we start; and given a procedure declaration as shown, we finish with $Sqr\ (n+1)$. The intermediate step is precisely the form of a substitution by value of $n+1$ for n, and it is that whole block that is refined to $Sqr\ (n+1)$.

To help with setting refinements out that way, we can make the move from (i) to (ii) in a single step, thus avoiding having to deal with the intermediate local block explicitly. We have for example

Law 11.1 value assignment Given a procedure declaration that refines

$$\mathbf{procedure}\ Proc\ (\mathbf{value}\ f : T)\ \widehat{=}\ w, f := E, ?\ ,$$

we have the following refinement:

$$w := E[f \backslash A] \quad \sqsubseteq \quad Proc\ (A)\ .$$

The actual parameter A may be an expression, and it should have type T. (If it does not, the refinement remains valid but subsequent type checking will fail.) As usual, variables w and f must be disjoint.
□

That the procedure body in Law 11.1 may alter f may seem odd for a value parameter; but such alterations have no effect outside the procedure, since f is not in the frame of the left-hand side of the law.

Independently of procedures, Law 11.1 could have been written

$$(w, f := E, ?)[\mathbf{value}\ f : T \backslash A] \quad = \quad w := E[f \backslash A]\ ,$$

but as we shall usually be using parameters and procedures together, we give the combined form in the law.

In speaking above about ‘a procedure body that refines’, we mean one whose *body* is a refinement of $w, f := E, ?$. One such refinement is $w, f := E, ?$ itself; another is $w := E$ (forgoing the opportunity of changing f); yet another is $w, f := E, F$ for any expression F whatever.

Our squaring example above would now be set out (assuming the same procedure declaration)

$$
\begin{array}{ll}
& s := (n+1)^2 \\
= & s := (n^2)[n \backslash n+1] \\
\sqsubseteq & \text{“\textit{value assignment} 11.1”} \\
& Sqr\ (n+1)\ .
\end{array}
$$

$$
\begin{array}{rcl}
\textbf{skip} & \sqsubseteq & (p := q)[\textbf{value}\ q \backslash p] \\
p := r & \sqsubseteq & (p := q)[\textbf{value}\ q \backslash r] \\
p\colon [p = p_0] & \sqsubseteq & p\colon [p = q]\,[\textbf{value}\ q \backslash p] \\
\textbf{skip} & \sqsubseteq & (\textbf{choose}\ p)[\textbf{value}\ p \backslash p]
\end{array}
$$

Figure 11.3 Value substitutions

Note that we have used the fact that $s, n := n^2, ? \ \sqsubseteq \ s := n^2$.

A similar package exists for specifications: it is given by

Law 11.2 value specification Given a procedure declaration that refines

$$\textbf{procedure}\ \mathit{Proc}\ (\textbf{value}\ f : T)\ \mathrel{\widehat{=}}\ w, f\colon [\mathit{pre}\ ,\ \mathit{post}]\ ,$$

with *post* containing no f (but possibly f_0), the following refinement is valid:

$$w\colon [\mathit{pre}[f \backslash A]\ ,\ \mathit{post}[f_0 \backslash A_0]] \quad \sqsubseteq \quad \mathit{Proc}\ (A)\ ,$$

where A_0 is $A[w \backslash w_0]$.

□

As in *value assignment* 11.1, the procedure body can alter f, and here is an example of where that is useful. Suppose we have a procedure declaration

$$\textbf{procedure}\ \mathit{Fact}\ (\textbf{value}\ n : \mathbb{N})\ \mathrel{\widehat{=}}\ p, n\colon [0 \le n\ ,\ p = n_0!]\ ,$$

which we might have used to make the following refinement steps:

$$
\begin{array}{ll}
 & \{0 < q\}\ p := (q \ominus 1)! \\
= & p\colon [0 < q\ ,\ p = (q \ominus 1)!] \\
\sqsubseteq & p\colon [(0 \le n)[n \backslash (q \ominus 1)]\ ,\ (p = n_0!)[n_0 \backslash (q_0 \ominus 1)]] \\
\sqsubseteq & \text{“declaration of } \mathit{Fact}\text{”} \\
 & \mathit{Fact}\ (q \ominus 1)\ .
\end{array}
$$

Later, we could return to *Fact* itself and, since n is in the frame, develop the procedure body as follows:

$$
\begin{array}{ll}
 & p, n\colon [0 \le n\ ,\ p = n_0!] \\
\sqsubseteq & p := 1; \\
 & \textbf{do}\ n \ne 0 \rightarrow p, n := p \times n, n \ominus 1\ \textbf{od}\ .
\end{array}
$$

In Figure 11.3 are some other examples of value substitutions.

11.4.1 Substitution by result

Useful though it is, our substitution by value cannot by itself take us from Procedure *Sort* to the remaining assignment $q, r := q \sqcap r, q \sqcup r$. The procedure assigns to p and q, but we want an assignment to q and r — and value substitution cannot change that.

Substitution by result is complementary to substitution by value: it takes a value 'out of' a procedure rather than 'into' it. Thus, for example, if we want to adapt our assignment $r := \sqrt{f}$ to assign to s rather than to r, we could use $(r := \sqrt{f})[\mathbf{result}\ r : \mathbb{R} \backslash s]$. That follows from the general form of substitution by result, given by defining $prog[\mathbf{result} f : T \backslash a]$ to be

$$
\begin{array}{ll}
|[& \mathbf{var}\ l : T \bullet \\
 & prog[f \backslash l]; \\
 & a := l \\
]| \ .
\end{array}
$$

The actual parameter this time must be a simple variable, since it is assigned to. (That is why we write a, rather than A as before.) For $r := \sqrt{f}$, the local block would be

$$
\begin{array}{ll}
|[& \mathbf{var}\ l : \mathbb{R} \bullet \\
 & l := \sqrt{f}; \\
 & s := l \\
]| \ ,
\end{array}
$$

which equals $s := \sqrt{f}$, just as we hoped.

As with substitution by value, however, we can avoid setting out the intermediate local blocks (if we want to). The law for assignments is

Law 11.3 <u>result assignment</u> Given a procedure declaration that refines

$$\mathbf{procedure}\ Proc\ (\mathbf{result}\ f : T) \ \widehat{=}\ w, f := E, F \ ,$$

with f not occurring in E or in F, we have the following refinement:

$$w, a := E, F \quad \sqsubseteq \quad Proc\ (a) \ .$$

Variables a and f need not be different from each other, but w must be disjoint from both.

□

The reason that f cannot appear in the expressions E, F is that they would then depend on the initial value of f; and glancing at the local-block form of substitution by result shows that the procedure body can have no access to those initial values.

For procedures whose bodies are specifications, we have this law:[2]

[2]See also Ex. 11.8.

$$\begin{array}{rcl} \textbf{skip} & \sqsubseteq & (p := q)[\textbf{result}\ p\backslash q] \\ r, q := q, r & \sqsubseteq & (p, q := q, r)[\textbf{result}\ p\backslash r] \\ q\colon [q \neq 0] & \sqsubseteq & p\colon [p \neq 0]\,[\textbf{result}\ p\backslash q] \\ q\colon [q = q_0] & \sqsubseteq & p\colon [p = q_0]\,[\textbf{result}\ p\backslash q] \end{array}$$

Figure 11.4 Result substitutions

Law 11.4 result specification Given a procedure declaration that refines

$$\textbf{procedure}\ Proc\ (\textbf{result}\ f : T)\ \widehat{=}\ w, f\colon [pre\ ,\ post[a\backslash f]]\ ,$$

with f not occurring in *pre*, and neither f nor f_0 occurring in *post*, we have the following refinement:

$$w, a\colon [pre\ ,\ post] \quad \sqsubseteq \quad Proc\ (a)\ .$$

Again, variables a and f need not be different from each other, but w must be disjoint from both.

□

With Law 11.3, the square root example would be done in just one step:

$$\begin{array}{ll} & s := \sqrt{f} \\ \sqsubseteq & \text{“}\textit{result assignment}\ 11.3\text{”} \\ & Sqrt\ (s)\ . \end{array}$$

Figure 11.4 gives further examples of result substitutions.

Note that in all substitutions, if the formal parameter f is actually a list then it cannot contain repeated variables. That is because the same restriction applies to the simple substitution $[f\backslash a]$ from which the others are constructed: $[x, x\backslash 0, 1]$, for example, is meaningless. Since in result substitution the assignment $a := f$ appears as well, also the actual parameters must be distinct from each other; but that restriction does not apply to value substitution.

11.5 Multiple substitutions

An obvious possibility with the square root procedure is now to parametrize both its ‘input’ and its ‘output’ — then it would be much more useful, and we could for example realise $s := \sqrt{2}$. We do that by putting our two kinds of substitution together: in this case, the effect of $[\textbf{value}\ f : \mathbb{R};\ \ \textbf{result}\ r : \mathbb{R}\backslash 2, s]$ on $r := \sqrt{f}$ would be

$$
\begin{array}{l}
|[\ \ \mathbf{var}\ l, m : \mathbb{R} \bullet \\
\quad l := 2; \\
\quad m := \sqrt{l}; \\
\quad s := m \\
]|\ ,
\end{array}
$$

where the simple substitution used was $[f, r \backslash l, m]$. That means our general square-root-finding procedure might be declared

$$\mathbf{procedure}\ Sqrt\ (\mathbf{value}\ f : \mathbb{R};\ \ \mathbf{result}\ r : \mathbb{R})\ \ \widehat{=}\ \ r := \sqrt{f}\ ,$$

and we would write just $Sqrt\ (2, s)$ to invoke it as above.

Although there are laws (of course!) for dealing directly with multiple substitutions, they tend to be more trouble than they are worth: they simply pile up the various simple substitutions in the separate laws. For multiple substitutions, therefore, we will stick with the local-block form of development.

11.6 Substitution by value-result

We still have our original $q, r := q \sqcap r, q \sqcup r$ to deal with, and at this stage it is tempting to suggest declaring

$$\mathbf{procedure}\ Sort\ (\mathbf{value}\ p, q : T; \mathbf{result}\ p, q : T)\ ,$$

so that $Sort\ (q, r, q, r)$ would mean

$$
\begin{array}{l}
|[\ \ \mathbf{var}\ l, m : T \bullet \\
\quad l, m := p, q; \\
\quad l, m := l \sqcap m, l \sqcup m; \\
\quad p, q := l, m \\
]|\ .
\end{array}
$$

In doing so, we would be brushing aside the feeling that somehow we should be declaring four local variables, one for each formal parameter, rather than just two.

In fact for situations like the above, we use our third kind of substitution, value-result. The local block above would then result from declaring

$$\mathbf{procedure}\ Sort\ (\mathbf{value\ result}\ p, q : T)\ ,$$

and using $Sort\ (q, r)$. In general, $prog[\mathbf{value\ result}\ f : T \backslash a]$ means

$$
\begin{array}{l}
|[\ \ \mathbf{var}\ l : T \bullet \\
\quad l := a; \\
\quad prog[f \backslash l]; \\
\quad a := l \\
]|\ .
\end{array}
$$

$$
\begin{array}{rcl}
p := r & \sqsubseteq & (p := q)[\textbf{value result } q \backslash r] \\
r := q & \sqsubseteq & (p := q)[\textbf{value result } p \backslash r] \\
q := p & \sqsubseteq & (p := q)[\textbf{value result } p, q \backslash q, p] \\
p\colon [r \geq 0 \ , \ p = r_0] & \sqsubseteq & p\colon [q \geq 0 \ , \ p = q_0]\,[\textbf{value result } q \backslash r] \\
r\colon [q \geq 0 \ , \ r = q_0] & \sqsubseteq & p\colon [q \geq 0 \ , \ p = q_0]\,[\textbf{value result } p \backslash r] \\
r\colon [s \geq 0 \ , \ r = s_0] & \sqsubseteq & p\colon [q \geq 0 \ , \ p = q_0]\,[\textbf{value result } p, q \backslash r, s]
\end{array}
$$

Figure 11.5 Value-result substitutions

It is indeed a combination of value and result.

As usual, there are laws for the special cases in which the procedure body is an assignment or a specification and, as earlier, we assume that the variables w are distinct from f and a. Here is a law for assignments:

Law 11.5 value-result assignment Given a procedure declaration that refines

$$\textbf{procedure } \mathit{Proc}\ (\textbf{value result } f : T) \ \widehat{=}\ w, f := E, F \ ,$$

we have the following refinement:

$$w, a := E[f \backslash a], F[f \backslash a] \quad \sqsubseteq \quad \mathit{Proc}\ (a) \ .$$

□

And for specification-bodied procedures we have

Law 11.6 value-result specification Given a procedure declaration that refines

$$\textbf{procedure } \mathit{Proc}\ (\textbf{value result } f : T) \ \widehat{=}\ w, f\colon [\mathit{pre} \ , \ \mathit{post}[a \backslash f]] \ ,$$

with *post* not containing f, we have the following refinement:

$$w, a\colon [\mathit{pre}[f \backslash a] \ , \ \mathit{post}[f_0 \backslash a_0]] \quad \sqsubseteq \quad \mathit{Proc}\ (a) \ .$$

□

Figure 11.5 gives examples of value-result substitutions.

11.7 Syntactic issues

We follow common practice and use parentheses $(\cdots)$ for procedure parametrization, reserving brackets $[\cdots]$ for substitution. Formal parameters (that is, in the declaration, or on the left of the '\') will be separated by ';' except that we allow for example $(\cdots \textbf{value } x; \ \ \textbf{value } y \cdots)$ to be written $(\cdots \textbf{value } x, y \cdots)$, and so

$$
\begin{array}{l}
|[\ \ \textbf{procedure}\ Sort\ (\textbf{value result}\ x, y : T) \\
\quad \widehat{=}\ \textbf{if}\ x \geq y\ \textbf{then}\ x, y := y, x\ \textbf{fi}\bullet \\
\\
\quad Sort\ (p, q); \\
\quad Sort\ (q, r); \\
\quad Sort\ (p, q) \\
]|
\end{array}
$$

Figure 11.6 Parametrized procedure calls

on. Actual parameters (at the point of call, or on the right of the '\') are as before separated by ','. The correspondence between actual and formal parameter is therefore by position.

The formal parameters, with their types, act as local variables for the procedure body. Those types should be chosen so that an assignment taking actual to formal parameters (for value and value-result), or taking formal to actual parameters (for result and value-result), would be well-typed.

Returning finally to our original example, we have Figure 11.6, in which we have changed the name of the formal parameters. (Such name changes make no difference as long as variable capture does not result.) Note that all three calls must be parametrized.

11.8 Substitution by reference

The most common substitution techniques provided in programming languages are call by value and call by *reference*.

For variables, substitution by reference is identical to simple substitution, and it is dangerous.

Consider the following:

$$
\begin{array}{ll}
 & z := 0 \\
\sqsubseteq & z := 1;\ \ z := 0 \\
\sqsubseteq & (y := 1;\ \ x := 0)[\textbf{reference}\ x, y \backslash z, z] \\
\sqsubseteq & \textbf{procedure}\ Wrong\ (\textbf{reference}\ x, y)\ \ \widehat{=}\ \ y := 1;\ \ x := 0\bullet \\
 & Wrong\ (z, z)\ .
\end{array}
$$

(We have omitted types.)

Now if we refine the body of *Wrong* as follows, the resulting program is equivalent to $z := 1$ (see Exercise 11.5):

$$Wrong \quad \sqsubseteq \quad x := 0;\ \ y := 1\ .$$

Our difficulty is due to *aliasing*: the substitutions in each case have identified two variables that were distinct. This can occur explicitly — as in [**reference** $x, y \backslash z, z$] above — or implicitly as in the substitution $(a := f!)$[**reference** $f \backslash a$].

Aliasing is permitted with any of the substitutions of Section 11.2 however, and that is why we use them: there is no need to check. The problem is that substitution by reference is often more efficient, and certainly is more common.[3] Fortunately, we have the following:

> When aliasing is *absent*, substitution by reference is identical to substitution by value-result.

Thus our techniques are available in most practical cases.

11.9 Exercises

Ex. 11.1 ♡ Simplify the following:

1. $(a := f + 1)$[**value** $f \backslash a$]
2. $(f := a + 1)$[**result** $f \backslash a$]
3. $(f := a + 1)$[**value result** $f \backslash a$]
4. $(f := f + 1)$[**value result** $f \backslash a$]

Ex. 11.2 ♡ Supply the missing substitution(s):

1. $n := (n+1)! \sqsubseteq (f := a!)$[**value** ?; **result** ?\?, ?]
2. $a :> a \sqsubseteq (a :> b)$[**value** ?\?]
3. $\quad\; x \colon [x \neq 0\ ,\ x = 1/x_0]$
   $\sqsubseteq\; q \colon [p \neq 0\ ,\ p \times q = 1]$ [**value** ?; **result** ?\?, ?]

Ex. 11.3 ♡ Assuming the procedure declaration

$$\textbf{procedure}\ \mathit{Sqrts}\ (\textbf{value}\ a : \mathbb{R};\ \textbf{result}\ b : \mathbb{R})\ \ \widehat{=}\ \ b \colon [0 \leq a\ ,\ b^2 = a]$$

is in scope, show that the following refinement is valid:

$$x \colon \left[0 \leq x\ ,\ x^2 = x_0\right] \quad \sqsubseteq \quad \mathit{Sqrts}\ (x, x)\ .$$

Ex. 11.4 ♡ We noted that 'procedureless' programs can be recovered by replacing procedure names with their bodies. Consider the following case:

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

[3] In *Pascal* and *Modula-2* it is call by *var*, for example, and it is the default in *FORTRAN*. In *C*, however, call by value is used; but the effect of call by reference is regained by using pointers.

Explain why the equality is dubious. Argue instead that the correct result is $a := 1$. *Hint*: Rename the local variable.

Ex. 11.5 Verify that the program of Section 11.8 is equivalent to $z := 1$ by removing the procedure *Wrong* from it.

Ex. 11.6 ♡ Show that

$$
\begin{array}{ll}
 & p, q, r \colon \left[\begin{array}{c} p \leq q \leq r \\ \llbracket p, q, r \rrbracket = \llbracket p_0, q_0, r_0 \rrbracket \end{array} \right] \\
\sqsubseteq & p, q := p \sqcap q, p \sqcup q; \\
 & q, r := q \sqcap r, q \sqcup r; \\
 & p, q := p \sqcap q, p \sqcup q \ .
\end{array}
$$

Hint: Recall Insertion Sort.

Ex. 11.7 ♡ What is the invariant needed for the refinement in Section 11.4 of *Fact* to the iteration

$$
\begin{array}{l}
p := 1; \\
\textbf{do}\ n \neq 0 \rightarrow p, n := p \times n, n \ominus 1\ \textbf{od}\ ?
\end{array}
$$

Ex. 11.8 ♡ Show that the following is equivalent to Law 11.4:

Given a procedure declaration that refines

$$\textbf{procedure}\ \mathit{Proc}\ (\textbf{result}\ f : T)\ \widehat{=}\ w, f \colon [\mathit{pre}\ ,\ \mathit{post}]\ ,$$

with f not occurring in *pre*, and neither a nor f_0 occurring in *post*, we have the following refinement:

$$w, a \colon [\mathit{pre}\ ,\ \mathit{post}[f \backslash a]] \quad \sqsubseteq \quad \mathit{Proc}\ (a)\ .$$

Again, variables a and f need not be different from each other, but w must be disjoint from both.

Under what circumstances would you prefer one over the other?