Chapter 13

Recursive procedures

Recursive procedures are those that 'call themselves', and the following factorial procedure is perhaps one of the best-known examples:

$$\begin{array}{ll} & \textbf{procedure}\ \mathit{Fact}\ (\textbf{value}\ n : \mathbb{N}) \\ \widehat{=} & \textbf{if}\ n = 0 \to f := 1 \\ & [\!]\ \ n > 0 \to \\ & \qquad \mathit{Fact}\ (n \ominus 1); \\ & \qquad f := f \times n \\ & \textbf{fi}\ . \end{array}$$

If we know that $\mathit{Fact}\ (n \ominus 1)$ assigns $(n \ominus 1)!$ to f, then we can deduce that $\mathit{Fact}\ (n)$ assigns $n!$ to f; and to know that $\mathit{Fact}\ (n \ominus 1)$ assigns $(n \ominus 1)!$ to f we must know...

In this chapter we see how to develop recursive procedures directly from their specifications (without the '...' above). The principal feature, beyond what we know about procedures already, is the use of a variant expression to ensure that the recursion terminates properly. It is essentially the same technique we used in iterations.

13.1 Partial correctness

Here is how we might develop the factorial procedure above, using a straightforward case analysis into an alternation:

$$\begin{array}{lll} & \textbf{procedure}\ \mathit{Fact}\ (\textbf{value}\ n : \mathbb{N})\ \widehat{=}\ f := n! & \\ \sqsubseteq & \text{“the type } \mathbb{N} \text{ of } n \text{ gives cases } n = 0 \text{ and } n > 0\text{”} & \\ & \textbf{if}\ n = 0 \to \{n = 0\}\ f := n! & \text{(i)} \\ & [\!]\ \ n > 0 \to \{n > 0\}\ f := n! & \text{(ii)} \\ & \textbf{fi} & \\ \text{(i)}\ \sqsubseteq & \text{“}0! = 1\text{”} & \\ & f := 1 & \end{array}$$

(ii) $\sqsubseteq$ "*following assignment* 3.5"
$f\colon [n > 0\ ,\ f \times n = n!]\,;$ $\triangleleft$
$f := f \times n$

$\sqsubseteq$ $f\colon [n \ominus 1 \geq 0\ ,\ f = (n_0 \ominus 1)!]$
$\sqsubseteq$? "*value specification* 11.2"
$Fact\ (n \ominus 1)$.

Only the last step is dubious, using the specification of *Fact* within its own development — yet clearly we must do something of that kind for recursive procedures. Before seeing exactly what it is, however, we should find out why we cannot accept developments like the above just as they are. Consider therefore this alternative development of *Fact*:

procedure *Fact* (**value** $n : \mathbb{N}$) $\widehat{=}$ $f := n!$
$\sqsubseteq$?! "recursive reference to *Fact*"
$Fact\ (n)$.

That astonishingly short development gives the code

procedure *Fact* (**value** $n : \mathbb{N}$) $\widehat{=}$ $Fact\ (n)$,

and — what is worse — the same strategy would 'work' for any procedure at all.

The reason it does *not* work is that we have taken no account of termination; both versions of *Fact* have the effect of assigning $n!$ to f if they terminate, but only the first version terminates. The second recurses forever, so that any statement about its effect on termination is vacuously true.

Similarly ignoring termination for iterations would have allowed us to argue that

$$f := n! \quad \sqsubseteq ? \quad \textbf{do}\ \mathsf{true} \rightarrow \textbf{skip od}\ .$$

If that iteration terminated it would indeed assign $n!$ to f — but it never does terminate.

Such 'refinements', ignoring termination, establish what is known as *partial correctness*: that one program refines another 'except for termination'. Unhelpful as it sounds, such partial refinements (as we might call them) are useful ways of having a quick look ahead in a development to see whether it is worth proceeding that way[1] — for if one cannot reach even partial correctness, there is clearly no point in trying the approach at all.

Our normal refinement relation expresses *total* correctness, however, and takes termination into account. For that, we need variants.

[1] Partial correctness was the basis, in [Hoa69], of early arguments for rigour in program development. It has the advantage that its rules are simpler; but then one gets less from them.

13.2 Variants for recursion

The extra we pay for total correctness is that we must declare a named variant expression for the development of any recursive procedure. In the factorial example, it would look like this:

$$\begin{array}{ll} & \textbf{procedure } \mathit{Fact} \ (\textbf{value } n : \mathbb{N}) \ \widehat{=} \ f := n! \\ \sqsubseteq & \textbf{variant } N \textbf{ is } n \bullet \\ & \cdots \end{array}$$

In general, the effect of a variant declaration **variant** V **is** E, for name V and integer-valued expression E, is to require that any recursive reference to the procedure be formulated as if the procedure included $0 \leq E < V$ in its precondition. (References from *outside* the procedure itself, however, need not include the extra precondition.) In the factorial example, therefore, we would have had to refer to

$$\begin{array}{rl} & \{0 \leq n < N\} \ f := n! \\ \text{or equivalently} & f\colon [0 \leq n < N \ , \ f = n!] \end{array}$$

at the point of recursive reference to *Fact*, rather than just $f := n!$ on its own. In doing so we are stipulating (as with iterations) that the variant (n) must have decreased strictly ($\cdots < N$), but not below 0 ($0 \leq \cdots$), before the recursive call is made.

In order to be able to include such an assertion, we allow the introduction of $\{V = E\}$ at the point immediately after the declaration of the variant. (Alternatively, it is just included as a conjunct in the precondition, if that first step is a specification.) The effect on the factorial development would be

$$\begin{array}{lll} \sqsubseteq & \textbf{variant } N \textbf{ is } n \bullet & \\ & \{n = N\} & \\ & \textbf{if } n = 0 \rightarrow \{n = 0\} \ f := n! & \triangleleft \\ & [\!] \ \ n > 0 \rightarrow \{n > 0\} \ f := n! & \text{(i)} \\ & \textbf{fi} \ , & \end{array}$$

and in fact there is no reason it cannot immediately be distributed through the guard of the second branch of the alternation in that same step; we would write (instead of the above) just

$$\begin{array}{lll} \sqsubseteq & \textbf{variant } N \textbf{ is } n \bullet & \\ & \textbf{if } n = 0 \rightarrow \{n = 0\} \ f := n! & \triangleleft \\ & [\!] \ \ n > 0 \rightarrow \{N = n > 0\} \ f := n! & \text{(i)} \\ & \textbf{fi} \ . & \end{array}$$

(Since the first branch does not lead to a recursive call, we have no need of the variant there.) Then the development would continue as before, but this time carrying the variant along:

$$
\begin{array}{lll}
\text{(i)} & \sqsubseteq & \text{“\textit{following assignment} 3.5”} \\
& & f\colon [N = n > 0 \;,\; f \times n = n!]\,; \qquad \triangleleft \\
& & f := f \times n \\
& \sqsubseteq & f\colon [N > n \ominus 1 \geq 0 \;,\; f = (n \ominus 1)!] \\
& \sqsubseteq & \text{“\textit{value specification} 11.2”} \\
& & f\colon [N > n \geq 0 \;,\; f = n!]\,[\mathbf{value}\ n : \mathbb{N} \backslash n \ominus 1] \\
& \sqsubseteq & \text{“note that } N > n \geq 0 \text{ is correctly placed”} \\
& & \mathit{Fact}\ (n \ominus 1)\ .
\end{array}
$$

The second, bogus, development of *Fact* is no longer possible:

$$
\begin{array}{ll}
& \mathbf{procedure}\ \mathit{Fact}\ (\mathbf{value}\ n : \mathbb{N}) \ \widehat{=}\ f := n! \\
\sqsubseteq & \mathbf{variant}\ N\ \mathbf{is}\ n\bullet \\
& \{n = N\}\ f := n! \\
\sqsubseteq & \text{“What now?”}
\end{array}
$$

The assignment cannot be replaced by *Fact* (n), as we did before, because the precondition is not $N > n \geq 0$ as it must be.

13.3 A complete example

Here, as a second example, we develop a recursive searching procedure. Consider a finite set es of type **finset** E, each of whose elements might satisfy a given property p. (The elements could be integers, for example, and the property ‘is even’.) For $e : E$, the formula $\mathsf{p}\, e$ means “p holds for e”.

We develop a recursive procedure which, assuming that some element of s satisfies p, assigns to e one such element; the abstract declaration is

$$
\begin{array}{l}
\mathbf{procedure}\ \mathit{Find}\ (\mathbf{value}\ es : \mathbf{finset}\ E) \\
\widehat{=}\ e\colon [(\exists\, e : es \bullet \mathsf{p}\, e) \;,\; e \in es \wedge \mathsf{p}\, e]\ .
\end{array}
$$

Our strategy is simply to choose an element e from es at random. If it satisfies p then we are finished; if it does not, we invoke *Find* recursively on es with the non-p-satisfying e removed. For conciseness, we define $\mathsf{P}\, es$ to be $(\exists\, e : s \bullet \mathsf{p}\, e)$, a convenient abbreviation for ‘es contains an element satisfying p’. Here is our first step:

$$
\begin{array}{ll}
\sqsubseteq & \mathbf{variant}\ V\ \mathbf{is}\ \#es\bullet \\
& e\colon [es \neq \{\} \;,\; e \in es]\,; \\
& e\colon [V = \#es \wedge e \in es \wedge \mathsf{P}\, es \;,\; e \in es \wedge \mathsf{p}\, e]\ . \qquad \triangleleft
\end{array}
$$

We have moved both $V = \#es$ and $\mathsf{P}\, es$ through to the precondition of the second command, since neither contains variables in the frame of the first command. The precondition $es \neq \{\}$ of the first command itself follows from the original precondition $\mathsf{P}\, es$.

Now we have chosen an element e, and there are two cases: either e satisfies p or it doesn't. We treat them with an alternation:

$$
\begin{array}{lll}
\sqsubseteq & \textbf{if } \mathsf{p}\, e \rightarrow e\colon [e \in es \wedge \mathsf{p}\, e\ ,\ e \in es \wedge \mathsf{p}\, e] & \triangleleft \\
 & [\!]\ \neg\, \mathsf{p}\, e \rightarrow & \\
 & \qquad e\colon [V = \#es \wedge e \in es \wedge \mathsf{P}\, es \wedge \neg\, \mathsf{p}\, e\ ,\ e \in s \wedge \mathsf{p}\, e] & \text{(i)} \\
 & \textbf{fi} & \\
\sqsubseteq & \textbf{skip}\ . &
\end{array}
$$

The first branch has been easily disposed of.

Consider now the precondition of the second branch: if there is *some* element of es satisfying p, but it is not e, then that element must lie in $es - \{e\}$ and so $\mathsf{P}(es - \{e\})$. We can also strengthen the postcondition, allowing us to make the recursive call:

$$
\begin{array}{ll}
\text{(i)}\ \sqsubseteq & e\colon [V > \#(es - \{e\}) \wedge \mathsf{P}(es - \{e\})\ ,\ e \in es - \{e_0\} \wedge \mathsf{p}\, e] \\
\sqsubseteq & \mathit{Find}\ (s - \{e\})\ .
\end{array}
$$

We have not bothered with $0 \leq \#es$, since set sizes are non-negative by definition.

13.4 Epilogue: recursion blocks

What we have seen above is all we need for developing recursive procedures, but there is a special case that deserves a closer look. Sometimes the recursive procedure is declared 'just to be able to recurse', rather than to make the procedure widely available. As matters stand, we have to do that in order to have a name with which to indicate the point of recursion.

We will allow ourselves to write parameterless recursions that are *not* procedures as follows:

$$\textbf{re}\ R \bullet\ \mathit{prog}\ \textbf{er}\ ,$$

for some name R and program fragment *prog*. The effect is precisely as if R had been declared locally, as a procedure, called once, and then forgotten about:

$$
\begin{array}{ll}
|[& \textbf{procedure}\ R \mathrel{\widehat{=}} \mathit{prog} \bullet \\
 & R \\
]|\ . &
\end{array}
$$

If we were to develop a recursion block directly, rather than first developing a 'temporary' procedure as above, we would set it out as in this example:

$$
\begin{array}{lll}
 & f, n := n!, ? & \\
\sqsubseteq & f := 1; & \\
 & f, n := f \times n!, ? & \triangleleft
\end{array}
$$

$$
\begin{array}{ll}
\sqsubseteq & \textbf{re } F \textbf{ variant } N \textbf{ is } n \bullet \\
& \quad \textbf{if } n = 0 \rightarrow \textbf{skip} \\
& \quad [\!]\ \ n > 0 \rightarrow \{N = n > 0\}\ f, n := f \times n!, ? \qquad \triangleleft \\
& \quad \textbf{fi} \\
\sqsubseteq & f, n := f \times n, n \ominus 1; \\
& \{N > n \geq 0\}\ f, n := f \times n!, ? \qquad \triangleleft \\
\sqsubseteq & F\ .
\end{array}
$$

Collecting the code, and collapsing the development, we can see that we have shown

$$
\begin{array}{ll}
& f, n := n!, ? \\
\sqsubseteq & f := 1; \\
& \textbf{re } F \bullet \\
& \quad \textbf{if } n = 0 \rightarrow \textbf{skip} \\
& \quad [\!]\ \ n > 0 \rightarrow f, n := f \times n, n \ominus 1;\ F \\
& \quad \textbf{fi} \\
& \textbf{er}\ ,
\end{array}
$$

and it is interesting to note that, since we began developing the recursion block only *after* the first refinement step, the final code is not in its entirety a recursion: only the second half is.

In fact what we have seen above is an example of a more general phenomenon known as *tail recursion*, and summed up in this equality:

$$
\begin{array}{ll}
& \textbf{do } G \rightarrow \textit{prog} \textbf{ od} \\
= & \textbf{re } D \bullet \\
& \quad \textbf{if } G \textbf{ then } \textit{prog};\ \ D \textbf{ fi} \\
& \textbf{er}
\end{array}
$$

Using the equality (and remembering that n is a natural number, so that we can rewrite the guards $n = 0$ and $n \neq 0$) converts the above factorial program to the more familiar

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

Tail recursion is more general than 'ordinary' iteration, however, as illustrated by this second equality:

$$
\begin{array}{l}
\textbf{loop} \\
\quad \textit{prog}1; \\
\quad \textbf{if } G \textbf{ then exit fi}; \\
\quad \textit{prog}2 \\
\textbf{end}
\end{array}
$$

$$
\begin{array}{l}
= \textbf{re } L \cdot \\
\quad\quad prog1; \\
\quad\quad \textbf{if } \neg G \textbf{ then } prog2;\ \ L \textbf{ fi} \\
\quad \textbf{er} \ .
\end{array}
$$

The **loop** $\cdots$ **exit** $\cdots$ **end** construction iterates its body 'until an **exit** is executed'. The above not only makes that precise, but shows how to develop programs of that form.

13.5 Exercises

Ex. 13.1 ♡ Would *Find* terminate if es were infinite? Where in its development do we use the fact that es is finite?

Ex. 13.2 ♡ For $f, n : \mathbb{N}$, show that

$$f\colon [f = n!] \quad \sqsubseteq \quad Fact'(n, 1) \ ,$$

where

$$
\begin{array}{l}
\textbf{procedure } Fact' \ (\textbf{value } m, k : \mathbb{N}) \\
\mathrel{\widehat=} \ \textbf{if } m = 0 \rightarrow f := k \\
\quad\ \ [\!] \ \ m > 0 \rightarrow Fact' \ (m \ominus 1, m \times k) \\
\quad\ \ \textbf{fi} \ .
\end{array}
$$

Hint: Consider $f := m! \times k$.

Ex. 13.3 *Linear search* Assuming declarations $as : \textbf{seq}_N\, A$ and $i : \mathbb{N};\ a : A$, show that

$$
\begin{array}{ll}
 & i\colon [a \in as \Rightarrow a = as[i]] \\
\sqsubseteq & i := 0; \\
 & \textbf{loop} \\
 & \quad \textbf{if } i \geq N \textbf{ then exit fi}; \\
 & \quad \{i < N\} \ \textbf{if } a = as[i] \textbf{ then exit fi}; \\
 & \quad i := i + 1 \\
 & \textbf{end}
\end{array}
$$

by refining the left-hand side to the tail-recursive code

$$
\begin{array}{l}
i := 0; \\
\textbf{re } L \cdot \\
\quad \textbf{if } i \geq N \rightarrow \textbf{skip} \\
\quad [\!] \ \ i < N \rightarrow \\
\quad\quad\quad \{i < N\} \\
\quad\quad\quad \textbf{if } a = as[i] \rightarrow \textbf{skip} \\
\quad\quad\quad [\!] \ \ a \neq as[i] \rightarrow i := i + 1;\ L \\
\quad\quad\quad \textbf{fi} \\
\quad \textbf{fi} \\
\textbf{er} \ .
\end{array}
$$

Hint: Do not bother about the 'obvious' rewriting of the recursion block to introduce the **loop** and **exit**s as given above; concentrate on reaching the recursion-block itself. Its specification should be $i\colon [a \in as{\downarrow}i_0 \Rightarrow a = as[i]]$.

What is the point of the assumption $\{i < N\}$? Need it be there? *Hint*: See Exercise 9.17 concerning undefined expressions.

Ex. 13.4 Develop an iteration (rather than a recursion) from the specification

$$e, es\colon [\mathsf{P}\ es\ ,\ e \in es_0 \wedge \mathsf{p} e]\ ,$$

similar to *Find* in Section 13.3 except that es may now be changed. *Hint*: Introduce a logical constant S to capture the initial value of the set es, and use the invariant $e \in es \wedge es \subseteq S \wedge \mathsf{P}\ es$.

Ex. 13.5 Show that

$$\begin{array}{ll} & w\colon [inv\ ,\ inv \wedge \neg G] \\ \sqsubseteq & \mathbf{re}\ D\boldsymbol{\cdot} \\ & \quad \mathbf{if}\ G\ \mathbf{then}\ w\colon [G\ ,\ inv\ ,\ 0 \le E < E_0]\,;\ \ D\ \mathbf{fi} \\ & \mathbf{er} \end{array}$$

by carrying on after this first step:

$$\begin{array}{ll} & w\colon [inv\ ,\ inv \wedge \neg G] \\ \sqsubseteq & \mathbf{re}\ D\ \mathbf{variant}\ V\ \mathbf{is}\ E\boldsymbol{\cdot} \\ & w\colon [V = E\ ,\ inv\ ,\ \neg G]\ . \end{array}$$