

Chapter 5

Procedures and Recursion

5.1 Introduction

Procedures¹ offer a well-known way of reusing code. A procedure may be viewed as a named block of code, characterised by its pre- and postconditions. It may be called (or invoked) from other parts of a program. A correctness-by-construction approach can be used to derive the body of the procedure, thus ensuring that it conforms to its stated pre- and postconditions. However, to date we have not explicitly shown how *calling* a procedure can be incorporated into our refinement rules. Section 5.2 provides the relevant refinement rules, while Sect. 5.3 provides a broad strategy for deriving procedures.

The rest of the chapter focusses on recursion. This is a well-known and powerful algorithmic device in terms of which procedures *call themselves* during execution—hence the unsettling aphorism:

To understand recursion, you have to understand recursion!

Although it takes a little time to become familiar with the idea of recursion, once the ice has been broken, recursive algorithms turn out to have a kind of magically succinct quality about them.

Recursion is useful in solving problems through a *Divide and Conquer* strategy: a problem is broken down into smaller pieces that can be easily solved, and then all the smaller solutions are merged together. Recursive procedures are also often used to traverse recursive data structures, such as lists, graphs, trees and lattices.

Many problems are amenable to both recursive and iterative algorithmic solutions. For example, Kaldewaij [25] uses correctness-by-construction techniques to derive iterative solutions for both Quicksort and Mergesort—classically presented

¹Synonyms are *subprocedure*, *subprogram*, *routine*, *subroutine*, *function* and *method*. In this text, we will keep to the terms *procedure* and *function* as they were classically used in languages such as Pascal.

as recursive algorithms. Recursive algorithmic solutions often have advantages over their equivalent iterative versions. Many are more elegantly expressed in their recursive forms, making such algorithms easier for programmers to understand, remember and implement. In the case of template meta-programming in C++, variables cannot be used to support iteration, so recursion for performing repetition [2] is obligatory. On the downside, recursion is more space- and time-expensive when implemented on a real machine.

Nevertheless, on grounds of elegance alone, it is worth investigating how to incorporate recursion into the correctness-by-construction approach. In Sect. 5.4, refinement rules for deriving recursive procedures will be presented, as well as a discussion of total correctness. Section 5.5 shows how variants can be used to ensure termination of recursive programs. The section offers a strategy for constructing such procedures, somewhat analogous to the loop construction strategy of Chap. 2. Section 5.6 then provides a number of examples to demonstrate how the material in Sects. 5.2–5.5 can be used to derive recursive algorithms in practice.

5.2 Procedures

This section draws on the refinement rules provided by Morgan’s refinement calculus. A procedure is a named block of code, characterised by a pre- and postcondition, and possibly by so-called “formal” parameter variables. It can be invoked by giving its name and so-called “actual” parameters to be passed on to and/or retrieved from the procedure. The scope of the parameters can be set in different ways. However, for simplicity, in this discussion we will assume that all variables to be considered *other than actual and formal parameters* have global scope—i.e. they are visible from both the calling environment, and within the procedure.

5.2.1 Parameterless Procedures

The simplest of procedure refinement rule deals with procedures with no parameters. Let $R()$ denote a call to a procedure defined as **proc** $R()$ S **corp**—i.e. the procedure has name R and body S . The following refinement applies.

Rule 10. $Spec(P, S, Q) \sqsubseteq Spec(P, R(), Q)$

In other words, if we wish to derive code that terminates in $State_Q$ if commenced in $State_P$, and we already know that the body, S , of the parameterless procedure, $R()$ complies with this requirement, then we might as well just call $R()$! We don’t even need to know the details of what S does—if, given the precondition, $R()$ terminates and produces the specified postcondition, then it is good enough.

Consider, for example, the following simple procedure which sets a global variable, x , to 1.

proc *SetToOne*() $x := 1$ **corp**

The body of this procedure just happens to comply with the following specification.²

$$Spec(x < 20, x : S, x > 0)$$

It is easy enough to prove that the body of *SetToOne*() indeed complies with this specification (a simple exercise left to the reader). But whether we know that $Spec(x < 20, x : S, x > 0)$ holds as a matter of proof, or as a matter of externally supplied information, Rule 10 asserts that invoking *SetToOne*() from $State_{x < 20}$ will result in a new state in $State_{x > 0}$.

Clearly, $x : [x < 20, x > 0]$ is not the most specific characterisation of what the body of *SetToOne*() accomplishes. Such a specification would require maximal weakening of the precondition and strengthening of the postcondition, namely $x : [\text{true}, x = 1]$. Rule 10 would also apply in this case, so that we could assert that:

$$[\text{true}, x = 1] \sqsubseteq SetToOne()$$

Parameterless procedures such as *SetToOne*() are, however, very limited in their usefulness. For example, *SetToOne*() is restricted to both the global variable x and the constant value 1. The procedure would be far more useful if it could accept, as a parameter, a variable other than x on which to perform its assignment, and/or a value other than 1 to be used in the assignment.

There are three classically-known types of parameters: “pass-by-value”, “pass-by-result” and “pass-by-value-result.”³ Refinement rule are defined to handle each of these types of parameters in procedures.

In the rule explanations that will follow below, a *formal parameter* is the variable declared in the procedure definition whose scope is limited to the procedure. An *actual parameter* refers to the variable or value sent through at the procedure call. In the following code snippet, z is the formal parameter and a is the actual parameter.

```
proc Proc( $z$ ) ... corp
:
:
 $a := p * q + r;$ 
Proc( $a$ )
```

²Recall the three equivalent notational options introduced in Chap.2. There we introduced $Spec(x < 20, x : S, x > 0)$ as just another form of the Hoare triple notation $\{x < 20\} x : S \{x > 0\}$. Morgan’s notation of $x : [x < 20, x > 0]$ suppresses explicit reference to abstract code, S , whereas concrete code is explicitly given and any inference about such code’s pre- and postconditions has to rely on previous refinement steps. We freely alternate between notations in order to highlight various aspects of a specification.

³Some contexts speak of “in”, “out” and “in out” parameters, respectively. The terms “call-by-value” and “call-by-reference” are also encountered, the latter being more or less equivalent to “pass-by-value-result”.

This example takes some notational liberties, in that *Proc* does not specify the type of its formal parameter as pass-by-value, pass-by-result or pass-by-value-result. As will be seen below, this is normally required.

We will now indicate the refinement rules that can be used in the presence of each of these three types of parameters. Throughout we will refer to a procedure $R(z)$, that has the formal parameter z , and which is invoked as $R(a)$, where a is the actual parameter. We will mention in passing various constraints that apply to the contents of the pre- and postconditions when these rules are applied. However, for the sake of simplicity, we will not fully elaborate on them, nor methodically check for compliance with those constraints in subsequent examples, since—in the main—they forbid rather extreme situations which are seldom relevant.

5.2.2 Pass by Value

A pass-by-value parameter initialises the formal parameter's value to that of the actual parameter's value, but changes made to the value of the formal parameter during execution of the procedure do not affect the value of the actual parameter, a . Essentially the parameter is used to pass a *value* to the procedure, and therefore the parameter a is best thought of as an expression (which is evaluated before control is passed to the procedure body) rather than as a variable.⁴

Suppose we are given the procedure specified as **proc** $R(\text{value } z) \ w, z : R_{\text{body}}$ **corp.** This specification emphasises that the frame variables (or perhaps variable lists) w and z may change when R_{body} is executed. The pass-by-value rule to be given below indicates the circumstances under which $R(a)$ may be invoked.

The rule will rely on the notation a_0 to denote the actual parameter, a , after all occurrences of w in a have been substituted by w_0 , i.e. $a_0 \equiv a[w \setminus w_0]$. Now to see why this makes sense, one has to understand that the expression a may include the variable w . For example, a could be the expression $w + 4x$ (x being some other arbitrary variable). Since frame variable w may change during the execution of R , we have an interest in noting its initial value w_0 , as well as the initial value of a , namely $a_0 \equiv w_0 + 4x$. Note that there is no need to be concerned about how x will change during a call to R : we may assume that it does not change at all, since it does not appear as a frame variable in relation to R_{body} .

Using this notation, the pass-by-value refinement rule is as follows.

⁴An expression is a programming construct that returns a *value*. A variable to which a value has been assigned is thus one special form of an expression. Other kinds of expressions are constants, expressions with operators, etc. A function call is also conventionally regarded as an expression.

Rule 11. *Pass by value rule*⁵**Given:** $\text{proc } R(\text{value } z) \{P\} w, z : R_{\text{body}} \{Q\} \text{ corp}$ **Rule:** $\{P[z \setminus a]\} w : S \{Q[z_0 \setminus a_0]\} \sqsubseteq \{P[z \setminus a]\} R(a) \{Q[z_0 \setminus a_0]\}$

Suppose we have a library containing the utility procedure R . To be useful, we would obviously expect that its documentation contains information about the pre- and postconditions, P and Q , under which it operates—i.e. under which $\{P\} w, z : R_{\text{body}} \{Q\}$ is **true**. Of course, the details of R_{body} is assumed to be unknown. However, we do know that R_{body} may only change values of the pass-by-value formal parameter z and the globally known variable w . (Throughout, we assume z and w to be single variables. The discussion trivially generalises to the case where they represent lists of variables instead.) Note that P and/or Q are allowed to refer to variables w and z .

The rule indicates that the a call to $R(z)$, i.e. the concrete code $R(a)$, then conforms to the specification

$$\{P[z \setminus a]\} R(a) \{Q[z_0 \setminus a_0]\}.$$

Moreover, the rule indicates that this specification is a refinement of any abstract specification that has the same pre- and postconditions and allows for the changes in w —i.e. it is a refinement of $\{P[z \setminus a]\} w : S \{Q[z_0 \setminus a_0]\}$.

Note that it would not make any sense to specify in the postcondition, Q , any kind of expectations on the range of *final* values of the formal value parameter, z . Why not? Well, because the scope of z is limited to within R_{body} , and any such information would be of no use to a caller of R . However, it is well within expectation that the designer of R_{body} might need to specify, in postcondition Q , something about how the initially passed value of z , namely z_0 , features in relation to other variables that have been manipulated during the execution of R . For example, the purpose of R_{body} may have been to ensure that $w > z_0$.

Now suppose that you need to change the value of w at a point where you know that predicate P' will hold, and you believe that by using the actual parameter a in the call $R(a)$, your purpose would be achieved—i.e. say Q' will hold. How can you know whether or not the call $R(a)$ complies with your requirements in a given context? Rule 11 will assist you in answering your question.

The information you have at hand is that $\{P\} w, z : R_{\text{body}} \{Q\}$ holds. The rule tells you that by calling $R(a)$, you may be confident that $\{P[z \setminus a]\} R(a) \{Q[z_0 \setminus a_0]\}$ will hold.

If P' and Q' do not coincide exactly with $P[z \setminus a]$ and $Q[z_0 \setminus a_0]$ respectively, you would still not know whether or not the call to $R(a)$ achieves your purpose. To be affirmed of that, you would need to show is that

$$\{P'\} w, z : S \{Q'\} \sqsubseteq \{P[z \setminus a]\} R(a) \{Q[z_0 \setminus a_0]\}$$

⁵The rule is given in Hoare triple notation, because the pre- and postconditions that apply when the call is issued are explicitly shown. For this reason Hoare notation will also be used in subsequent rules. However, the essential equivalence of the two notations should constantly be borne in mind. In Morgan's notation this particular rule is: $w : [P[z \setminus a], Q[z_0 \setminus a_0]] \sqsubseteq R(a)$.

You would have to rely on other refinement rules to show this—typically using rules relating to strengthening postconditions, weakening preconditions, etc. You might even have to rely on the sequence rule where $P[z \setminus a]$ and $Q[z_0 \setminus a_0]$ feature as *mid* predicates en route to attaining Q' from P' .

As an example of the rule in action, consider the following procedure, which merely sets the global variable, w , to its pass-by-value parameter incremented by 1.

proc $R(\text{value } z) \ w := z + 1$ corp

Now suppose that the designer of this procedure had placed it in a library and, for reasons related to the context of its intended use, advertises that $R(\text{value } z)$ conforms to the specification:

$$w, z : [w \geq 0 \wedge z < 100, w > z_0]$$

It is a simple matter to show that the procedure's body indeed conforms to the specification. (Of course, it also conforms to various other specifications, and in practice, it might have been more sensible for the designer to specify a precondition that is as weak as possible, and a postcondition that is as strong as possible, for example $w, z : [\text{true}, w = w_0 + 1 \wedge z = z_0 + 1]$. However, the given specification is used for illustrative reasons.)

Note that, as required for the application of the rule, the postcondition is independent of the formal parameter, z . It indeed reference its initial value z_0 , but this was tolerated by the rule's constraints. As a result we may apply Rule 11, using $(w \geq 0 \wedge z < 100)$ and $(w > z_0)$ as P and Q , respectively.

Let us consider applying the rule under two circumstances. In the first case, suppose we want to use an actual parameter of 5, i.e. we wish to call $R(5)$. To apply the rule we need to determine the precondition under the substitution $[z \setminus 5]$, which becomes

$$w \geq 0 \wedge 5 < 100 \equiv w \geq 0 \wedge \text{true} \equiv w \geq 0.$$

Also needed is the postcondition under the substitution $[z_0 \setminus a_0]$, where

$$a_0 \equiv a[w \setminus w_0] \equiv 5[w \setminus w_0] \equiv 5$$

The postcondition under the substitution $[z_0 \setminus 5]$ then becomes

$$(w > w_0 z_0)[z_0 \setminus 5] \equiv (w > 5)$$

Rule 11 then assures us that:

$$\begin{aligned} & \{(w \geq 0)\} w : S \{(w > 5)\} \\ & \sqsubseteq \{(w \geq 0)\} R(5) \{(w > 5)\} \end{aligned}$$

Thus, Rule 11 assures us that if $w > 0$ and the call $R(5)$ is made, then the call will terminate and $w > 5$ when it does so.

To illustrate the use of the rule when the actual parameter is slightly more complicated, suppose we wished to make the call $R(3w + 1)$. In this case, we need to make the substitution $[z \setminus 3w + 1]$ in the precondition to get: $(w \geq 0) \wedge (3w_0 + 1 < 100)$

Since $a_0 = (3w + 1)[w \setminus w_0] = 3w_0 + 1$, we need the substitution $[z_0 \setminus 3w_0 + 1]$ in the postcondition. This gives $(w > 3w_0 + 1)$. The result would be the following application of Rule 11:

$$\begin{aligned} & \{(w \geq 0) \wedge (3w_0 + 1 < 100)\} w : S \{(w > 3w_0 + 1)\} \\ & \sqsubseteq \{(w \geq 0) \wedge (3w_0 + 1 < 100)\} R(3w_0 + 1) \{(w > 3w_0 + 1)\} \end{aligned}$$

It does no harm at this stage to eliminate the explicit reference to initial values in P , so that, because of Rule 11 we could have confidence in the following specification:

$$\{(0 \leq 3w + 1 < 100)\} R(3w + 1) \{(w > 3w + 1)\}$$

Of course, the developer who wished to ensure that the postcondition holds after the call to $R(3w + 1)$ would have to ensure that the precondition holds before the call. That seems like a lot more trouble than it is worth in the case of this rather contrived example.

In summary, we have seen that if we know something about behaviour of a procedure's body in terms of pre- and postconditions that comply with the rule's restrictions, and if we invoke the procedure with appropriate actual parameters, then Rule 11 informs us of a pre-post specification to which that invocation will conform.

5.2.3 Pass by Result

A “pass-by-result” actual parameter *has to be* a variable—it may not be any other kind of expression. It is assigned the value of its corresponding formal parameter when the procedure terminates. Such parameters pass values “out” of the procedure whereas pass-by-value parameters pass values “into” the procedure.

The pass-by-result rule is given below, where it is now assumed that a in the call $R(a)$ is a variable.

Rule 12. Pass by result rule

Given: $\text{proc } R(\text{result } z) \{P\} w, z : R_{\text{body}} \{Q\} \text{ corp}$

Rule: $\{P\} w, a : S \{Q[z \setminus a]\} \sqsubseteq \{P\} R(a) \{Q[z \setminus a]\}$

The rule is contingent on certain constraints in the way P and Q are constituted: z_0 may not appear in Q and z may not appear in P . These constraints are entirely sensible.

z is used to return a result from R_{body} ; when the call is made it is assumed to be unassigned (to have no value). Thus, its initial value, z_0 , is irrelevant to the final outcome of R_{body} and therefore ought not to appear in the postcondition, Q .

Similarly, it would not make sense to refer to z in the precondition, P . Since z 's value is not communicated to R_{body} , its initial value cannot play any meaningful role in the procedure's outcome and therefore has no place in the procedure's precondition, P .

A simple example illustrates the rule in action. Consider the following procedure.

proc $R(\text{result } z), w := x + 1, w + 1$ **corp**

Here, x may be viewed as a global variable, or perhaps as a local constant. Whatever the case, the procedure's designer would have to advertise that the procedure conforms to a specification that assumes knowledge that the user knows something about the value of x . For example, the following specification may be advertised:

$$z, w : [P, Q] \text{ where } P \equiv \text{true and } Q \equiv (w > w_0 \wedge z > x)$$

(That the body actually adheres to this claim can readily be verified.)

Suppose that we, as users of the procedure, require that the call $R(a)$ should result in $Q' \equiv (w > w_0 \wedge a > x)$. From the designer's specifications it is evident that $Q' \equiv Q[z \setminus a]$. The rule allows us to conclude that

$$\begin{aligned} & \{\text{true}\} w, a : S \{(w > w_0 \wedge a > x)\} \\ & \sqsubseteq \{\text{true}\} R(a) \{(w > w_0 \wedge a > x)\} \end{aligned}$$

which is just another way of saying that the call $R(a)$ is a refinement of a specification that seeks to attain the postcondition from an arbitrary (**true**) starting scenario, and that, in turn, is a longwinded way of saying that the $R(a)$ will achieve our desired postcondition, irrespective of the starting state.

5.2.4 Pass by Value Result

A “pass-by-value-result” parameter is a combination of the previous two parameter types. The actual parameter, which is again constrained to be a variable, both initialises the value of the formal parameter at the start of the procedure and the value of the formal parameter is passed back to the actual parameter at the termination of the procedure. The rule indicates the circumstances under which $R(a)$ may be invoked when we are given the procedure **proc** $R(\text{value result } z) R_{body}$ **corp**. The rule appears as a combination of Rule 11 and Rule 12.

Rule 13. *Pass by value-result rule*

Given: **proc** $R(\text{value result } z) \{P\} w, z : R_{body} \{Q\}$ **corp**

Rule: $\{P[z \setminus a]\} w, a : S \{Q[z_0, z \setminus a_0, a]\} \sqsubseteq \{P[z \setminus a]\} R(a) \{Q[z_0, z \setminus a_0, a]\}$

In contrast to the pass-by-result rule, the rule does not require that z may not be appear in P . Similarly, in contrast to the pass-by-value rule, the rule does not require that z may not appear in Q .

Consider the simple example that differs from the previous one only in that z is now passed by value-result, and z replaces the role of x in the previous example.

proc $R(\text{value result } z)$ $z, w := z + 1, w + 1$ **corp**

It is easy to verify that the procedure's body complies with the specification:

$$\{\text{true}\} z, w := x + 1, w + 1 \{(w > w_0 \wedge z > z_0)\}$$

Taking P in the rule as **true**, Q as $(w > w_0 \wedge z > z_0)$ and applying the rule's substitutions, we get:

$$\begin{aligned} & \{\text{true}\} w, a : S \{(w > w_0 \wedge a > a_0)\} \\ & \sqsubseteq \{\text{true}\} R(a) \{(w > w_0 \wedge a > a_0)\} \end{aligned}$$

Thus, invoking $R(a)$ from any start state guarantees that both w and a will be larger than their respective values before the invocation.

5.2.5 Functions

Functions are easily refined as they are special cases of procedures. A function returns values, and thus any function can be expressed as a procedure with an additional pass-by-result parameter for each of its return values. For example, the following snippet

proc $Example(\text{result } x, \text{result } y, \text{value } x) \dots x, y := 0, 1$ **corp**
 \dots
 $Example(p, q, r)$

is equivalent to

func $Example(\text{value } z) : \langle x, y \rangle \dots a, b := 0, 1$ **ncuf**
 \dots
 $p, q := Example(r)$

To refine a specification into a function call, simply show that the refinement is valid for the procedure version of the procedure and then rewrite the procedure in its function form.

5.3 Procedure Refinement Strategy

The procedure refinement rules can be used as the basis of the following steps to develop a procedure.

1. Choose a name for the procedure that describes its functionality. Suppose we choose *Compute* for illustrative purposes. We also use *Parms* as a temporary placeholder for the unspecified parameters. The procedure then has form:

proc *Compute*(*Parms*) *Body* **corp**

2. Decide what the procedure should do in terms of the pre- and postconditions of *Body*. This will put the procedure in the form:

proc *Compute*(*Parms*) {*P*} *Body* {*Q*} **corp**

3. Decide on what inputs the procedure should take and what outputs it should produce. Then add parameters to reflect these. The pre- and postconditions should provide a clue as to the mode of each parameter. Variables in the precondition only are probably pass-by-value parameters; variables in the postcondition only are probably pass-by-result parameters; and variables in both the pre- and postcondition are probably pass-by-value-result parameters. This will put the procedure in the following form:

proc *Compute*(**value** *in*; **result** *out*; **value result** *inout*)
 {*P*} *in, out, inout* : *S* {*Q*}
corp

where *in*, *out* and *inout* may represent lists of parameters of those respective modes. Note that at this stage *S* represents an abstraction of the code that is yet to be instantiated.

4. Refine the body of the procedure into code using refinement laws such as the assignment and selection laws. This will put the procedure in the form:

proc *Compute*(**value** *in*; **result** *out*; **value result** *inout*)
 {*P*} ... code ... {*Q*}
corp

5. Decide on the actual parameters to be used when calling the procedure from the main program. Let *ina*, *outa* and *inouta* represent the actual pass-by-value, pass-by-result and pass-by-value-result parameters, respectively. The refinement rules indicate that a call to the procedure with these actual parameters will result in the following correct specification:

$$\begin{aligned} &\{P[in, inout \setminus ina, inouta]\} \\ &\quad \text{Compute}(ina, out, inouta) \\ &\{Q[in_0, in, 'out, inout_0, inout \setminus ina_0, ina, outa, inouta_0, inouta]\} \end{aligned}$$

6. If the procedure has pass-by-result parameters, it may be refactored into a function, making the necessary adjustments to the call in the main program. This step is entirely optional, but may make the final program easier to read and understand.

5.4 Recursive Procedures

Now that the refinement laws for procedures have been presented, we should be able to refine recursive procedures. Let us begin with a classic recursive example, the factorial function.

The factorial of some positive integer n , written $n!$, is defined as the number $1 \times 2 \times \cdots \times n$, or, in product of a sequence notation:

$$n! = \prod_{k=1}^n k$$

$0!$ is considered to be a special case and is defined as having the value 1.

Following the strategy in Sect. 5.3, we choose a name for the procedure giving

proc *Factorial*(*Parms*) *Body corp*

The procedure should take a non-negative integer, say n , as input and produce as output its factorial value, say f . We replace *Body* with this specification:

$$\{n \geq 0\} n, f : \text{Body} \{f = n_0!\}$$

Note that we use n_0 in the postcondition instead of n . This clarifies that even if the value of n changes within the procedure, f must eventually be the factorial of the initial value of n . If, instead, the specification had simply been $\{n \geq 0\} \text{Body} \{f = n!\}$, then $n, f := 0, 1$ would be a valid refinement of the specification, but contrary to what we intended to specify.

Since n appears in the precondition but not in the postcondition (n_0 appears in the postcondition instead), n may be a pass-by-value parameter (or even a pass-by-value-result parameter, but not a pass-by-result parameter).

On the other hand, f does not appear in the precondition, but does appear in the postcondition. Hence it may be a pass-by-result parameter (but not a pass-by-value parameter and not a pass-by-value-result parameter). The procedure now looks like this:

proc *Factorial*(**value** n , **result** f) $\{n \geq 0\} n, f : \text{Body } \{f = n_0!\} \text{ corp}$

The next step is to refine the body of the specification into code. Since this is a recursive procedure, we know that as part of the code refinement, *Factorial* should call itself. Since the formal and actual parameters of a recursive function will have the same names, we distinguish the formal parameter corresponding to some actual parameter x by x' .

Using this notation, and leaving aside termination issues for the moment, suppose that we wanted to argue that the following refinement is allowed by the procedure refinement rules:

$$\begin{aligned} & \{n' \geq 0\} n, f : \text{Body } \{f' = n'_0!\} \\ \sqsubseteq & \text{ Using Rule 11 for } n \text{ and Rule 12 for } f \\ & \{n' \geq 0\} \text{Factorial}(n', f') \{f' = n'_0!\} \end{aligned} \quad (5.1)$$

This code is supposed to set f' to $n'!$ when it terminates, and since $f = f'$ and $n = n'$ in the first recursive call, the final outcome should indeed then be that $f = n$.

The problem here, though, is that the refinement is premised on the assumption that

proc *Factorial*(**value** n , **result** f) $\{n \geq 0\} n, f : \text{Body } \{f = n_0!\} \text{ corp}$

is given (check the rules to see this), and in particular, that

$$\{n \geq 0\} n, f : \text{Body } \{f = n_0!\} \quad (5.2)$$

is given—i.e. known to be a **true** predicate. However, we do not know yet that the predicate is **true**. To prove it **true**, we would have to show that if the precondition holds, and *Body* executes, then *Body* will terminate and the postcondition will hold. This cannot be done directly, since the contents of *Body* has not yet been determined.

We could, in principle, make an assumption about the content of *Body*. We could say: suppose predicate 5.2 is **true**, and on this assumption, we apply a procedure refinement rule. Do we then end up with something contradictory or not? If so, then the assumption would have been invalid; otherwise it can be accepted.

In the present case, if we made that assumption and applied the procedure rule as just discussed above, then *Body* would contain a recursive call to itself,

and the call would have the actual parameters that correspond exactly to the formal parameters. The procedure would clearly end up recursing infinitely! In fact, therefore, predicate 5.2 would have been shown, *ex post facto*, to be **false**.

In deriving recursive procedures, we therefore face the two common problems hinted at above: finding a way of recursively calling the procedure that guarantees termination; and applying the refinement rules on the assumption that their application is valid, and only being able to verify that application thereafter. This latter verification can usually be done by using induction as a proof technique. In the material below we shall not carry out these proofs, since this will deflect from our main purpose. However, they are not difficult, and can easily be developed by the interested reader.

The following section, then, proposes a strategy for developing terminating recursive procedures. Even though the strategy does not pretend to be a universal recipe for developing recursive algorithms, it will be useful for many problems.

5.5 Terminating Recursive Programs

Chapter 2 presented a strategy for guaranteeing loop termination. Using this strategy for iteration as a guide, we will formulate a strategy to develop recursive procedures that are guaranteed to terminate once called. This strategy is offered, not as a panacea, but as an approach that will work for most recursive algorithms.

The second step of Chap. 2's iteration strategy mentions a predicate, G , which serves as the loop's condition. While G is **true** the loop continues to execute, and when G becomes **false** the loop terminates. It turns out that a similar predicate will be useful in recursive procedures as well.

For any recursive procedure we need to find one or more *base cases*. A base case refers to a condition under which the result to be returned is directly known or can easily be computed—i.e. a recursive call is not to be made. Suppose that there are $i > 0$ different base cases and that they occur respectively under the conditions B_1, B_2, \dots, B_i . Let $G = (B_1 \vee B_2 \vee \dots \vee B_i)$. The general idea is therefore to execute one of the base cases whenever conditions warrant it, and to make a recursive call whenever $\neg G$ holds.

However, merely to include code to handle base cases is not enough. Even though the recursion is guaranteed to terminate if G holds, there is no guarantee that conditions will eventually be reached to ensure that G does in fact hold.

Again we look to the iteration strategy for inspiration. This strategy, as part of its third step, introduced a variant, V , which must satisfy the conjuncts $(0 \leq V) \wedge (V < V_0)$ at the end of each iteration of the loop (as part of the loop body's postcondition).

For recursion, we also introduce a variant V , but with a few special properties.

- V is formulated in terms of variables and/or formal parameters. This has several implications. For discussion purposes below, we denote the variant as

$V(in, inout, x)$, where in is a **value** parameter, $inout$ is a **value result** parameter and x is a variable (or possibly a list of variables).

- Since V is used for logical reasoning only, the scope of x is not critical. It may be global or local. Local implies that, in the current recursive execution, the variable's value from a previous recursive execution will be unknown at the *code level*. However, reasoning at an *abstract/logical level* about its value in previous recursions and how its value may affect V , remains a perfectly legitimate exercise.
- In reasoning about the value of the variant at any particular stage of the computation, the formal parameters in and $inout$ obviously need to be replaced by the value of the actual variables (say ina and $inouta$) at each stage.
- In the context of recursion, a variant will not be formulated in terms of **result** parameters.
- The variant expression should consistently decline with each recursive invocation of the procedure, i.e. the predicate $(0 \leq V) \wedge (V < V_0)$ should hold. But this raises two related questions. At which point should the predicate hold? And what should V_0 designate at that point?
 - In answer to the first question, we require that $(0 \leq V)$ and $(V < V_0)$ should hold just before executing the first command in the body of the called recursive procedure—i.e. $(0 \leq V)$ and $(V < V_0)$ should be conjuncts of the recursive procedure's precondition.
 - In answer to the second question, we need to consider two scenarios. In the first scenario, we assume that a recursive call had been issued by a previous execution of the procedure. In that case, $V_0 = V_0(in, inout, x)$ is taken as the variant's value at a particular point in the previous recursive execution; namely, the point just before determining which guarded command of the select statement should be executed. At that point, V_0 's value is such that the $\neg G$ guarded command will be selected. Subsequent to selecting that command but prior to issuing the recursive call, V_0 's value might be changed.

The second scenario needing consideration is when a procedure is called for the first time—i.e. it is called from a main program. In this case, for convenience we shall assume that the value of $V_0 = Max$, where Max is the maximum value that can be represented in the system—i.e we assume that $V < V_0$ always holds at the first call.

These two additional considerations (base cases and variants) suggest a strategy for developing recursive procedures that are guaranteed to terminate. The recursive refinement strategy is as follows:

1. Follow steps 1 through 3 of the general strategy for defining a procedure presented in Sect. 5.3—i.e. choose a name for the procedure, determine the pre- and postconditions of the procedure and determine the parameters for the procedure.

2. Identify base case guards, B_1, B_2, \dots, B_i . Also identify the values R_1, R_2, \dots, R_i that should be returned in each of these cases respectively. Introduce a select command with guards B_1, B_2, \dots, B_i and $\neg G$.⁶ Following the convention used in the general strategy for defining a procedure, the procedure should now be in the form:

```

proc Compute(value in; result out; value result inout)
  {pre}
  if  $B_1 \rightarrow \{pre \wedge B_1\} out := R_1 \{post\}$ 
    :
   $\parallel B_i \rightarrow \{pre \wedge B_i\} out := R_i \{post\}$ 
   $\parallel \neg G \rightarrow \{pre \wedge \neg G\} S \{post\}$ 
  fi
  {post}
corp

```

3. Define a variant, V , for the recursion. As shown in the procedure outline below, revise the precondition by adding $(0 \leq V) \wedge (V < V_0)$ as additional conjuncts. The revised precondition is denoted as pre' .

```

proc Compute(value in; result out; value result inout)
  { $pre' \equiv (pre \wedge (0 \leq V) \wedge (V < V_0))$ }
  if  $B_1 \rightarrow \{pre' \wedge B_1\} out := R_1 \{post\}$ 
    :
   $\parallel B_i \rightarrow \{pre' \wedge B_i\} out := R_i \{post\}$ 
   $\parallel \neg G \rightarrow \{pre' \wedge \neg G\} S \{post\}$ 
  fi
  {post}
corp

```

4. Refine $\{pre \wedge \neg G\} S \{post\}$. Notionally, one might typically use the sequence rule to arrive at the form

$$\{pre \wedge \neg G\} S0; \{mid0\} S1; \{mid1\} S2 \{post\}$$

⁶It is not necessary to prove the proviso for the select rule here since the disjunction of the guards is $(G \vee \neg G)$ which is always **true**. Since all preconditions everywhere imply **true**, this select command as part of the recursive refinement strategy will always be valid.

where $S0$ involves a phase of preparation for a recursive call in $S1$. This preparation might typically include a decrementing of V . Indeed, very often no explicit preparation is necessary so that $S0$ is simply empty, and may be ignored.

The choice of the *mid1* predicate should articulate one or more subproblems implicit in *post* that can be solved by one or more recursive calls to this procedure. $S1$ should then be refined to attain *mid1*. The actual parameters of all recursive calls (and global variable values before the calls) should be such that they comply with the preconditions of the recursively called procedure, specifically in regard to the variant.

In each recursive call revisions to the pass-by-value parameters, *in*, and/or pass-by-value-result parameters *inout* will typically drive down the variant in the recursively called procedure. For descriptive purposes below, we notionally describe this by assuming functions F and F' which transform *in* and *inout* to their values required in the call. These functions may be evaluated as part of $S0$ indicated above, and/or they might represent direct substitutions of formal parameters by appropriate expressions (in the case of *in* variables). Thus, we assume that $S1$ may be represented by the recursive call $\text{Compute}(F(in), out, F'(inout))$. Of course, in practice this might be very simplistic, since additional computation might be required before and after the recursive call, and indeed, more than one recursive calls may be required.

5. Derive code, here generically denoted by a function H , such that

$$\{mid\} out := H(out) \{post\}$$

is satisfied. After refinement, this puts the procedure in the form:

```

{pre ∧ 0 ≤ V < V0}
if B1 → out := R1
    ⋮
    || Bi → out := Ri
    || ¬G → {Possible preparation code for recursive call}
        Compute(F(in), out, F'(inout));
        out := H(out)
fi
{post}

```

6. Finally, follow steps 5 and 6 of the general strategy for defining a procedure presented in Sect. 5.2. That is, refactor the procedure into a function if desired, and call the procedure from the main program. In the case where the variant V includes global variant variables, the main program call may involve initialising these variables to appropriate values to satisfy $0 \leq V < V_0$. Recall, however, that we will always assume that this predicate is satisfied at the first invocation of the recursive procedure.

5.6 Recursive Examples

In this section, we will use the recursive refinement strategy to develop several recursive algorithms from specification.

5.6.1 Factorial

First, we will use the recursive refinement strategy to develop a terminating refinement of *Factorial* developed in Sect. 5.4. To recap, after steps 1 through 3 of the general procedure refinement strategy, we have a *Factorial* procedure of the form

proc *Factorial* (**value** n , **result** f) $\{n \geq 0\} n, f : S \{f = n_0!\}$ **corp**

so step 1 of the recursive refinement strategy is complete. Step 2 requires the identification of base cases. To complete this step, consider the definition

$$n! = \begin{cases} \prod_{k=1}^n k & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases} \quad (5.3)$$

which shows that $0!$ is a base case since its value is known and trivially returnable.

Many implementations of recursive factorial use only this base case and work properly, but from a semantic point of view, $0!$ is considered a special case of the factorial function rather than a trivially simple case (since $\prod_{k=1}^0 k$ is not defined). Keeping this in mind, $1!$ is trivially simple and also a good semantic fit since it represents the simplest defined case of $\prod_{k=1}^n k$. For this reason, we declare $B_1 \triangleq (n = 0)$ and $B_2 \triangleq (n = 1)$. Since the precondition specifies that $(n \geq 0)$ we may take $\neg G$ to be

$$\neg G \equiv (n \neq 0 \wedge n \neq 1) \wedge (n \geq 0) \equiv (n \geq 2)$$

The refinement is therefore:

```
proc Factorial(value  $n$ , result  $f$ )
   $\{n \geq 0\}$ 
  if  $(n = 0) \rightarrow f := 1$ 
  ||  $(n = 1) \rightarrow f := 1$ 
  ||  $(n \geq 2) \rightarrow \{n \geq 2\} S \{f = n_0!\}$ 
  fi
   $\{f = n_0!\}$ 
corp
```

For step 3, it is necessary to define a variant V which will satisfy $0 \leq V < V_0$ as part of the precondition of each recursive call. Intuitively, n makes a good choice for V , since n is bounded from below by zero. (We know that when n is 0 or 1 there is no recursive call.)

We now have $V \equiv n$, and we add $0 \leq n < n_0$ to the precondition. The precondition to the procedure is therefore modified to

$$(n \geq 0) \wedge (0 \leq n < n_0) \equiv (n \geq 0) \wedge (n < n_0)$$

The resulting procedure outline is therefore:

```

proc Factorial(value  $n$ , result  $f$ )
     $\{(n \geq 0) \wedge (n < n_0)\}$ 
    if  $(n = 0) \rightarrow f := 1$ 
    ||  $(n = 1) \rightarrow f := 1$ 
    ||  $(n \geq 2) \rightarrow \{n \geq 2\} S \{f = n_0!\}$ 
    fi
     $\{f = n_0!\}$ 
corp

```

Recall that we assume that when the initial call from the main program is made, then the variant part of the precondition is initially met by default.

Step 4 now advises us to use the composition rule, as part of step 4, to refine the specification of the last guard into

$$\{n \geq 2\} n, f : S1 \{mid\}; n, f : S2 \{f = n_0!\}$$

The challenge is to determine *mid* in such a way that: (1) $S1$ can be refined to a recursive call that attains *mid*; and (2) as part of $S2$, something can be done to f so that $f = n_0!$ is satisfied.

Equation (5.3) shows that the factorial's definition can be restated recursively as

$$n! = n \times (n - 1)! \quad (5.4)$$

provided that $n > 0$. This broadly suggests the way ahead:

- Define *mid* as $\{f = (n_0 - 1)!\}$ and satisfy *mid* by regarding $S1$ as the recursive call *Factorial* $((n - 1), f)$.
- Then refine $S2$ to $f := f \times n$ to attain the postcondition $\{f = n_0!\}$. (Thus, in terms of the earlier discussion, the function $H(f)$ is defined as $f \times n$.)

Let us therefore show that the following is true.

$$\{n \geq 2\} \text{Factorial}((n - 1), f) \{f = (n_0 - 1)!\}; f := f \times n \{f = n_0!\} \quad (5.5)$$

We begin by addressing the first part of (5.5). Assume that the specification of the *Factorial* program is correct, i.e. that $\{(n \geq 0) \wedge (n < n_0)\} R_{body} \{f = n_0!\}$ is **true**, where R_{body} designates the body of the program that is still being evolved. (As suggested earlier, this can later be verified by an inductive argument.) Noting that $(n - 1)$ is to be used as an actual parameter for the formal parameter n we apply Rule 11 to get:

$$\begin{aligned}
 & \{(n \geq 0) \wedge (n < n_0)[n \setminus (n - 1)]\} n, f : S1 \{f = n![n \setminus (n - 1)]\} \\
 \sqsubseteq & \{\text{Rule 11}\} \\
 & \{(n - 1 \geq 0) \wedge (n - 1 < n_0)\} \text{Factorial}((n - 1), f) \{f = (n - 1)!\} \\
 \equiv & \{\text{Since } (n - 1 \geq 0) \equiv (n \geq 1) \text{ and since } (n - 1 < n_0) \equiv \text{true}\} \\
 & \{(n \geq 1)\} \text{Factorial}((n - 1), f) \{f = (n - 1)!\} \tag{5.6}
 \end{aligned}$$

This refinement sequence assures us that starting from $n \geq 1$ and calling $\text{Factorial}((n - 1), f)$ will guarantee that $f = (n - 1)!$ will subsequently hold. The postcondition of (5.6) happily co-incides with the first postcondition encountered in the specification (5.5). However, specification (5.5) has a stronger precondition than (5.6), namely $(n \geq 2)$ instead of $(n \geq 1)$. Clearly, then,

$$(n \geq 2) \text{Factorial}((n - 1), f) \{f = (n - 1)!\} \tag{5.7}$$

is guaranteed to hold, if (5.6) holds.⁷

Seen in terms of the function F mentioned in the narrative of step 4, this function has implicitly been defined as $F(x) = x - 1$, so that the recursive call $\text{Factorial}(n - 1, f)$ and $\text{Factorial}(F(n), f)$ are equivalent.

To verify the assignment in the second part of (5.5) we need to prove that $f = (n_0 - 1)! \Rightarrow f = n_0![f \setminus f \times n]$. This follows directly done.

Sticking all the pieces of code together gives the final refinement of the factorial procedure:

```

proc Factorial(value  $n$ , result  $f$ )
   $\{n \geq 0\}$ 
  if  $(n = 0) \rightarrow f := 1$ 
  ||  $(n = 1) \rightarrow f := 1$ 
  ||  $(n \geq 2) \rightarrow \text{Factorial}(n - 1, f); f := n \times f$ 
  fi
   $\{f = n!\}$ 
corp

```

⁷Note carefully that we are not arguing here that (5.7) refines (5.6). It patently does not, since it involves a strengthening of the precondition, not a weakening thereof. Indeed, the reverse is true, by virtue of the “weaken precondition” refinement rule, i.e. (5.7) \sqsubseteq (5.6). Rather, we are claiming that if a specification is known to be true for a weak precondition, it is guaranteed to remain true for a stronger precondition.

While this refinement is already complete, in the interests of completing all the steps in the strategy, the procedure can be easily refactored into a function by making f into a return value and compacting the recursive call and assignment of f into a single statement as follows:

```
func Factorial(value  $n$ ) :  $\langle f \rangle$ 
   $\{n \geq 0\}$ 
  if  $(n = 0) \rightarrow f := 1$ 
  ||  $(n = 1) \rightarrow f := 1$ 
  ||  $(n \geq 2) \rightarrow \{n \geq 0\} f := n \times \text{Factorial}(n - 1) \{f = n!\}$ 
  fi
   $\{f = n!\}$ 
cnuf
```

Finally, suppose that a main program has been specified by

$$\{x = 7\} y : S \{y = x_0!\}$$

Intuitively, we would want to replace S by a recursive call to the derived function, and return its value in y , i.e. we would simply say that this specification is adhered to by $y := \text{Factorial}(x)$. Let us verify this intuition by meticulous adherence to the refinement rules. Since we rely on our convention that this first call always complies with the variant used to derive the recursive procedure, we will not include the variant as part of the procedure's precondition.

$$\begin{aligned}
 & \{x = 7\} y : S \{y = x_0!\} \\
 \sqsubseteq & \quad \{\text{Weaken precondition}\} \\
 & \{x > 0\} y : S \{y = x_0!\} \\
 \equiv & \quad \{\text{Reversed substitution}\} \\
 & \{n > 0[n \setminus x]\} y : S \{f = n_0![n_0, f \setminus x_0, y]\} \\
 \sqsubseteq & \quad \{\text{Rule 11 (pass-by-value) and Rule 12 (pass-by-result)}\} \\
 & \{n > 0[n \setminus x]\} y := \text{Factorial}(x) \{f = n_0![n_0, f \setminus x_0, y]\} \\
 \equiv & \quad \{\text{Forward substitution}\} \\
 & \{x > 0\} y := \text{Factorial}(x) \{y = x_0!\}
 \end{aligned}$$

This final form gives code that is a refinement of the specification stated for the main program. It guarantees that if the actual variable x is initialised to *any* positive value, the code will terminate and deliver in y a value that is the factorial of x . In particular, if x is initialised to 7 to conform with the main program's precondition, the code will compute 7!. Henceforth, we will take the liberty of following our intuition, instead of giving this complete justification for making the recursive call from the main program.

5.6.2 Searching a List

In Chap. 3, an iterative linear search algorithm was developed that searches through elements of an array. Here we develop a recursive linear search algorithm that searches through elements of a list.

Consider a list \mathcal{L} of elements E_0, E_1, \dots, E_{n-1} where n is the length of \mathcal{L} and each element contains some arbitrary integer value. The following concepts and operations are defined:

- $Val(E)$ denotes the value of the element E .
- $H(\mathcal{L})$ denotes the first element (index zero) of the list \mathcal{L} .
- $T(\mathcal{L})$ denotes a sublist of \mathcal{L} that contains all the elements of \mathcal{L} in order, except for $H(\mathcal{L})$.⁸
- A special “terminating element” T is defined as having the property that $Val(T)$ returns the special value `NULL`.
- The predicate $NT(\mathcal{L})$ returns `true` if and only if the last element of list \mathcal{L} and only the last element is a terminating element. We say that such a list is null-terminating.
- The length of a null-terminating list, $\mathcal{L}.len$, does not include the terminating element, so for an empty null-terminating list \mathcal{L} , we have $\mathcal{L}.len = 0$.⁹

Let x denote some integer value and \mathcal{L} denote a non-terminating list as defined above. We are going to consider a problem whose precondition is:

$$pre \triangleq NT(\mathcal{L})$$

We will develop a recursive linear search algorithm to search \mathcal{L} for the value of x . If the value of x appears in an element within \mathcal{L} , then variable i must represent the index of that element (starting from index zero), otherwise i must have the value -1 .

We rely on a similar predicate to the one used for developing the iterative array linear search algorithm developed in Chap. 3, namely

$$appears(\mathcal{L}, V) \triangleq \exists j : [0, \mathcal{L}.len) \cdot (V = Val(\mathcal{L}_j))$$

which is `true` if an element with the value of V appears in \mathcal{L} and `false` otherwise.

As discussed above, x may or may not match an element in \mathcal{L} and the postcondition must take both of these possibilities into account. Thus, the postcondition can be expressed as

$$post(\mathcal{L}, x, i) \triangleq (appears(\mathcal{L}, x) \wedge Val(E_i) = x) \vee (\neg appears(\mathcal{L}, x) \wedge (i = -1))$$

⁸*Head* and *Tail* are analogous to the *CAR* and *CDR* pointers in LISP.

⁹This terminating list design is based on the *Typelist* struct developed by Alexandrescu for the *Loki* library [2].

The specification of the problem can be stated formally as

$$i : [NT(\mathcal{L}), post(\mathcal{L}, x, i)]$$

Commencing at step 1 of the suggested strategy for developing recursive procedures, note that the procedure that we will develop finds an index of an element. Hence we select *IndexOf* as its name. The formal specification statement of *IndexOf* is almost the same as that just given for the problem, namely $i : [NT(L), post(L, z, i)]$. The only difference is that the specification of *IndexOf* relies on formal parameters L and z , whereas the previous specification referenced the specific values \mathcal{L} and x which will later serve as actual parameters for *IndexOf*.

Since L appears in both the pre- and postcondition we could pass it as a value-result parameter. However, since the procedure should not change the list itself, it is safer to pass L by value. z appears in the postcondition but once again, z should not be changed as part of the algorithm, so we also pass it by value. i is the output of the procedure and is passed by result. This gives us an initial form of

```

proc IndexOf(value  $L$ , value  $z$ , result  $i$ )
    { $NT(L)$ }
     $i : S$ 
    { $post(L, z, i)$ }
corp

```

With step 1 complete, step 2 requires that base cases be identified and handled. One obvious base case is where the list is empty (except, of course, for its terminating element whose presence is enforced by the precondition $NT(L)$). In such a case, the list cannot contain z , so i should be assigned to -1 to satisfy the postcondition.

A second base case exists when the first element of a list is an element with the value of z . Formally stated, when $Val(H(L)) = z$ we can return the index 0.

As suggested by step 2, we introduce a selection statement to handle these base cases. To reduce notational clutter, we leave out from the precondition to S explicit reference to the conjunct corresponding to $\neg G$ as well as to $V = V_0$. These conjuncts should always be regarded as implicitly present in subsequent reasoning. The result is:

```

proc IndexOf(value  $L$ , value  $z$ , result  $i$ )
    { $NT(L)$ }
    if ( $Val(H(L)) = NULL$ )  $\rightarrow i := -1$ 
    || ( $Val(H(L)) = z$ )  $\rightarrow i := 0$ 
    || (( $Val(H(L)) \neq NULL \vee (Val(H(L)) \neq z)$ )  $\rightarrow$ 
        { $NT(L) \wedge ((Val(H(L)) \neq NULL \vee (Val(H(L)) \neq z))$ }
         $S$ 
        { $post(L, z, i)$ }
    fi
    { $post(L, z, i)$ }
corp

```

Now we must define a variant in such a way that we can guarantee the recursive calls terminate. By passing an empty null-terminating list to the procedure it will not recurse, so by passing ever smaller sublists recursively, we can guarantee termination. Keeping this in mind, a good choice for V appears to be $V = L.len$. Step 3 is now complete.

Step 4 requires the use of composition to split the specification of the recursive case. We need to choose mid such that it is satisfied by a recursive call. Consider information that is inherent in the guard of this recursive case. It consists of two disjuncts. One of the disjuncts affirms that the $H(L) \neq z$; the other, $H(L) \neq NULL$, affirms that there is at least one more non-terminating element remaining in the tail of the list. This information indicates that a reasonable aim for mid would be to assert that i indicates where—if at all— z is to be found in the tail of the list. This is precisely what is asserted by $post(T(L), z, i)$, which we therefore select as our version of mid in this case.

From this point, to cut back on notational clutter, we will not refer to the guard conjuncts $H(L) \neq z$ and $H(L) \neq NULL$ in our subsequent reasoning. Neither will we refer to the variant, except to note that it will indeed decline from one recursive call to the next.

We want to refine as follows:

$$\begin{aligned} & \{NT(L)\} S \{post(L, z, i)\} \\ \sqsubseteq & \{\text{Sequence rule}\} \\ & \{NT(L)\} S1 \{post(T(L), z, i)\} ; S2 \{post(L, z, i)\} \end{aligned} \quad (5.8)$$

$$\begin{aligned} \sqsubseteq & \{\text{By some suitable justification}\} \\ & \{NT(L)\} IndexOf(T(L), z, i); \{post(T(L), z, i)\} \\ & S2 \\ & \{post(L, z, i)\} \end{aligned} \quad (5.9)$$

How do we justify the second refinement step—i.e. the step that leads from (5.8) to (5.9)? Clearly we have to appeal to Rule 11.

Rule 11 assumes the correctness of the body of *IndexOf* with respect to its stated pre- and postconditions. For our purposes, this correctness is assumed but it can be separately verified using induction subsequent to its derivation.

The rule then requires us to make the substitution $[L \setminus T(L)]$ in the precondition $\{NT(L)\}$ of the body of *IndexOf*. The rule also requires us to make the substitution $[L_0 \setminus T(L_0)]$ in the postcondition $\{post(L, z, i)\}$ of the body of *IndexOf*. Here, the L in postcondition $\{post(L, z, i)\}$ references the incoming list, L . It may therefore be treated as L_0 and replaced by its initial tail, denoted by $T(L)$. Applying the rule then explains the first step of following refinement sequence.

$$\{NT(T(L))\} S1 \{post(T(L), z, i)\} \quad (5.10)$$

$\sqsubseteq \{ \text{Using Rule 11} \}$

$$\{NT(T(L))\} IndexOf(T(L), z, i) \{post(T(L), z, i)\}$$

$\sqsubseteq \{ \text{Since } NT(L) \iff NT(T(L)) \}$

$$\{NT(L)\} IndexOf(T(L), z, i) \{post(T(L), z, i)\} \quad (5.11)$$

But notice carefully the following: we wanted to show (5.8) \sqsubseteq (5.9), and we have shown that (5.10) \sqsubseteq (5.11) holds instead. Fortunately, (5.9) and (5.11) match. However, the preconditions in (5.8) and (5.10) do not match. Nevertheless, as pointed out above, the precondition in (5.8) is equivalent to the precondition in (5.10)—i.e. $NT(L) \iff NT(T(L))$. We may therefore confidently assert that (5.8) \sqsubseteq (5.9).

The function F mentioned in step 4 of the refinement strategy corresponds in this present instance to $T(x)$. At the conclusion of this step we have:

```

proc IndexOf(value L, value z, result i)
  {NT(L)  $\wedge$  0  $\leq$  L.len < L0.len}
  if (Val(H(L))) = NULL  $\rightarrow$  i := -1
  || (Val(H(L))) = z  $\rightarrow$  i := 0
  || ((Val(H(L)))  $\neq$  NULL  $\vee$  (Val(H(L)))  $\neq$  z))  $\rightarrow$ 
    {NT(L)}
    IndexOf(T(L), z, i);
    {post(T(L), z, i)}
    S2
    {post(L, z, i)}
  fi
  {post(L, z, i)}
corp

```

To execute step 5, we note that if the index of element x in a list is denoted by i , then the index of x in a superlist created by appending an element to the beginning of the list must be $i+1$. Thus, if the sought-after entry is found at index i in the tail of L , then it must be at $i + 1$ in L itself. This suggests that we can define the H function as $H(x) = x + 1$.

Note, however, that we must also consider the possibility that the element does not occur in the sublist, ($i = -1$), and in such a case, i retains the value $i = -1$ with respect to the full list. This leads to the full definition of

$$H(x) = \begin{cases} x + 1 & x \geq 0 \\ x & x = -1. \end{cases}$$

Using an “as” command to represent this conditional change in the value of x , we have the complete code for the recursive function *IndexOf*:


```

proc IndexOf(value L, value z, result i)
    {NT(L) ∧ 0 ≤ L.len < L0.len}
    if (Val(H(L))) = NULL → i := -1
    || (Val(H(L))) = z → i := 0
    || ((Val(H(L))) ≠ NULL) ∨ (Val(H(L))) ≠ z)) →
        {NT(L) ∧ 0 ≤ L.len < L0.len}
        IndexOf(T(L, z), i);
        as (i ≥ 0) → i := i + 1 sa
        {post(L, z, i)}
    fi
    {post(L, z, i)}
corp

```

If desired, the procedure can now be refactored into a function. Either way, the procedure is ready to be called from a main program.

5.6.3 Evaluating an Expression Tree

Expression trees are often used in compilers to represent expressions as part of syntax analysis [1]. In this section a recursive algorithm will be developed to evaluate the expression represented in a binary expression tree (BET).

The primary use of recursion in this algorithm will be to traverse a recursive structure in order to carry out a specific computation. As such, it will be prudent to define exactly what is expected of this recursive structure.

The structure to be used is a BET. The predicate $BET(T)$ evaluates to **true** if and only if T abides by the following rules:

- $G(T, A)$ is a connected acyclic graph with set of nodes T and set of arcs A . Because A is not further referenced in this discussion, we shall simply refer to the graph as T .
- $T = N_0, N_1, \dots, N_{k-1}$, where $|T| = k < \infty$ denotes the number of nodes in T .
- Each node in T has exactly one parent, which is another node in T , except for the single “root” node. The $Root(T)$ function returns the root node of T .
- Each node in T has either zero children or two children. The predicate $Term(N)$ is **true** if and only if the node N has zero children.
- If a node N has two children, one is referred to as the left child and the other is the right child and these are returned for node N by $L(N)$ and $R(N)$ respectively. The parent of both $L(N)$ and $R(N)$ is of course N .
- If $Term(N)$ is **false** for some N , then N holds a reference to an operator. For simplicity we will deal only with two binary operators. $Op(N)$ returns either the special value ‘+’ for addition or the special value ‘×’ for multiplication.

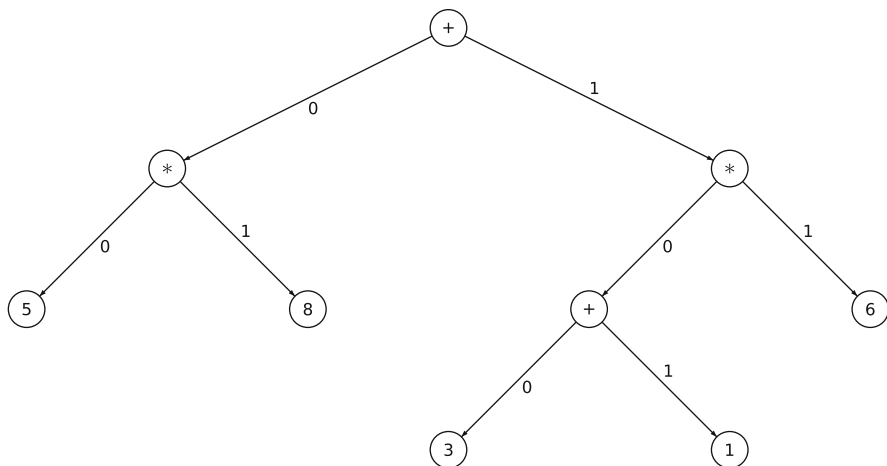


Fig. 5.1 A binary expression tree that evaluates to 64

- If $Term(N)$ is **true** for some N , then N holds a reference to an integer value. This value is denoted by $Val(N)$
- The function $T(R)$ returns the tree T formed with the node R as its root.

The above rules imply that all the nodes N in a BET T share a single common ancestor: the root node of the tree.

A BET is evaluated by evaluating each node, starting from leaf nodes and working upwards. A leaf node, N has the value $Val(N)$. The value of a non-leaf node, N , whose two children are leaf nodes L and R respectively is $Val(L) Op(N) Val(R)$. This value is recursively used to evaluate non-leaf nodes higher up in the tree. The value of the BET (i.e. the value of the binary expression which the tree represents) is given is the value of the tree's root.

The value of the BET T will be denoted by $Eval(T)$. Note carefully that $Eval(T)$ as used here is a mathematical function, not the name of a procedure or function in a computer program.

Consider, for example, the BET in (Fig. 5.1). The evaluation of the binary tree T is $(5 \times 8) + ((3 + 1) \times 6) = 64$, i.e. $Eval(T) = 64$.

We develop here a recursive procedure to evaluate an arbitrary BET that conforms to the description given above.

Step 1 requires that we pick a name, specification and choose parameters for the procedure to be defined. Since we are evaluating a tree, *Evaluate* will be the name for the procedure. Note throughout that this procedure name should be sharply distinguished from the mathematical function previously mentioned, *Eval*, which returns an integer value.

The procedure's precondition asserts that the tree structure, T , to be traversed is a BET as defined above, since we will be using these rules to refine the

specification, so

$$pre \triangleq BET(T)$$

The postcondition asserts that some integer variable, say r , has the value of the evaluated tree, so

$$post \triangleq r = Eval(T)$$

T and r appear in the pre- and postconditions, so they make good candidates for *Evaluate*'s parameters. However, we know that later we will probably want to pass subtrees recursively to *Evaluate*, so a reference to a single root node, N , may be more useful than a reference to a tree T . Whenever we wish to refer to the full tree T , we can easily represent it as $T(N)$. Moreover, we can represent the two subtrees of node N as $T(L(N))$ and $T(R(N))$. Rewriting the pre- and postcondition in this form, the resulting specification becomes:

$$r : [BET(T(N)), r = Eval(T(N))]$$

This particular recursive algorithm does not change the tree structure or values in the nodes, so we pass N by value.¹⁰ The output variable r only appears in the postcondition and should be passed by result. We now have a procedure skeleton:

```

proc Evaluate(result  $r$ , value  $N$ )
    {  $BET(T(N))$  }
     $r : S$ 
    {  $r = Eval(T(N))$  }
corp

```

The specification needs to be refined with respect to S , the body of the recursive procedure to be developed.

It is necessary to determine base conditions where no recursive call is required. From the BET rules, it is known that when a node is terminal (it has no children) it holds a reference to an integer value that represents its evaluation. Under such circumstances, $Val(N)$ can simply be returned through an assignment of r . A select statement can be used to deal with this single base case. This leads to the procedure outline:

¹⁰However, a node will generally be represented by an object in an object-oriented programming languages, and most of these languages pass object parameters by a reference to the object. Semantically, a pass by reference is the same as a pass by value result, so in this example N could have been passed by value result. This would help to develop an algorithm that will be correct when implemented in an object oriented language such as Java, where objects can only be passed by reference.

```

proc Evaluate(result  $r$ , value  $N$ )
    { $BET(T(N))$ }
    if  $Term(N) \rightarrow r := PVal(N)$ 
    ||  $\neg Term(N) \rightarrow r : [BET(T(N)), r = Eval(T(N))]$ 
    fi
    { $r = Eval(T(N))$ }
corp

```

Ideally, each node in the tree should be visited exactly once for a complete traversal. As it turns out, this can be achieved by calling *Evaluate* on the left and right children of each non-terminal node N as long as we start at the root of the full tree: since the BET rules assert that the tree is connected and each non-root node has a parent, then clearly each node will be traversed (at least) at some point as the child of its parent beginning with the children of the root.

The rules also state that there are no loops or cycles among nodes, so each node will be traversed at most once. Since each node must be visited at least once and at most once, each node must be visited exactly once. This fact makes it easy to find a variant V .

We assume a global variable, L , is a set of nodes that is initialized to \emptyset before *Evaluate* is called. Assume that the call to *Evaluate* passes on the BET B as actual parameter.

Each time a node is visited *Evaluate* must insert the visited node into L , thus increasing its size. This way, as B is traversed, L will become larger until it is equal in size to B (at which point all the nodes of B will have been visited and no more recursive calls should be allowed.) This gives a variant that depends on the actual parameter used during the call, namely $V = |B| - |L|$. The code thus far is

```

proc Evaluate(result  $r$ , value  $N$ )
    { $BET(T(N)) \wedge (0 \leq |B| - |L| < |B| - |L_0|)$ }
     $L := L \cup \{N\};$ 
    if  $Term(N) \rightarrow r := Val(N)$ 
    ||  $\neg Term(N) \rightarrow r : [BET(T(N)), r = Eval(T(N))]$ 
    fi
    { $r = Eval(T(N))$ }
corp

```

Note that we have inserted at the beginning of the code an assignment statement to update L , since every entry into the procedure represents a visit to a node. This means that for every subsequent recursive call, the variant relative to its value at the beginning of this current recursive execution will have declined by one.

The recursive refinement strategy now suggests the use of composition to split the specification of the $\neg G$ guard into two. A *mid* must be found such that it is satisfied by a recursive call to *Evaluate*.

The evaluation of any tree is found by performing the operation of its root on operands derived from evaluating its left and right subtrees respectively. This suggests that *mid* should be satisfied by not one, but two recursive calls to *Evaluate*. As this is the case, it does not seem possible to store the result of the recursive calls in r as the second call will overwrite the first. Instead, it makes sense to introduce two temporary variables t_1 and t_2 to store the values of the evaluation of the left and right subtrees respectively. Later we will find a formula to derive the value of r from t_1 and t_2 . By using the composition rule a second time we are left with the sequence of specifications

$$\begin{aligned}
 & r : [BET(T(N)), r = Eval(T(N))] \\
 \sqsubseteq & \quad \{\text{Composition rule applied twice}\} \\
 & L, r, t_1, t_2 : [BET(T(N)), t_1 = Eval(T(L(N)))]; \\
 & L, r, t_1, t_2 : [t_1 = Eval(T(L(N))), \\
 & \quad \quad \quad t_1 = Eval(T(L(N))) \wedge t_2 = Eval(T(R(N)))]; \\
 & L, r, t_1, t_2 : [t_1 = Eval(T(L(N))) \wedge t_2 = Eval(T(R(N))) \\
 & \quad \quad \quad r = Eval(T(N))]
 \end{aligned}$$

Noting that the predicate $BET(T(N))$ is invariant, it is carried over into each *mid* predicate as an extra conjunct, and we get

$$\begin{aligned}
 \sqsubseteq & \quad \{\text{Rules 11 and 12}\} \\
 & \{BET(T(N))\} \\
 & \quad Evaluate(t_1, T(L(N))); \\
 & \{BET(T(N)) \wedge t_1 = Eval(T(L(N)))\} \\
 & \quad Evaluate(t_2, T(R(N))); \\
 & BET(T(N)) \wedge t_1 = Eval(T(L(N))) \wedge t_2 = Eval(T(R(N))); \\
 & \quad L, r, t_1, t_2 : S \\
 & \quad r = Eval(T(N))\}
 \end{aligned}$$

We will not justify the refinement step in detail. It relies on Rules 11 and 12 and requires substituting formal parameters with actual parameters, as specified by those rules. It also requires of us to ensure that only BETs are passed as actual parameters. That the left- and right subtrees passed as actual parameters in the recursive calls are indeed BETs follows directly from the fact that $BET(N)$ holds prior to the relevant call. This is the reason for writing $BET(N)$ explicitly as a conjunct in the various preconditions above.

The foregoing refinement can now be inserted into the code derived thus far. Refinement of the last part of the above specification can then proceed in terms of step 5 of our general strategy, the purpose being to derive appropriate code for S .

What is needed is a function H that will assign to r a value, in terms of t_1 and t_2 such that $r = \text{Eval}(T(N))$ is **true**. As previously pointed out, the value of a tree is equal to the root's binary operator applied to the evaluation of its left subtree (which we happen to have stored in t_1) and the evaluation of its right subtree (which we happen to have stored in t_2 .) Since N is non-terminal ($\text{Term}(N)$ returned **false** during the guard evaluation of the selection statement) the BET rules tell us that the function $\text{Op}(N)$ will return the operator of N , which will be either “+” or “×”. Using this knowledge, H can be written as

$$H(N, x, u, v) = \begin{cases} x := u + v & \text{if } \text{Op}(N) = “+” \\ x := u \times v & \text{if } \text{Op}(N) = “\times” \end{cases}$$

and this refines the procedure to its final form. The main program can now be refined to a call to *Evaluate*, sending through the root of B as a parameter:

```

proc Evaluate(result  $r$ , value  $N$ )
    {BET( $T(N)$ )}
     $L := L \cup \{\text{Address}(N)\};$ 
    if  $\text{Term}(N) \rightarrow r := \text{Val}(N)$ 
    ||  $\neg \text{Term}(N) \rightarrow$ 
        Evaluate( $t_1, T(L(N))$ );
        Evaluate( $t_2, T(R(N))$ );
        if ( $\text{Op}(N) = “+”$ )  $\rightarrow r := t_1 + t_2$ 
        || ( $\text{Op}(N) = “\times”$ )  $\rightarrow r := t_1 \times t_2$ 
        fi
    fi
    { $r = \text{Eval}(T(N))$ }
corp

```

We draw attention to the following general matters regarding *Evaluate()*.

Firstly, note that the global variant variable, L , need not have been global at all. Instead, L could easily have been passed by value-result to the procedure. However, passing these extra parameters to a procedure can make it cluttered and difficult to understand. In fact, L can be refactored out of the program altogether—its only purpose was to make explicit a variant in support of reasoning about termination. This would make the procedure slightly more portable, in the sense that the user (caller) would not need to initialise L or be otherwise concerned with it.

As a second closing observation, note that although the final algorithm defined for *Evaluate()* is relatively short and simple, this only because the algorithm relies heavily on the fact that it is traversing a well defined BET structure with many helpful restrictions and rules. When writing an algorithm to traverse over a recursive

structure like a tree, graph or lattice, it pays off to spend time identifying and defining what is expected from the structure. Doing so will make the algorithm simpler to write, as there is more information with which to work.

5.6.4 MergeSort

The final example details the derivation of the well-known von Neumann Merge-Sort algorithm. The algorithm is based on the idea that two sorted arrays can be easily merged into a single sorted array. A single array is broken down into many arrays of size one and zero, then these mini arrays are all merged together again to form a sorted array.

The function to merge two lists together is iterative, and thus will not be refined here. Instead, we will simply assume that it is already implemented with the signature

func *Merge*(**value** *Y*, **value** *Z*) : $\langle X \rangle$

and satisfies the specification: $X : [\text{sorted}(Y) \wedge \text{sorted}(Z), (X = Y \cup Z) \wedge \text{sorted}(X)]$

The first step, as always, is to choose a name, parameters and specification for the procedure. *MergeSort* is as good a name as any, and it should take an array as input. Its output should be the sorted array. To keep these two concerns separate, we pass the input array *A* by value and the output sorted array *S* by result. Let *perm*(*A*, *S*) denote that *S* is a permutation of *A*. This gives us:

$S : [\text{true}, \text{perm}(A, S) \wedge \text{sorted}(S)]$

The obvious base cases are that either *A* is empty or *A* contains only one element. In either case, *A* is already sorted and $S := A$ satisfies the specification. This gives:

```
proc MergeSort(value A, result S)
  if (A.len = 0) → S := A
  || (A.len = 1) → S := A
  || (A.len > 1) → S : [true, perm(A, S) ∧ sorted(S)]
fi
corp
```

The variant is where things get tricky. Consider the diagrams in Fig. 5.2a, b. Clearly there are two different “stages” in performing a merge-sort: The array is split, then merged. However, these actions are interleaved: calls to merge will happen in between calls to split, and thus a variant must be found to take into account both of these types of calls. Each of these two stages needs to be considered when constructing the variant, so *splitPart* + *mergePart* seems like a good place to start.

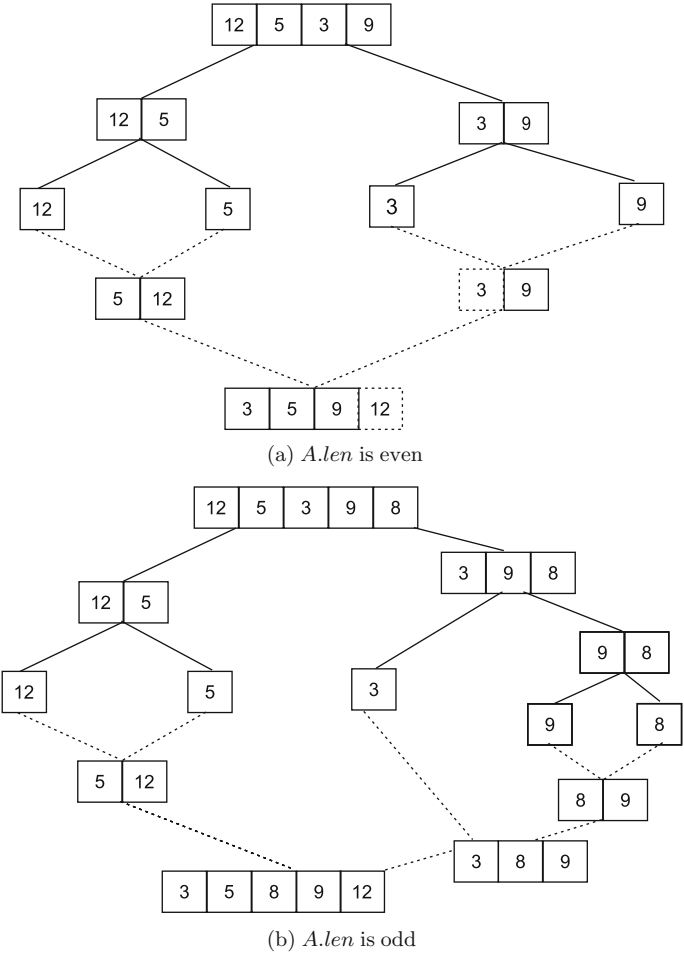


Fig. 5.2 MergeSort where *A.len* is even and where *A.len* is odd. (a) *A.len* is even (b) *A.len* is odd

We begin by considering the *splitPart*. The diagrams show that splitting the array forms a kind of tree (shown as black edges) with a particular depth. We begin by defining the maximum size, *maxSize*, of this tree to be the depth of the tree multiplied by the size of the array. This way we can also define a global variant variable, say *splitCount*, that stores combined sizes of the subarrays that have been split. As the arrays are split, their sizes are added to the variant variable, so that $(maxSize - splitCount)$ approaches zero.

The depth of the tree has a logarithmic relation to the size of the array. Since the array is split in two, the depth of the tree is $\log_2(A.len)$ rounded up to the closest integer, in addition to the depth (1) of the original array. This makes

$maxSize = (\log_2(A.len) + 1) \times size$ and thus $splitPart = (((\log_2(A.len) + 1) \times size) - splitCount)$. Now to consider the $mergePart$. The diagrams show that the merge part also forms a kind of tree (shown as grey edges) and has depth equal to one less than the split tree: $\log_2(A.len)$. After each merge, we add the size of the newly formed subarray to a global variable, say $mergeCount$. This way, as the subarrays are merged together $((\log_2(A.len) \times A.len) - mergeCount)$ approaches zero.

Combining all of this together we are left with

$$V = (\log_2(A.len) \times (A.len) - splitCount) + (\log_2(A.len) \times (A.len) - mergeCount)$$

The recursive refinement strategy now suggests the use of the composition rule. In a similar way to the *Evaluate* problem from the previous section, an array can be thought of as the concatenation of its left subarray and its right subarray. We should use the composition rule twice so that the specification is divided into three parts: The first part should be satisfied by a recursive call on the left subarray and the second part should be satisfied by a recursive call on the right subarray. To store the sorted versions of these arrays we will use two temporary variables T_1 and T_2 . We define

$$mid1 \triangleq (perm(A_{[0, A.len/2]}, T_1) \wedge sorted(T_1)) \text{ and}$$

$$mid2 \triangleq (perm(A_{[A.len/2, A.len]}, T_2) \wedge sorted(T_2))$$

which leads to the specification of the body of the $\neg G$ guard as

$$S : [true, mid1]; S : [mid1, mid1 \wedge mid2]; S : [mid1 \wedge mid2, perm(A, S) \wedge sorted(S)]$$

Before refining specifications to recursive calls we need to make sure the variant is satisfied. This can be done by making sure that $splitCount$ increases in size before either left or right is called and that $mergeCount$ increases after merge is called. The easiest way to achieve this is to carry out the assignment of $splitCount$ before the select statement and the assignment of $mergeCount$ after the call to $Merge()$. We can now refine to recursive calls, leaving the program in the form:

```

proc MergeSort(value A, result S)
  if (A.len = 0)  $\rightarrow$  S := A
  || (A.len = 1)  $\rightarrow$  S := A
  || (A.len > 1)  $\rightarrow$ 
    MergeSort(A[0, A.len/2], T1); MergeSort(A[A.len/2, A.len], T2)
    S : [mid1  $\wedge$  mid2, perm(A, S)  $\wedge$  sorted(S)]
  fi
corp

```

Now all that is required is to find a H which assigns the value of S in terms of T_1 and T_2 . It turns out that a call to $Merge(T_1, T_2)$ produces the required value of S , so

$$S : [mid1 \wedge mid2, perm(A, S) \wedge sorted(S)] \sqsubseteq S := Merge(T_1, T_2)$$

We must also remember to add $mergeCount := mergeCount + S.len$ to satisfy the variant. This leaves the procedure in its final form:

```

proc MergeSort(value A, result S)
  splitCount = splitCount + A.len;
  if (A.len = 0)  $\rightarrow$  S := A
  || (A.len = 1)  $\rightarrow$  S := A
  || (A.len > 1)  $\rightarrow$ 
    MergeSort(A[0,A.len/2), T1)
    ; MergeSort(A[A.len/2,A.len), T2)
    ; S := Merge(T1, T2)
    ; mergeCount := mergeCount + S.len
  fi
corp

```

Now the procedure can be called from a main program that sets the initial values of the global variant variables, *mergeCount* and *sortCount* to zero. As in the previous example, however, these counters are not an essential part of the algorithm, but were introduced in order to support reasoning about termination.

5.7 Conclusion

This chapter has illustrated how recursive algorithms can be derived in a formal correctness-by-construction fashion. A general strategy for deriving recursive algorithms has been outlined and illustrated by way of a number of examples. The strategy relies on refinement rules that indicate how a procedure invocation can be a refinement of a specification. The specification, in turn, corresponds to the pre- and postcondition of the called procedure, but where formal parameters are substituted by actual parameters. The way in which the substitution is to take place depends on the mode of the parameter: pass by value, result, or value-result.

Two important issues arise in the applying these rules. In the first place, the rules assume that the called procedure is correct in that it has a body that conforms to a specified pre- and postcondition. However, in the case of recursion, there is no way of knowing whether or not the called recursive program is indeed correct, because it is still under construction. It can only be formally shown to be correct once the algorithm has been developed. Induction would be the most suitable mathematical strategy for providing such proofs. However, no such proofs have been given in this chapter. Although we recognise the value and validity of mathematical rigour in providing such proofs, we nevertheless regard such proofs as outside the scope of this text. Instead, this chapter serves as a starting point for those who would wish to pursue more rigourously the theme of refinement calculus in the context of procedure invocation and recursive procedure construction.

A second issue worth mentioning at this point is the introduction of a variant as part of the general solution strategy. Although the formal definition of a variant

indeed gives confidence that the recursion will not be infinite, in most situations it might be something of an overkill. This became evident in the last two examples, where we saw that variables keeping track of the variant could be refactored out of the final solution—they serve no other role in the program except to indicate that progress is being made towards arriving at the base cases. From a pragmatic point of view, one could dispense with a variant if you have sufficient assurance that your successive recursive calls will always be on ever-smaller problems and therefore that the base case scenarios will eventually be reached. This is specifically the case when the recursive calls deal with data structures such as graphs, trees, lists, sets, etc which are partitioned into smaller data structures at each call.

The algorithm to be derived in the next chapter will depend significantly on recursion. However, we will neither prove the final algorithm correct by use of induction, nor will we formally define a variant to describe its progress towards termination, since this will be self-evident.



<http://www.springer.com/978-3-642-27918-8>

The Correctness-by-Construction Approach to
Programming

Kourie, D.G.; Watson, B.W.

2012, XIV, 266 p., Hardcover

ISBN: 978-3-642-27918-8