6
Proof

In this chapter we describe how you can use SPARK to prove certain correctness properties of your programs. When you ask to "prove" your code, the SPARK tools will by default endeavor to prove that it will never raise any of the predefined language exceptions that we describe in Section 6.1. If you additionally include pre- and postconditions, loop invariants, or other kinds of *assertions*, the tools will also attempt to prove that those assertions will never fail.

It is important to understand that proofs created by the SPARK tools are entirely static. This means if they succeed, the thing being proved will be true for every possible execution of your program regardless of the inputs provided. This is the critical property of proof that sets it apart from testing.

However, Ada assertions are also executable under the control of the assertion policy in force at the time a unit is compiled. Assertions for which proofs could not be completed can be checked when the program is run, for example during testing, to help provide a certain level of confidence about the unproved assertions. Testing can thus be used to complement the proof techniques described here to obtain greater overall reliability. We discuss this further in Section 8.4.

6.1 Runtime Errors

A *logical error* is an error in the logic of the program itself that may cause the program to fail as it is executing. It is an error that, in principle, arises entirely because of programmer oversight. In contrast, an *external error* is an error caused by a problem in the execution environment of the program, such as being unable to open a file because it does not exist. If a program is correct, it should not contain any logical errors. However, external errors are outside of a program's control and may occur regardless of how well constructed the

program might be. A properly designed program should be able to cope with any external errors that might arise. However, the handling of external errors is outside the scope of this book and is a matter for software analysis, design, and testing (see Black, 2007).

We distinguish a *runtime error* as a special kind of logical error that is detected by Ada-mandated checks during program execution. Examples of runtime errors include the attempt to access an array with an out of bounds index, arithmetic overflow, or division by zero. Other kinds of logical errors include calling a subprogram without satisfying its precondition, but those errors are not checked by the Ada language itself. However, you can include custom assertions to check for such errors. We describe how to do so in Section 6.2.

For programs written in an unsafe language such as C, runtime errors often produce "undefined behavior," referred to as *erroneous execution* in the Ada community, that commonly result in a crash. Such errors are often caused by exploitable faults and indicate potential security problems as well. In contrast, Ada raises an exception when a runtime error occurs. A careful programmer will provide exception handlers[1] to deal with runtime errors in some sensible way by, for example, reporting the error or even attempting to recover from it.

Unfortunately, in a high-integrity system after-the-fact handling of runtime errors is not acceptable. If a program does something illogical, such as divide by zero, that means whatever the programmer intended to happen did not actually happen. An exception handler can make a guess about how to proceed from that point, but it is only a guess. Because the error should never have occurred in the first place, the exception handler cannot know with certainty how to respond. For example, retrying the failed operation may cause the same illogical computation to be attempted again, potentially resulting in an infinite loop of ineffective error handling actions.

In a high-integrity context, it is important that runtime errors never occur. The ability SPARK gives you to construct a mathematical proof that a program has no runtime errors justifies SPARK's lack of support for exception handling. A program proved free of runtime errors will never raise an exception and thus has no need for exception handling. In Section 7.1 we discuss the issues arising when calling full Ada libraries from SPARK and how to deal with the exceptions those libraries might produce.

As a side effect, it may also be possible to create more efficient programs with SPARK than with full Ada, both in terms of space and time. If the program has been proved exception-free, it can be compiled with all runtime checks disabled. This allows the compiler to create a faster and more compact executable than might otherwise be the case. Finally, it may be possible to enjoy additional

savings by using a reduced Ada runtime system without exception handling support. We discuss how and when to suppress runtime checks in Section 6.8.

Proving that a program is free of runtime errors is the first level of proof to pursue in SPARK programming. It is important to understand, however, that showing freedom of runtime errors does not show that the program is "correct" in the intuitive sense. SPARK also allows one to prove higher level correctness properties. The techniques for doing so are discussed in the later sections of this chapter. Those techniques extend the methods described here. Proving freedom from runtime errors is the basis from which more complex proofs are constructed.

6.1.1 *Predefined Exceptions*

The SPARK language allows the programmer to explicitly raise an exception, but the SPARK tools attempt to prove that the **raise** statement will never actually execute. This feature allows you to, for example, declare a Not_Implemented exception and raise it in subprograms that are unfinished. The SPARK tools will not able to prove such a subprogram as long as there is a way for the exception to be raised, serving as a reminder that more work needs to be done[2].

The only exceptions that could conceivably arise in a pure SPARK program are those that are predefined in the Ada language and raised automatically by the Ada compiler when language-mandated checks fail. There are four predefined exceptions. They are potentially raised for a wide variety of reasons. In this section we describe these four predefined exceptions and outline how SPARK manages them.

Program_Error

The Program_Error exception arises in full Ada for a variety of program organizational problems that cannot always be detected by the compiler. The complete list of situations where Program_Error might be raised is long, but as one example, the exception will be raised if the execution of a function ends without returning a value. To see how that might happen, consider the following function that searches an array of integers and returns the index associated with the first occurrence of zero in the array.

```
subtype Index_Type is Natural range 0 .. 1023;
type Integer_Array is array(Index_Type) of Integer;

function Search_For_Zero (Values : in Integer_Array) return Index_Type is
begin
   for Index in Index_Type loop
```

```
      if Values (Index) = 0 then
         return Index;
      end if;
   end loop;
end Search_For_Zero;
```

The function ends as soon as the search is successful by returning from inside the loop. What happens if the array does not contain a zero value? In full Ada the function "falls off" the end and Program_Error is raised. This function is not necessarily wrong. If it is only used on arrays that contain at least one zero value, it will behave as intended. However, this function as written entails the possibility that an exception will be raised and that is not acceptable in a high-integrity application.

The preceding function is not legal SPARK because it can be shown statically to raise an exception – Program_Error in this case – under certain circumstances. One way to fix the problem would be to add a return statement to the path that currently falls off the end. The programmer would need to decide what value should be returned if a zero does not appear in the array.

An alternative approach would be to add a precondition to the function that ensures it is always called with an array containing at least one zero value. It is also necessary to add an explicit **raise** statement at the end of the function. The modifications are shown as follows:

```
subtype Index_Type is Natural range 0 .. 1023;
type Integer_Array is array(Index_Type) of Integer;

function Search_For_Zero (Values : in Integer_Array) return Index_Type
  with Pre => (for some Index in Index_Type => Values(Index) = 0)
is
begin
   for Index in Index_Type loop
      if Values (Index) = 0 then
         return Index;
      end if;
   end loop;
   raise Program_Error;
end Search_For_Zero;
```

The SPARK tools will attempt to prove that the explicit **raise** can never execute, and the precondition allows that proof to be successful. The SPARK tools will also attempt to prove that the precondition is satisfied at every call site. We discuss preconditions in Section 6.2.1.

The explicit **raise** also prevents the flow of control from falling off the end of the function and triggering the automatically generated exception. The SPARK tools do not see this as an error but instead regard the explicit **raise** as a signal that you want the tools to prove the raise statement will never execute.

You can also use an explicit **raise** of Program_Error as a way of documenting arbitrary "impossible" events in your program. The SPARK tools will then try to prove that those events are, indeed, impossible, as you believe. Consider, for example, a simple package for managing a street light. The specification might look as follows:

```
pragma SPARK_Mode (On);
package Lights is
   type Color_Type is (Red, Yellow, Green);

   function Next_Color (Current_Color : in Color_Type) return Color_Type;

end Lights;
```

Function Next_Color takes a color and returns the next color to be used in a specific sequence. An enumeration type defines the three possible color values. The body of this package is shown as follows:

```
pragma SPARK_Mode(On);
package body Lights is

   function Next_Color (Current_Color : in Color_Type) return Color_Type is
      Result : Color_Type;
   begin
      case Current_Color is
         when Red =>
            Result := Green;
         when Yellow =>
            Result := Red;
         when Green =>
            Result := Yellow;
      end case;
      return Result;
   end Next_Color;

end Lights;
```

Function Next_Color uses a **case** statement to handle each of the possible colors. Ada's full coverage rules for case statements, described in Section 2.1.2,

ensure that every color is considered. Suppose now that a new color is added to the set of possible colors so the definition of Color_Type becomes

```
type Color_Type is (Red, Yellow, Green, Pink);
```

The Ada compiler will require a case for Pink be added to the case statement in Next_Color. Suppose, however, that you believe the logic of your program is such that Next_Color will never be called with a Pink value. To satisfy the compiler you might include an empty case such as

```
when Pink =>
   null;
```

This executes the special *null statement* in that case, which does nothing. However, if calling Next_Color with a Pink value is a logical error, this approach just hides the error and is thus highly undesirable.

Another strategy would be to explicitly raise Program_Error as a way of announcing the error in terms that cannot be easily swept under the rug:

```
when Pink =>
   raise Program_Error;
```

Now the SPARK tools will attempt to prove this case can never happen, thus verifying what you believe to be true about your program. However, with Next_Color written as we have shown so far, that proof would fail. Using a precondition makes the proof trivial:

```
function Next_Color (Current_Color : in Color_Type) return Color_Type
  with Pre => Current_Color /= Pink;
```

Of course, now the SPARK tools will try to prove that this precondition is satisfied at each call site.

Tasking_Error

Ada has extensive support for writing programs built from multiple, interacting tasks. Certain error conditions can arise in connection with these tasking features, and in that case Tasking_Error is raised. The version of SPARK 2014 available at the time of this writing does not support any of Ada's tasking features and, thus, avoids Tasking_Error entirely.[3] However, we note that because SPARK allows you to build programs from a mixture of SPARK and full Ada, as we discuss in Section 7.1, it is possible to use the SPARK tools to analyze the sequential portions of a larger concurrent program.

Storage_Error

In an Ada program, Storage_Error is raised if a program runs out of memory. There are two primary ways that can happen. First, the program might run out

of heap space while objects are dynamically allocated. Second, the program might run out of stack space to hold local variables while subprograms are called.

SPARK avoids Storage_Error from heap exhaustion by disallowing heap allocated memory. Unfortunately, avoiding stack overflow is more difficult, and SPARK by itself cannot guarantee that a program will not raise Storage_Error.

However, an analysis can be done to ensure a program will not run out of stack space. The approach requires two steps. First, it is necessary to forbid all forms of recursion, both direct where a subprogram calls itself and indirect where two (or more) subprograms call each other. It is very difficult to put a static bound on the amount of stack space required by a recursive program because the number of recursive invocations is, in general, only discovered dynamically. The SPARK tools do not detect recursion, but additional tools such as AdaCore's GNATcheck can do so.

If a program is not recursive, and if it does not make any indirect calls (SPARK forbids such calls), it is possible to build a tree representing the subprogram calls it might make. Such a tree is referred to as a *call tree*. For example, if procedure A calls procedures B1 and B2, then the tree would have B1 and B2 as children of A.

With a complete call tree in hand, one must then compute the stack space required on each path from the tree's overall root, representing the main procedure, to each leaf. Some of these paths may not be possible executions and could potentially be ruled out by a static analysis. Some of these paths may contain dynamically sized local variables, and an upper bound on the size of those variables would need to be statically determined. Although the SPARK tools do not do this analysis, GNATstack from AdaCore or StackAnalyzer from AbsInt are two tools available at the time of this writing that do.

Alternatively, stack usage can be dynamically analyzed during testing. It may then be possible to obtain a worst case upper bound on the required stack size via reasoning about the test cases. However the bound is obtained, stack overflow can be avoided by providing the program with stack space that equals or exceeds that bound.

Technically, the Ada standard allows the compiler to raise Storage_Error during the execution of potentially any construct at all. Indeed some compilers do make implicit use of additional storage in a manner that is not visible to the programmer. Thus, completely ruling out Storage_Error also requires tools that are aware of the compiler's implementation strategies.

Constraint_Error

We have seen how SPARK manages Program_Error, Tasking_Error, and Storage_Error, making it possible to write programs that will not raise those exceptions.

The last exception predefined by Ada is Constraint_Error. This exception is raised whenever a constraint is violated, such as when (a) a value is assigned to a variable that violates the range constraint of that variable's type; (b) an array element is accessed with an index that is out of bounds for that array; or (c) overflow occurs during arithmetic operations.

Certain sources of Constraint_Error in full Ada cannot arise in SPARK because of limitations in the SPARK language. For example, Ada raises Constraint_Error if an attempt is made to dereference a null access value (pointer). That is not possible in SPARK because SPARK forbids access values. However, many sources of Constraint_Error are possible in SPARK. Guaranteeing that Constraint_Error cannot arise requires constructing proofs based on the structure of the code being analyzed.

6.1.2 *Verification Conditions*

At each point in your program where a language-mandated check must occur, the SPARK tools generate a *verification condition*, also called a *proof obligation* in some literature. Each verification condition is a logical implication as described in Section 5.3. The hypotheses of the verification condition are taken from the the code leading to that program point with every possible path to the program point being considered. The conclusion of the verification condition is, in essence, that the exception will not be raised. That is, the conclusion states the condition that would cause the exception is false. If the verification condition is proved, then the hypotheses imply the exception will never occur.

In a typical program, a large number of verification conditions are generated, some with complex hypotheses. The SPARK tools make use of automatic theorem provers to discharge the verification conditions without human intervention. At least that is the idea. Usually, it is necessary to help the tools by organizing the program in a particular way or by providing additional information to the tools. We discuss these techniques in the context of several examples in the following sections of this chapter. Once we have discharged all of the verification conditions, we have great confidence that our program contains no runtime errors.

6.2 Contracts

Some of the most significant additions to Ada 2012 relative to earlier versions of the language are the facilities supporting *contract based programming*. This is a style of programming in which the semantics of subprograms and the

Table 6.1. Ada/SPARK assertions

Aspect/Pragma	Supported in SPARK	SPARK only	Cross reference
Assert	Yes	No	Section 6.3.1
Assert_And_Cut	Yes	Yes	Section 6.3.2
Assume	Yes	Yes	Section 6.3.3
Contract_Cases	Yes	Yes	Section 6.2.6
Dynamic_Predicate	No	No	Section 6.2.5
Initial_Condition	Yes	Yes	Section 6.2.2
Loop_Invariant	Yes	Yes	Section 6.4
Loop_Variant	Yes	Yes	Section 6.5
Post	Yes	No	Section 6.2.2
Pre	Yes	No	Section 6.2.1
Refined_Post	Yes	Yes	Section 6.2.3
Static_Predicate	Yes	No	Section 6.2.5
Type_Invariant	No	No	Section 6.2.4

properties of types are formally specified in the program itself by way of assertions written by the software designer. We use the term *assertion* to refer to a specific condition encoded into the program and reserve the term *contracts* to refer to the abstract concept of using assertions to formalize a program's behavior.

In Ada, assertions created during program design are executable. During runtime they can be checked to ensure that the behaviors specified by the designers are being realized. The SPARK tools go beyond this dynamic checking and endeavor to statically prove that none of the assertions will ever fail.

Table 6.1 shows all the assertion forms available to an Ada/SPARK programmer with references to where we discuss each form in more detail. At the time of this writing not all assertion forms that exist in Ada are yet supported by SPARK. Unsupported assertions are indicated as such in the table. Furthermore, SPARK adds some assertion forms that are not part of standard Ada. Those that are SPARK specific are also indicated in the table.

Whenever a program unit is compiled, some *assertion policy* is in effect. The Ada standard defines only two such policies, Check and Ignore, although compilers are allowed to provide additional policies.

If the assertion policy is Check, then any assertion that fails (evaluates to false) causes an Assertion_Error exception from the Ada.Assertions package to be raised. It is important to understand that this check occurs at runtime and recovery must be done at runtime, for example, by way of a suitable exception handler. In this respect, assertions are similar to the runtime errors described

in Section 6.1. However, unlike runtime errors, it is up to the programmers to explicitly include assertions in their programs. Furthermore, assertions can be used to check whatever conditions the programmers deem appropriate.

An assertion policy of Ignore causes the compiler to remove the assertion checks so they will not be tested at runtime. In that case, failures of the assertions are not detected, probably causing other, less well-behaved failures. One of the main goals of SPARK is to allow proof that all assertions never fail.[4] In that case, the program can be compiled with an assertion policy of Ignore without being concerned about unexpected failures. In addition to increasing the reliability of the program, SPARK can improve its performance because the assertions consume resources, such as processor time, to check.

In the following subsections we describe how to specify assertions and give some hints about their use. Our approach is to show examples of increasing complexity and realism to give a feeling for how SPARK assertions work in practice.

6.2.1 *Preconditions*

Ada allows subprograms to be decorated with preconditions. A *precondition* is a logical expression (an expression with Boolean type) that must hold (evaluate to True) whenever the subprogram is called. Preconditions are specified using the Pre aspect. Because the expression used to define the precondition is arbitrary, it can be used to encode conditions of any complexity.

As an example, consider a package, Shapes, that does some geometric computations on two-dimensional shapes. All coordinates used by the package are in Cartesian form and constrained to a workspace in the range of −100 to 100 pixels in each dimension. An abbreviated specification of the package follows:

```
pragma SPARK_Mode(On);
package Shapes is

   subtype Coordinate_Type is Integer range −100 .. +100;
   subtype Radius_Type     is Coordinate_Type range 0 .. 10;

   type Circle is
      record
         Center_X : Coordinate_Type;
         Center_Y : Coordinate_Type;
         Radius   : Radius_Type;
      end record;
```

```
   -- Return True if X, Y are inside circle C.
   function Inside_Circle (X, Y : in Coordinate_Type;
                           C    : in Circle ) return Boolean
     with
       Pre => C.Center_X + C.Radius in Coordinate_Type and
              C.Center_X - C.Radius in Coordinate_Type and
              C.Center_Y + C.Radius in Coordinate_Type and
              C.Center_Y - C.Radius in Coordinate_Type;
end Shapes;
```

Here, a circle is described by the coordinates of its center and a radius. The function Inside_Circle takes a pair of X, Y coordinates and a circle and returns true if the given coordinates are inside the given circle.

The precondition given on Inside_Circle enforces the rule that the circle must be entirely contained in the workspace. We do not want to consider circles that overlap the allowed range of coordinates such as a circle with a center near the boundaries of Coordinate_Type and with a large radius.

Because the precondition is part of the subprogram's declaration, it is known to all users of the subprogram and becomes part of the subprogram's interface. It is the caller's responsibility to ensure that the precondition is satisfied. If it is not (and if the assertion policy is Check), then an exception will be raised at runtime. However, the SPARK tools will generate a verification condition at each location where Inside_Circle is called that, if proved, shows the precondition is satisfied at each of those places.

Notice that ordinary Ada subtype constraints are a kind of precondition. Consider, for example, a function Fibonacci that computes Fibonacci numbers.[5] If it is declared as

```
function Fibonacci (N : in Natural) return Natural;
```

it will raise Constraint_Error even for fairly small values of N. This is because the Fibonacci sequence grows very rapidly and causes an arithmetic overflow to occur inside the function if N is even moderately large. In particular, for systems using 32 bits for type Integer, the largest Fibonacci number that can be calculated is for $N = 46$. One could express this constraint by adding a precondition:

```
function Fibonacci (N : in Natural) return Natural
  with
    Pre => N <= 46;
```

However, it is more appropriate to use Ada's facilities for creating scalar subtypes instead of a precondition.

```
subtype Fibonacci_Argument_Type is Natural range 0 .. 46;

function Fibonacci (N : in Fibonacci_Argument_Type) return Natural;
```

Now an attempt to use a value out of range will cause Constraint_Error rather than Assertion_Error as a failed precondition would do, but the SPARK tools will attempt to prove that the error cannot occur in either case.

Using subtypes when possible is best because, being simpler, they are easier for the compiler to manage and optimize. They will also continue being checked even if the assertion policy is Ignore. However, Ada's scalar subtypes are relatively limited because they can only express certain restricted kinds of constraints such as range constraints. In contrast, preconditions are entirely general and can be used to express conditions of arbitrary complexity. In the example of Inside_Circle , scalar subtypes cannot capture the desired condition. A small circle right at the edge of the coordinate system might still be acceptable. Trying, for example, to constrain the type used to represent the coordinates of the circle's center is not a solution. The acceptability of a circle depends on the interaction between the values of its components.

In Section 5.4 we introduced Ada's syntax for quantified expressions using **for all** or **for some**. Preconditions, and assertions in general, can often make good use of quantified expressions particularly when arrays are involved. In fact, it is primarily their use in assertions that motivated the addition of quantified expressions to the Ada language.

As an example, consider a procedure that implements the binary search algorithm. This algorithm takes an array and a data item and efficiently checks to see if the array contains the data item. However, the algorithm requires that it be given an array in sorted order. The following specification of package Searchers contains the declaration of a procedure Binary_Search[6] along with a precondition:

```
package Searchers
  with SPARK_Mode => On
is
   subtype Index_Type is Positive range 1 .. 100;
   type Array_Type is array(Index_Type) of Integer;

   procedure Binary_Search (Search_Item : in  Integer;
                            Items       : in  Array_Type;
                            Found       : out Boolean;
                            Result      : out Index_Type)
      with
         Pre =>
```

```
            (for all J in Items'Range =>
               (for all K in J + 1 .. Items'Last => Items(J) <= Items(K)));
end Searchers;
```

The precondition uses two nested for-all quantified expressions to assert that each item of the array comes before (or is the same as) all the items that follow it. The SPARK tools will, as usual, create a verification condition at each place where Binary_Search is called to prove that the precondition is satisfied.

It bears repeating that assertions in Ada are executable. With that in mind, notice that the precondition given for Binary_Search runs in $O(n^2)$ time. In contrast, the binary search algorithm itself runs in only $O(\log\ n)$ time.

An alternative way of expressing that the array is sorted is

```
Pre => (for all J in Items' First .. Items'Last - 1 =>
          Items (J) <= Items (J+1))
```

This has the advantage of running in only $O(n)$ time, and it might also be considered clearer. However, it is still asymptotically slower than the algorithm to which it is attached. Because assertions can use quantified expressions over large arrays, and even call recursive functions, they can consume large amounts of space and time. Expressive assertions thus have the potential to render unusable a program that would otherwise have acceptable performance characteristics. We discuss this issue further in Section 6.8.

6.2.2 *Postconditions*

A *postcondition* is a condition that is asserted to be true when a subprogram completes its actions and returns control to the caller. It describes the effects of the subprogram. Postconditions are, thus, formal statements derived from the functional requirements that our program must meet.

Postconditions are introduced with the Post aspect. As an example, consider the Searchers package again. A version of that package follows, where the Binary_Search procedure has been given a postcondition:

```
package Searchers2
  with SPARK_Mode => On
is
   subtype Index_Type is Positive range 1 .. 100;
   type Array_Type is array(Index_Type) of Integer;

   procedure Binary_Search (Search_Item : in  Integer;
                            Items       : in  Array_Type;
                            Found       : out Boolean;
                            Result      : out Index_Type)
```

```
with
  Pre =>
     (for all J in Items'Range =>
        (for all K in J + 1 .. Items'Last => Items(J) <= Items(K))),
  Post =>
     (if Found then Search_Item = Items (Result)
                else (for all J in Items'Range => Search_Item /= Items(J)));

end Searchers2;
```

The postcondition says that if the procedure reports it has found the search item, then that item exists at the reported position in the array. The else clause says that if the procedure does not find the search item, the item does not exist at any location in the array.

Preconditions and postconditions have a dual relationship. A precondition is an obligation on the caller to show, either by runtime testing or proof, that the condition is true before calling a subprogram. Inside the subprogram the precondition can be used in the hypotheses of verification conditions, being taken as a given in the context of the subprogram's body.

Postconditions, on the other hand, are obligations on the subprogram itself to show that the condition is true when the subprogram returns. The calling context can use the postcondition in the hypotheses of verification conditions that appear past the point of the call.

Callers are interested in weak preconditions that are easy to prove but strong postconditions that provide a lot of information they can use after the call. In contrast, implementers want strong preconditions that provide a lot of information in the subprograms being implemented but weak postconditions that are easy to prove. Both sides of the call want to make as few promises as possible but get as many promises as they can.

Of course in real programs, just as in real life, a balance must be struck. Postconditions describe what it means for a subprogram to be correct and thus would ideally be written as part of the subprogram's design. The more specific (stronger) a postcondition is, the more information about the subprogram's behavior it captures. Preconditions often need to be provided to support the postcondition.

For example, in the case of Binary_Search, the implementation has no chance of proving the postcondition unless it "knows" the array is already sorted. The algorithm depends on that. Thus, the precondition is necessary if the postcondition is to be proved.

The Binary_Search procedure has only **in** and **out** parameters. In the case of **in out** parameters (or In_Out global items), it is sometimes necessary to

reference the original value of the parameter (or global item) in the postcondition. As an example, consider a procedure that finds the smallest prime factor of a natural number and returns both the factor and the original number after the factor has been divided out:

```
procedure Smallest_Factor (N      : in out Positive;
                           Factor :    out Positive)
   with
      Post => Is_Prime(Factor) and
              (N = N'Old / Factor) and
              (N'Old rem Factor = 0);
```

Here we make use of a function Is_Prime that returns true if and only if its argument is a prime number.

The procedure changes the value of N. However, we can reference its original value in the postcondition using the 'Old attribute. This implies that the compiler must maintain a copy of the value of N before the procedure is called so it can use that copy when the postcondition is evaluated. Here, N is just an integer so keeping a copy of it is not expensive. Yet this is another example of how assertions can potentially consume significant resources; consider the case when the parameter is a large data structure.

In general, we can talk about the *prestate* of a subprogram as the state of the entire program just before the subprogram begins executing. Similarly, we can talk about the *poststate* of a subprogram as the state of the entire program after the subprogram returns. In these terms, the 'Old attribute can be said to make a copy of a part of the prestate for use in the evaluation of the postcondition.

Care is needed when using 'Old with arrays. Consider the following four expressions. Here, A is an array and Index is a variable of the array's index subtype.

- A'Old(Index) accesses the original array element at the position given by the current Index. Here *original* means part of the prestate, and *current* means the value when the postcondition is executing – that is, part of the poststate.
- A(Index'Old) accesses the current array element at the position given by the original Index.
- A'Old(Index'Old) accesses the original array element at the position given by the original Index. Both the original array (in its entirety) and the original Index are saved when the subprogram is entered.
- A(Index)'Old is largely the same as A'Old(Index'Old). In particular, it refers to the original value of the expression A(Index). However, only the original value of A(Index), not the entire array, is saved when the subprogram is entered.

The last case illustrates a general rule. The prefix of 'Old can be an arbitrary expression, the value of which is saved when the subprogram is entered. For example, (X + Y)'Old in a postcondition causes the original value of the expression X + Y to be saved and used when evaluating the postcondition. Each usage of 'Old in a postcondition implies the creation of a separate saved value. We also note that because the expression used as a prefix to 'Old is copied, it cannot have a limited type.[7]

The postcondition of Smallest_Factor shown earlier does not fully describe the intended behavior of the procedure. It only says that Factor is some prime factor of the original value of N, but not necessarily the smallest one. Although the postcondition is not as strong as it could be, it still conveys useful information into the calling context. Proving that the postcondition will always be satisfied is a partial proof of the correctness of the procedure. The remaining properties could be explored with testing. We describe the interplay between proof and testing in more detail in Section 8.4.

The postcondition can be strengthened as follows:

```
procedure Smallest_Factor (N      : in out Positive;
                           Factor : out Positive)
   with
      Post => (N = N'Old / Factor) and
              (N'Old rem Factor = 0) and
              (for all J in 2 .. Factor - 1 => N'Old rem J /= 0);
```

The additional quantified expression says there are no other factors of the original N that are smaller than Factor. It is no longer necessary to directly assert that Factor is prime because it is a mathematical fact that the smallest factor of a number will always be prime. Notice that now the postcondition relys on a mathematical property that is proved outside the scope of the program. This is an example of the interplay between design and implementation. Although SPARK allows you to formalize the construction of your software, it is not by itself a complete formal design methodology.

Function results are normally anonymous, so to reference them in a postcondition for a function, it is necessary to use the 'Result attribute. Consider a function version of Smallest_Factor that returns the smallest factor of a natural number:

```
function Smallest_Factor (N : in Positive) return Positive
 with
   Post => (N rem Smallest_Factor'Result = 0) and
           (for all J in 2 .. Smallest_Factor'Result - 1 => N rem J /= 0);
```

It is not necessary to use 'Old here because in this case N is an **in** parameter that the function cannot change.

Package Initial Conditions

As we describe in Section 3.4.1, when a package is elaborated, certain initialization activities can occur. Global variables in the package specification or body that have initialization expressions are given their initial values. Also, the package body can have a sequence of statements used to perform more complex package-wide initializations. The Initial_Condition aspect can be used on a package specification to assert a condition that is true after the package has been fully elaborated. Conceptually the aspect is like a kind of package-wide postcondition.

Typically, Initial_Condition makes sense in cases where a variable package has an abstract state that it initializes, as described in Section 4.3. The initial condition can then capture information about the result of that initialization. The SPARK tools generate a verification condition to show that the package's elaboration does indeed initialize the internal state as specified.

As an example, consider again the Datebook package discussed in Section 4.3. Here we show an abbreviated specification that includes a function returning the number of events in the datebook. For brevity, the other subprograms in the package are not shown.

```
with Dates;
package Datebook
   with
      Abstract_State      => State,
       Initializes        => State,
       Initial_Condition  => Number_Of_Events = 0
is
   function Number_Of_Events return Natural
      with
         Global => (Input => State);
end Datebook;
```

If the Initial_Condition aspect is used, it must appear after the Abstract_State and Initializes aspects, if they are present. It can also only use visible variables and subprograms. In the preceding example, the package initial condition asserts that after elaboration, the number of events in the datebook is zero. This is an intuitive expectation, now formally specified and checked by the SPARK tools.

Notice that function Number_Of_Events is declared after it is used in the Initial_Condition aspect of package Datebook. This order violates the scope

rules defined on page 30. Our Datebook example shows one of the few places where Ada allows use of as yet undeclared entities – assertions can reference names that are declared later in the same declarative part. We will make frequent use of this exception to Ada's scope rules.

6.2.3 *Private Information*

Consider the type package Shapes on page 164. In many applications it would be more appropriate to keep the representation of circles hidden by making type Circle private. However, by moving the details of Circle into the private section, we find another problem – the precondition for function Inside_Circle no longer compiles. The precondition of a public subprogram cannot make use of hidden (private) information.

To work around this issue, we can introduce a public function, In_Bounds, that tests if a Circle is entirely inside the workspace. The following package Shapes2 shows these changes:

```
pragma SPARK_Mode(On);
package Shapes2 is
   subtype Coordinate_Type is Integer range −100 .. +100;
   subtype Radius_Type is Coordinate_Type range 0 .. 10;
   type Circle is private;

   −− Return True if X, Y are inside circle C.
   function Inside_Circle (X, Y : in Coordinate_Type;
                           C    : in Circle) return Boolean
      with Pre => In_Bounds (C);

   −− Return True if C is entirely in the workspace.
   function In_Bounds (C : in Circle) return Boolean;

private
   type Circle is
      record
         Center_X : Coordinate_Type;
         Center_Y : Coordinate_Type;
         Radius   : Radius_Type;
      end record;
end Shapes2;
```

The precondition on Inside_Circle has been rewritten to make use of the new function. The body of function In_Bounds is written in the package body

and has access to the package's private information. Thus, it can make use of the representation of type Circle as necessary. Defining functions for use in assertion expressions can also improve the readability of such expressions as they become complex, and it simplifies the sharing of elaborate conditions between multiple assertions. Notice that as mentioned in the previous section, we are using In_Bounds in an assertion before it has been declared.

The function In_Bounds is a perfectly ordinary function. It can be called by clients of the package like any other public function. In fact, it offers a useful service that might be of interest to package clients. It is also possible to create functions that can only be used in assertion expressions. We discuss these *ghost functions* in more detail in Section 9.1.1.

As you might expect, postconditions on subprograms in the visible part of a package are also forbidden from using private information. We can introduce functions to specify the effect of a subprogram in abstract terms just as we did with the precondition of the Inside_Circle function. For example, the Shapes2 package does not provide a way to create initialized Circle objects. We can remedy this problem by adding a suitable constructor function to the package:

```
function Make_Circle (X, Y : in Coordinate_Type;
                      R    : in Radius_Type) return Circle
   with
      Post => In_Bounds (Make_Circle'Result);
```

Here we assume Make_Circle forces the resulting circle to be in bounds by adjusting the radius if necessary without indicating an error. How desirable such behavior is in practice will depend on the design of the overall application.

The SPARK tools can reason that Circle objects returned from Make_Circle will be acceptable to Inside_Circle without knowing anything about what In_Bounds does. The tools can treat the function abstractly. This relies on In_Bounds having no side effects, a requirement of all functions in SPARK, nor reading any global variables as specified in its data dependency contract (synthesized in this case).

Subprograms in a package body have full access to the information in the private part of the package specification. Helper subprograms inside the body of package Shapes2 may need to see the effects of Make_Circle in terms of the private structure of type Circle. Just knowing that the circle objects returned by Make_Circle are "in bounds" is not enough. Internal subprograms may need to know something about the relationship between the circle's center coordinates and radius. Internal subprograms are allowed to have that knowledge, but how can they be given it?

SPARK allows you to refine a postcondition in the body of a package to express it in terms of the private information available in the body. An implementation of Make_Circle that refines the postcondition in the specification using the Refined_Post aspect follows:

```
function Make_Circle (X, Y    : in Coordinate_Type;
                      Radius : in Radius_Type) return Circle
  with Refined_Post =>
    (Make_Circle'Result.Center_X + Make_Circle'Result.Radius
       in Coordinate_Type and
     Make_Circle'Result.Center_X - Make_Circle'Result.Radius
       in Coordinate_Type and
     Make_Circle'Result.Center_Y + Make_Circle'Result.Radius
       in Coordinate_Type and
     Make_Circle'Result.Center_Y - Make_Circle'Result.Radius
       in Coordinate_Type)
is
   R : Radius_Type := Radius;
begin
   if R >= Coordinate_Type'Last - X then
      R := Coordinate_Type'Last - X;
   end if;
   if R >= X - Coordinate_Type'First then
      R := X - Coordinate_Type'First;
   end if;
   if R >= Coordinate_Type'Last - Y then
      R := Coordinate_Type'Last - Y;
   end if;
   if R >= Y - Coordinate_Type'First then
      R := Y - Coordinate_Type'First;
   end if;
   return (X, Y, R);
end Make_Circle;
```

The effect of Make_Circle is described in internal terms. The SPARK tools will generate verification conditions to show that the body of the subprogram honors the refined postcondition. In addition, the tools will generate a verification condition that shows the precondition (if any), together with the refined postcondition, implies the publicly visible postcondition. This allows the refined postcondition to be stronger than the public postcondition (it can say more), but not weaker.

Callers of Make_Circle inside the package body will use the refined postcondition as it is written in terms of the private information of interest to them. The

general rule is that if the refined aspects are visible, they are used. This includes the Refined_Global and Refined_Depends aspects mentioned in Section 4.3.1.

Of course, the refined postcondition cannot be used by clients of the package to help prove verification conditions in the calling context. This is a necessary restriction because the refined postcondition is written in private terms to which the clients have no access. In effect, the public postcondition sanitizes the refined postcondition to make it appropriate for clients. If the private information changes, the refined postcondition might have to be updated, but the public postcondition captures essential design information and need not (should not) be changed.

If a refined postcondition is not used, the public postcondition takes its place and represents the only information known about the effect of the subprogram, even to internal callers. If the public postcondition is not used, it is taken to be true, which is easily proved from any refined postcondition that might exist.

In the earlier implementation of Make_Circle, the SPARK tools cannot prove that the public postcondition follows from the refined postcondition. This is because the public postcondition is written in terms of function In_Bounds, and the effect In_Bounds has on the private components of Circle is not known to the SPARK tools. We need to add a refined postcondition to In_Bounds:

```
function In_Bounds (C : in Circle) return Boolean
  with
    Refined_Post => In_Bounds'Result =
      (C.Center_X + C.Radius in Coordinate_Type and
       C.Center_X - C.Radius in Coordinate_Type and
       C.Center_Y + C.Radius in Coordinate_Type and
       C.Center_Y - C.Radius in Coordinate_Type)
is
begin
   return
     (C.Center_X + C.Radius in Coordinate_Type and
      C.Center_X - C.Radius in Coordinate_Type and
      C.Center_Y + C.Radius in Coordinate_Type and
      C.Center_Y - C.Radius in Coordinate_Type);
end In_Bounds;
```

The refined postcondition asserts that the value returned by In_Bounds is the same as that given by its implementation. Of course this is extremely redundant. SPARK has a special rule that helps in cases like this. When an expression function is used (see Section 2.2.2), the body of the expression function automatically serves as its postcondition. In effect, a postcondition

is generated that asserts the expression function returns the same value as its implementation. Such a postcondition is trivially proved, yet this behavior means expression functions can be thought of as fragments of logic that have been factored out of the assertions. In effect, expression functions are pure specification. Thus, we can more easily write In_Bounds like this:

```
function In_Bounds (C : in Circle ) return Boolean is
  (C.Center_X + C.Radius in Coordinate_Type and
   C.Center_X - C.Radius in Coordinate_Type and
   C.Center_Y + C.Radius in Coordinate_Type and
   C.Center_Y - C.Radius in Coordinate_Type);
```

Expression functions are often fairly short and make excellent candidates for being inline expanded.[8] A common idiom is to declare the function in the visible part of the package using the Inline aspect and implement it as an expression function in the private part of the package specification. Here the specification of Shapes3 illustrates this approach:

```
pragma SPARK_Mode(On);
package Shapes3 is
   subtype Coordinate_Type is Integer range -100 .. +100;
   subtype Radius_Type is Coordinate_Type range 0 .. 10;
   type Circle is private;

   -- Create a circle object.
   function Make_Circle (X, Y   : in Coordinate_Type;
                         Radius : in Radius_Type) return Circle
      with
         Post => In_Bounds (Make_Circle'Result);

   -- Return True if X, Y are inside circle C.
   function Inside_Circle (X, Y : in Coordinate_Type;
                           C    : in Circle ) return Boolean
      with Pre => In_Bounds (C);

   -- Return True if C is entirely in the workspace.
   function In_Bounds (C : in Circle ) return Boolean
      with Inline => True;

private
   type Circle is
      record
         Center_X : Coordinate_Type;
         Center_Y : Coordinate_Type;
```

```
      Radius    : Radius_Type;
   end record;

function In_Bounds (C : Circle) return Boolean is
  (C.Center_X + C.Radius in Coordinate_Type and
   C.Center_X - C.Radius in Coordinate_Type and
   C.Center_Y + C.Radius in Coordinate_Type and
   C.Center_Y - C.Radius in Coordinate_Type);
end Shapes3;
```

The SPARK tools can now prove that Make_Circle satisfies its public postcondition because it "knows" what In_Bounds does. Furthermore, the refined postcondition is no longer needed on Make_Circle in this case because In_Bounds captures all the necessary information in a way that is also usable to internal subprograms. Thus we have come full circle and can remove the Refined_Post aspect on Make_Circle as well.

The moral of this story is, try to implement functions used in public assertions as expression functions in the private section of a package's specification or in the package's body.

We finish this section by noting that private information may also include the internal state of a package such as any global data that it contains. Much of the previous discussion applies equally to the case when a public assertion needs to access such internal state. However, in that case the implementation of any functions used in the assertion would need to be in the package body where the package's internal state can be accessed; being in the private part of the specification would not be enough.

6.2.4 *Type Invariants*[9]

So far we have discussed assertions that are attached to subprograms and that describe conditions associated with calling subprograms or with the values returned by subprograms. However, Ada also allows you to attach assertions to the types themselves. Such assertions describe properties of all objects of those types. In this section we describe type invariants and give some hints about how SPARK may support them in the future.

In package Shapes3 in the previous section, a precondition was used to ensure that Circle objects are sensible before passing them to function Inside_ Circle . If the package provided many subprograms, each would presumably need similar preconditions for all Circle parameters.

Alternatively, one could apply postconditions on all subprograms to check that the Circle objects returned by them are sensible. This was done with the

Make_Circle function. Because the Circle type is private, it is not necessary to do both. Circles can only be changed by the subprograms in the package. If all subprograms that return circles return only valid circles, the subprograms that accept circles can just assume they are valid.

In any case, the assertion we are trying to apply – that all circles are entirely inside the workspace – is really a restriction on the type Circle and not a restriction associated with the procedures that manipulate circles. Ada provides a way to express this idea more directly using a *type invariant*. The following specification of package Shapes4 illustrates the approach:

```
pragma SPARK_Mode(On);
package Shapes4 is
   subtype Coordinate_Type is Integer range −100 .. +100;
   subtype Radius_Type is Coordinate_Type range 0 .. 10;
   type Circle is private
      with
         Type_Invariant => In_Bounds (Circle);

   -- Create a circle object.
   function Make_Circle (X, Y   : in Coordinate_Type;
                         Radius : in Radius_Type) return Circle;

   -- Return True if X, Y are inside circle C.
   function Inside_Circle (X, Y : in Coordinate_Type;
                           C    : in Circle) return Boolean;

   -- Return True if C is entirely in the workspace.
   function In_Bounds (C : in Circle) return Boolean
      with Inline;

private
   type Circle is
      record
         Center_X : Coordinate_Type;
         Center_Y : Coordinate_Type;
         Radius   : Radius_Type;
      end record;

   function In_Bounds (C : in Circle) return Boolean is
     (C.Center_X + C.Radius in Coordinate_Type and
      C.Center_X − C.Radius in Coordinate_Type and
      C.Center_Y + C.Radius in Coordinate_Type and
      C.Center_Y − C.Radius in Coordinate_Type);
end Shapes4;
```

The only change relative to the earlier Shapes3 package is that the condition on the circle being in the workspace has been moved from being a precondition of Inside_Circle and a postcondition of Make_Circle to being an invariant of the Circle private type. Notice that in the expression used for the type invariant the name of the type itself, Circle, is used as a stand-in for the object of that type being checked.

Type invariants can only be applied to private types. The condition they assert is only enforced at the "boundary" of the package that implements the type. Inside that package, objects may go through intermediate states where the invariant is temporarily false. However, the invariant is checked whenever a public subprogram returns to ensure that objects seen by the clients of the package are always in a proper state. In this respect, type invariants are somewhat like postconditions that are automatically applied to all public subprograms. Because SPARK does not currently support type invariants, their effect could be simulated, in large measure, by tediously defining appropriate postconditions.

Package Shapes4 as currently defined provides no default initialization for a Circle object. Merely declaring a Circle may cause the type invariant to fail as type invariants are also checked after default initialization and the initial values of the components of a Circle are indeterminate.

Sensible default initialization can be specified by simply adding appropriate initializers to the components of the record defining Circle:

```
type Circle is
   record
      Center_X : Coordinate_Type := 0.0;
      Center_Y : Coordinate_Type := 0.0;
      Radius   : Radius_Type := 0.0;
   end record;
```

A default initialized Circle will now obey its invariant.

Once SPARK supports type invariants, it will generate verification conditions at each place where an invariant check is needed that, if proved, will show that the check cannot fail.

6.2.5 *Subtype Predicates*

In addition to type invariants, Ada also allows assertions to be applied to nonprivate types in the form of *subtype predicates*. In some ways, subtype predicates are similar to constraints, such as range constraints, that limit the allowed values of a subtype. They are checked in similar places. However, it is natural to describe subtype predicates as assertions because, like other

kinds of assertions, they are conditions of arbitrary complexity provided by the programmer and are under the control of the the assertion policy.

Dynamic Predicates[10]

A type can be considered a set of values (a domain) and a set of operations that can be applied to those values. Ada's **subtype** declaration creates a subtype by specifying a subset of the domain of the base type. Consider, for example,

```
subtype Pixel_Coordinate_Type is Natural range 0 .. 1023;
```

Instead of being the entire set of values associated with Natural, Pixel_Coordinate_Type is associated with a subset of those values in the range from 0 to 1023.

While this facility is useful, it is also quite limited. To define a more complex subset requires a more general method of specification. Dynamic predicates allow you to define which values are in the subset by using an arbitrary condition, for example,

```
subtype Even_Type is Natural
  with Dynamic_Predicate => Even_Type mod 2 = 0;
```

As with the Type_Invariant aspect, when the name of the type appears in the condition, it is interpreted as a stand-in for the value being tested. If the condition is true the value is in the subtype being defined. In this example Even_Type has values that are even natural numbers.

As another example consider the following:

```
subtype Prime_Type is Natural
  with Dynamic_Predicate => Is_Prime (Prime_Type);
```

Here the values of Prime_Type are the natural numbers for which function Is_Prime returns true – presumably prime numbers.

The precise locations where dynamic predicates are checked is given in section 3.2.4 of the *Ada Reference Manual* (2012), but intuitively they are checked in the same places where the simpler constraints are checked: during assignment to an object, when passing values as parameters to subprograms, and so forth. For example, if E is of type Even_Type, an expression such as

```
E := (A + B) / 2;
```

raises Assertion_Error when the expression (A + B) / 2 results in an odd number. Notice that although the dynamic predicate is like a kind of user-defined constraint, the Constraint_Error exception is not used if the predicate fails. The checking of dynamic predicates is controlled by the assertion policy just as with other kinds of assertions.

When SPARK does support dynamic predicates, it will likely still impose some restrictions on their use as compared to full Ada. For example, consider the following type definition:

```
type Lower_Half is
   record
      X : Natural;
      Y : Natural;
   end record
with Dynamic_Predicate => Lower_Half.X > Lower_Half.Y;
```

Instances of the Lower_Half type represent points in the first quadrant that are below the line $y = x$. However, in full Ada the dynamic predicate is not checked when individual components are modified. Thus, if Point were a variable of type Lower_Half, the program could set Point.X := 0 without causing Assertion_Error to be raised. It is likely SPARK will close that loophole by forbidding dynamic predicates that depend on the components of a composite type such as in this example.

Dynamic predicates that depend on global variables, for example, by calling a function F that reads such a variable, also create problems. Consider, for example, the following dynamic predicate:

```
subtype Example_Type is Natural
  with Dynamic_Predicate => F (Example_Type);
                                -- function F reads some global  variable
```

Because the global variable might change during the life of the program, a value of type Example_Type might sometimes be valid (in the subtype) and sometimes not, even if the value itself does not change. It is likely SPARK will forbid examples such as this as well.

Static Predicates

Dynamic predicates are very general, but there is a price to be paid for their generality. It is not reasonable (or normally even possible) for the compiler to compute the membership of a subtype defined with a dynamic predicate. As a result, subtypes defined with dynamic predicates cannot be used in certain areas where subtypes are allowed.

As an example, consider the following case statement in which the selector, N, has type Natural:

```
case N is
  when Even_Type => ...
  when Prime_Type => ...
end;
```

Ada's full coverage rules require that every possible value of N be accounted for in the various **when** clauses. In this example that is not the case since because there are natural numbers that are neither even nor prime. However, the compiler cannot be expected to know this without understanding the detailed semantics of the predicates used to define the subtypes. Those predicates might involve calling functions of significant complexity, such as is the case for Prime_Type in this example. Thus, the example is ruled out because it uses subtypes defined with dynamic predicates.

Ada defines a more restricted form of subtype predicate, called a *static predicate*, that does support many (although not all) of the usual features of subtypes while still allowing some degree of customization. Furthermore, at the time of this writing, SPARK supports static predicates.

The precise rules for the kinds of predicate specifications that can be used as static predicates is given in section 3.2.4 of the *Ada Reference Manual* (2012). However, as one example, we show a static predicate using a membership test to specify a non-contiguous range of values. Consider a package for managing a game of Scrabble. The specification of the package might be, in part, shown as follows:

```
pragma SPARK_Mode(On);
package Scrabble is

   subtype Scrabble_Letter is Character range 'A' .. 'Z';

   subtype Scrabble_Value is Positive
      with Static_Predicate => Scrabble_Value in 1 .. 5 | 8 | 10;

   type Scrabble_Word is array( Positive range <>) of Scrabble_Letter;

   subtype Scrabble_Score is Natural range 0 .. 100;
   function Raw_Score (Word : in Scrabble_Word) return Scrabble_Score
      with Pre => (Word'Length <= 10);

end Scrabble;
```

Here a subtype Scrabble_Letter is used to constrain Character to just the uppercase letters used by the game. The subtype Scrabble_Value is defined with a static predicate to only contain the values that are actually used on the various letters. The type Scrabble_Word is an unconstrained array of letters intended to hold a single word. The function Raw_Score adds together the value of the letters in the given word and returns it as a value in the range from 0 to 100. The

precondition on Raw_Score ensures that words of no longer than ten characters are used. This is the justification for limiting the return value to 100 (the maximum value of a letter is 10). Notice that in this case the postcondition is stated using a subtype.

The type Scrabble_Word cannot easily be made into a fixed size of an array of ten Scrabble_Letter because there is no character, such as a space, in Scrabble_Letter to use as padding needed for short words. You might be tempted to define Scrabble_Letter using a static predicate like this:

```
subtype Scrabble_Letter is Character
  with Static_Predicate => Scrabble_Letter in 'A' .. 'Z' | ' ';
```

However, the body of the package uses Scrabble_Letter as an index subtype for an array, and subtypes with predicates can never be used in that situation. Here is the body of package Scrabble:

```
pragma SPARK_Mode(On);

package body Scrabble is

   type Scrabble_Value_Lookup is array (Scrabble_Letter) of Scrabble_Value;
   Lookup_Table : constant Scrabble_Value_Lookup :=
     ('A' => 1, 'B' => 3, 'C' => 3, 'D' => 2,
      'E' => 1, 'F' => 4, 'G' => 2, 'H' => 4,
      'I' => 1, 'J' => 8, 'K' => 5, 'L' => 1,
      'M' => 3, 'N' => 1, 'O' => 1, 'P' => 3,
      'Q' => 10, 'R' => 1, 'S' => 1, 'T' => 1,
      'U' => 1, 'V' => 4, 'W' => 4, 'X' => 8,
      'Y' => 4, 'Z' => 10);

   function Raw_Score (Word : in Scrabble_Word) return Scrabble_Score is
      Total_Score : Scrabble_Score := 0;
   begin
      for Letter_Index in Word'Range loop
         pragma Loop_Invariant (Total_Score <= 10*(Letter_Index - Word'First));
         Total_Score := Total_Score + Lookup_Table (Word (Letter_Index));
      end loop;
      return Total_Score;
   end Raw_Score;

end Scrabble;
```

A lookup table is defined to translate Scrabble letters into their corresponding values. It is declared as a constant with a static expression as an initializer so

that even though it is read by function Raw_Score, it is not necessary to declare it as global input.

The SPARK tools are able to work with the subtype defined with a static predicate and prove this function obeys its postcondition. The Loop_Invariant pragma is used to help the tools handle the loop. Loop invariants are discussed in detail in Section 6.4.

6.2.6 *Contract Cases*

It is common for preconditions to divide the input space of a subprogram into several equivalence classes (disjoint subdomains) where each class has its own postconditions. Although we can use Pre and Post aspects to handle contracts of arbitrary complexity, SPARK provides the Contract_Cases aspect to simplify writing contracts for subprograms that divide their input space into a substantial number of different equivalence classes.

As a simple example, consider a function Sign that takes an arbitrary integer and returns -1 if the integer is negative, 0 if the integer is zero, and $+1$ if the integer is positive. Such a function might be declared and contract specified as follows:

```
subtype Sign_Type is Integer range -1 .. 1;

function Sign (X : in Integer) return Sign_Type
   with
      Contract_Cases =>
         (X < 0 => Sign'Result = -1,
          X = 0 => Sign'Result = 0,
          X > 0 => Sign'Result = 1);
```

This example shows three contract cases. Each case consists of a Boolean condition intended to be checked when the subprogram is called followed by a Boolean consequent that is checked when the subprogram returns. In effect, each contract case is like a mini-precondition followed by a corresponding postcondition.

When the subprogram is called, all of the conditions are checked and exactly one must be true. When the subprogram returns the consequent associated with the condition that was true is checked. At runtime Assertion_Error is raised if

- none of the conditions are true at the time of the call,
- more than one of the conditions are true at the time of the call, or
- the consequent associated with the true condition is false when the subprogram returns.

The SPARK tools generate verification conditions to prove

- the conditions cover the entire input domain of the subprogram (and thus it will always be the case that one of them will be true),
- the conditions are mutually exclusive, and
- the subprogram always returns with the consequent true for each corresponding precondition.

In other words, the tools generate verification conditions to show that Assertion_Error will never be raised.

It is important that the contract cases divide the entire input domain into disjoint subdomains. To illustrate, consider a function In_Unit_Square as an extension of our Sign function. In_Unit_Square returns +1 if the given X, Y coordinates are inside a square centered on the origin with a side length of two. Otherwise the function returns 0 for points in the first and third quadrants and −1 for points in the second and fourth quadrants. The following specification of this function using Contract_Cases must ensure that each case is disjoint.

```
function In_Unit_Square (X, Y : in Integer) return Sign_Type
   with
     Contract_Cases =>
       (X >= 0 and Y >= 0 and not (X <= 1 and Y <= 1) =>
                 In_Unit_Square'Result = 0,
        X < 0 and Y >= 0 and not (X >= -1 and Y <= 1) =>
                 In_Unit_Square'Result = -1,
        X < 0 and Y < 0 and not (X >= -1 and Y >= -1) =>
                 In_Unit_Square'Result = 0,
        X >= 0 and Y < 0 and not (X <= 1 and Y >= -1) =>
                 In_Unit_Square'Result = -1,
        others => In_Unit_Square'Result = 1);
```

Here we use **others** to specify a case not handled by the other cases (the case where the given point is on the square). If **others** appears, it must be last. Because **others** is always true, it is trivial to show that at least one of the previous cases will always be available. Showing that the four cases, one for each quadrant, are really disjoint is less obvious, but the SPARK tools will take care of that.

The implementation of In_Unit_Circle could follow the structure of the contract cases, but it need not. Here is an implementation based on an **if** ... **elsif** ... chain:

```
function In_Unit_Square (X, Y : in Integer) return Sign_Type is
begin
   if X in -1 .. 1 and Y in -1 .. 1 then
```

```
      -- In the square.
      return 1;
   elsif X >= 0 and Y >= 0 then
      -- First quadrant.
      return 0;
   elsif X < 0 and Y >= 0 then
      -- Second quadrant.
      return -1;
   elsif X < 0 and Y < 0 then
      -- Third quadrant.
      return 0;
   else
      -- Fourth quadrant.
      return -1;
   end if;
end In_Unit_Square;
```

This implementation takes advantage of the fact that the first succeeding condition stops the comparisons. The conditions in the implementation are not mutually exclusive, but they do not need to be. The SPARK tools will generate verification conditions to show that this implementation does meet the contract cases provided in the specification. However, this example shows that as the number of parameters (and input global items) to the subprogram increases, the dimensionality of the space that must be partitioned over the contract cases increases as well.

Also notice that the consequent of a contract case is evaluated after the subprogram returns, and so it is permitted to use the 'Result and 'Old attributes there as in postconditions.

Finally we note that it is permitted to use the normal Pre and Post aspects with Contract_Cases. The semantics are largely intuitive: the pre- and postconditions must be obeyed in addition to the contract cases. See the SPARK *2014 Reference Manual* (SPARK Team, 2014a) for the full details.

6.2.7 *Runtime Errors in Assertions*

Because assertions are executable, the possibility exists that a runtime error could occur while the assertions are being evaluated. Thus, executing an assertion might raise an exception other than Assertion_Error because of problems in the assertion itself. The SPARK tools will generate verification conditions for

the assertions to prove that they, along with the rest of the program, are free of runtime errors.

If the assertion is possibly false or outright illogical, it is appropriate for the tools to object to it. However, there are situations in which an assertion is mathematically true and yet causes runtime errors when evaluated. There is only one way that can happen: arithmetic overflow. If the assertion contains other kinds of runtime errors such as division by zero or accessing an array out of bounds, the assertion does not make sense anyway.

As an example, consider the following silly procedure with a precondition that expresses a true fact about the parameters:

```
procedure Silly (X      : in  Positive ;
                 Y      : in  Positive ;
                 Result : out Positive )
   with
      Pre => ((X + Y) / 2 in Positive)
is
begin
   if X > 10 then
      Result := 1;
   else
      Result := Y / 2 + 1;
   end if ;
end Silly ;
```

The precondition asserts that the average of the two Positive parameters is in the range of Positive and then does some pointless computations that nevertheless are completely free of runtime error and obey the synthesized flow dependency contract. There is nothing wrong with this procedure, yet the evaluation of the precondition may raise Constraint_Error because X + Y might overflow the base type of Positive . The SPARK tools will generate an unprovable verification condition attempting to check that overflow will not occur.

One way to work around this is to write the assertions carefully so that they too will be free of runtime error. However, you may feel frustrated by this, particularly if you plan to deploy your program with assertion checking disabled, as we discuss in Section 6.8. The assertions are, after all, intended to be statements about the design of your system. Why can they not be evaluated in the pure mathematical world where concerns about machine limitations do not exist?

In fact, the GNAT compiler and SPARK tools provide some options for controlling the way overflow is handled. Three different overflow modes are provided:

- STRICT: Overflow is handled as according to the Ada standard. Arithmetic computations are done in a subtype's base type.
- MINIMIZED: Computations are done in an oversized integer type selected by the compiler such as Long_Long_Integer (which is 64 bits for the GNAT compiler). This will not prevent all possibilities of overflow, but it will prevent many common cases and remains reasonably efficient.
- ELIMINATED: Computations are done in an extended integer type with unbounded size. No overflow is possible although, conceivably, Storage_Error might be raised.

The overflow mode can be selected separately for both general computations in the normal part of your program and for computations done in assertions. By default the tools use STRICT for both. In cases where you plan to deploy with assertions disabled, you might consider changing the overflow mode on assertions to ELIMINATED. This reduces the number of verification conditions that must be proved in the assertions themselves, and the performance penalty of doing computations with extended integers will not be paid if the assertions are not actually executed anyway. See the GNAT and SPARK user's guides for more information about overflow mode handling. We show a more practical example of these issues in Section 6.7.

Keep in mind that the expressions in assertions are general and could call functions that are not in SPARK. Such functions might raise any exception at all. Assertions might also raise Storage_Error if they involve unbounded recursion. For example, if the postcondition on some function F called F itself, the postcondition of the second call would be checked as part of evaluating the postcondition of the first call, and so forth. This is not necessarily an error; if the postcondition is written properly, it will have a base case that does not entail calling F.

For example, consider the following specification of a function Fibonacci to compute Fibonacci numbers:

```
function Fibonacci(N : Natural) return Natural
  with
    Contract_Cases =>
      (N = 0 => Fibonacci'Result = 0,
       N = 1 => Fibonacci'Result = 1,
       others => Fibonacci'Result = Fibonacci(N - 1) + Fibonacci(N - 2));
```

We leave it as an exercise to the reader to consider what happens if the Fibonacci function is implemented recursively as well. It probably goes without saying that recursive assertions are best avoided.

6.3 Assert and Assume

The assertions we have seen so far are contractual in that they form part of the design of your system and should, ideally, be written when the package specifications are written. In this section we look at assertions that can be used in the body of a subprogram to make statements about a computation in progress. In essence these assertions are implementation details that, in a perfect world, would be unnecessary. However, because theorem proving is an evolving technology, it is sometimes necessary for you to assist the tools by providing "stepping stones" that allow an overall proof to be built from several simpler proofs. In the language of mathematics the assertions we cover in this section allow you to state, in effect, lemmas and corollaries to simplify proving more complex statements.

All three of the assertions in this section are provided as pragmas rather than aspects. This is because, unlike the previous assertions, they behave more like executable statements rather than as information associated with a declaration. The Assert pragma is part of Ada, but the other two – Assert_And_Cut and Assume – are specific to SPARK.

6.3.1 *Assert*

The Assert pragma allows you to specify an arbitrary condition that you believe to be true. The pragma can appear any place in a subprogram where a declaration or an executable statement can appear. Each assertion carries a proof obligation to show that the specified condition is always true at that point. You can use Assert as a kind of check on your thinking. Suppose in the middle of some complex subprogram you find yourself saying something like, "X should be greater than one here." You can express this idea in the program itself like this:

```
pragma Assert (X > 1);
```

When we humans reason about programs, we make many such statements in our minds. We then use these statements to reason about other constructs appearing later in the program. For example, we might say, "Because X was greater than one a few lines before, thus-and-such an assignment statement

will always assign an in-bounds value." For each Assert pragma the SPARK tools generate a verification condition to check it. Likewise, the tools will use the asserted condition in the hypotheses of following verification conditions. Thus for both humans and SPARK, assertions can help clarify what is happening in the program and simplify the process of deriving conclusions about what the code does.

To illustrate some of the issues, we present a somewhat contrived example of a subprogram that calculates a student's semester bill at a hypothetical state university. The specification of package Students provides the necessary type and subprogram declarations.

```
with Dates;
package Students
   with SPARK_Mode => On
is
   type    Gender_Type      is (Male, Female, Unspecified);
   type    Meal_Plan_Type   is (None, On_Demand, Basic, Full);
   type    Student_ID       is range 0 .. 999_999;
   subtype Valid_Student_ID is Student_ID range 1 .. Student_ID'Last;

   type GPA_Type  is delta 0.01 range 0.00 .. 4.00;
   type Money_Type is delta 0.01 digits 7 range -99_999.99 .. +99_999.99;

   type Student_Record is private;

   -- Adjusts the tuition based on student characteristics.
   function Compute_Bill (Student : in Student_Record;
                          Base_Tuition : in Money_Type) return Money_Type
      with
         Pre => (Base_Tuition in 0.00 .. 20_000.00);

private

   type Student_Record is
      record
         Birth_Date   : Dates.Date;
         ID           : Student_ID;   -- An ID of zero means not a real student.
         Gender       : Gender_Type;
         GPA          : GPA_Type;
         Part_Time    : Boolean;
         In_State     : Boolean;      -- True if the student is a state resident.
         Resident     : Boolean;      -- True if the student resides on campus.
         Meal_Plan    : Meal_Plan_Type;
```

```
      Self_Insured  : Boolean;      -- True if the student has insurance.
   end record;

end Students;
```

The function Compute_Bill takes a suitably defined student record and a base tuition value for in-state students. It returns the final bill computed as a possibly adjusted tuition, plus fees and insurance premiums, minus any grants received by the student. Notice that Compute_Bill includes a precondition that puts a limit on the size of the Base_Tuition. An alternative approach would be to define a subtype of Money_Type that encodes the constraint on base tuition values.

Similarly, Compute_Bill returns a value of type Money_Type suggesting that negative bills might be possible. If that is not intended, one could either define a suitable subtype of Money_Type or, perhaps, use a postcondition.

The following listing shows an implementation of package Students that passes SPARK examination:

```
package body Students
   with SPARK_Mode => On
is
   function Compute_Bill (Student : in Student_Record;
                          Base_Tuition : in Money_Type) return Money_Type is
      Tuition   : Money_Type;
      Fees      : Money_Type;
      Grants    : Money_Type := 0.00;
      Insurance : Money_Type := 0.00;
   begin
      Tuition := Base_Tuition;

      if not Student.In_State then
         -- Out of state tuition is 50% higher.
         Tuition := Tuition + Tuition/2;
      end if;

      -- Compute health insurance premium.
      if not Student.Self_Insured then
         Insurance := 1_000.00;
      end if;

      -- Compute base fees depending on full-time/part-time status.
      if Student.Part_Time then
         Fees := 100.00;
      else
```

```
      Fees := 500.00;
   end if;

   -- Room and board.
   if Student.Resident then
      Fees := Fees + 4_000.00;  -- Room.
      case Student.Meal_Plan is
         when None      => null;
         when On_Demand => Fees := Fees + 100.00;
         when Basic     => Fees := Fees + 1_000.00;
         when Full      => Fees := Fees + 3_000.00;
      end case;
   else
      -- Nonresident students getting a meal plan pay a premium.
      case Student.Meal_Plan is
         when None      => null;
         when On_Demand => Fees := Fees + 200.00;
         when Basic     => Fees := Fees + 1_500.00;
         when Full      => Fees := Fees + 4_500.00;
      end case;
   end if;

   -- University policy: give high achieving students a break.
   if Student.GPA >= 3.00 then
      Grants := Grants + 250.00;

      -- Special directive from the state for very high achieving women.
      if Student.GPA >= 3.75 and Student.Gender = Female then
         Grants := Grants + 250.00;
      end if;
   end if;

   return ((Tuition + Fees) - Grants) + Insurance;
end Compute_Bill;

end Students;
```

This implementation considers a number of conditions such as the different cost of meal plans for resident and nonresident students, different base fees for full-time and part-time students, and special grants given to high achieving students.

At the time of this writing, the SPARK tools have trouble proving that the final computation of the bill given by ((Tuition + Fees) − Grants) + Insurance is

in range of Money_Type. A careful study of the procedure shows that even if Tuition, Fees, Grants, and Insurance are all at their extreme values, the overall bill should still be in range. Doing this review is tedious because of the many paths through the subprogram that need to be considered. Also, the overall bill is computed by way of both additions and subtractions so one needs to consider all combinations of both upper and lower bounds on the computed values to be sure the final result remains in range in every case.

Ideally, the SPARK tools would do all this work. However, if the tools are having problems, you can provide hints in the form of Assert pragmas. For example, you might add the following assertions just before the **return** statement:

```
pragma Assert (Tuition    in   0.00 .. 30_000.00);
pragma Assert (Insurance  in   0.00 ..  1_000.00);
pragma Assert (Fees       in 100.00 ..  7_500.00);
pragma Assert (Grants     in   0.00 ..    500.00);
```

Armed with this knowledge the SPARK tools easily prove that the overall bill is in range. The tools are also able to prove the assertions themselves, provided a suitably large timeout value is used (see Section 9.3), thus proving the entire function free of runtime errors.

It is important to understand that the assertions are only hints to the SPARK tools. A future version of the tools, or perhaps a different back-end prover, might be able to prove the entire function without the help of the assertions. In this sense, the assertions are not contractual; they are not part of the subprogram's specification.

This example illustrates three important concepts:

- What assertions are needed, if any, depends on the capabilities of the SPARK tools and the theorem provers they use. Because the tools are always evolving, you may discover that assertions needed in the past are not needed in the future.
- The Assert pragma can be used to track down problems in the proving process. We discuss this more in Section 9.3.
- Even if the SPARK tools do not need the assertions to complete the proofs, they can still add valuable documentation to your program. Unlike ordinary comments, information documented in Assert pragmas is checkable by the tools or alternatively during runtime.

6.3.2 *Assert and Cut*

In the last section we made the statement, "The SPARK tools will use the asserted condition in the hypotheses of following verification conditions." But

what, exactly, do we mean by "following verification conditions"? We must first define this concept more precisely before the use and purpose of the Assert_And_Cut pragma will make sense.

At each place in the program where a check is needed, the SPARK tools generate a verification condition concluding that the check will succeed. The hypotheses used are gathered from statements encountered on the execution path from the beginning of the subprogram to the point of the check. If there is more than one path to the check, the tools must consider all of those paths. This is necessary to show that no matter how execution arrives at a particular point, the condition being checked will succeed. Consider the following simplified version of Compute_Bill:

```
function Compute_Bill (Student : in Student_Record;
                       Base_Tuition : in Money_Type) return Money_Type is
   Fees : Money_Type;
begin
   if Student.Part_Time then
      Fees := 100.00;
   else
      Fees := 500.00;
   end if;
   return Base_Tuition + Fees;
end Compute_Bill;
```

The SPARK tools will wish to show that Base_Tuition + Fees does not go out of range of Money_Type in the **return** statement. However, in the example there are two paths by which the final statement can be reached depending on the outcome of the conditional. Both of those paths must be considered.

The SPARK tools provide two basic strategies. Using the *one proof per check* strategy, the tools generate a single verification condition that simultaneously considers all the paths to the point of the check. Using the *one proof per path* strategy, the tools generate separate verification conditions for each path. There is also a progressive mode in which the tools first attempt a single proof for the check but, failing that, will attempt to prove individual paths.

The information known when attempting a proof depends on the path taken to reach the check. In the previous example, if the **then** branch of the conditional is taken, the prover knows that Student.Part_Time is true and Fees has the value 100.00. Conversely, if the **else** branch is taken, the prover knows that Student.Part_Time is false and Fees has the value 500.00. This knowledge is added to the hypotheses of the verification condition checking Base_Tuition + Fees.

In the case where one verification condition is generated for each path, the verification conditions are relatively simple but there are more of them. Also,

failure to prove a verification condition yields specific information about which path is causing the problem. On the other hand, where one verification condition is generated for each check that includes information from all paths leading to that check, the verification conditions are fewer but more complicated. Also, if the proof fails, teasing out specific information about the failure is harder.

It is important to understand that every check after the conditional will have two paths leading to it because of the two paths produced by the conditional. This includes following conditional statements. In general, as the control flow complexity of a subprogram increases, the number of paths tends to increase exponentially. Consider the following example, also a simplified version of Compute_Bill:

```
function Compute_Bill (Student : in Student_Record;
                       Base_Tuition : in Money_Type) return Money_Type is
   Fees   : Money_Type;
   Grants : Money_Type := 0.00;
begin
   if Student.Part_Time then
      Fees := 100.00;
   else
      Fees := 500.00;
   end if;

   if Student.GPA >= 3.00 then
      Grants := Grants + 250.00;

      -- Special directive from the state for very high achieving women.
      if Student.GPA >= 3.75 and Student.Gender = Female then
         Grants := Grants + 250.00;
      end if;
   end if;
   return (Base_Tuition + Fees) - Grants;
end Compute_Bill;
```

There are two paths through the first conditional statement. Each of those two paths split when Student.GPA >= 3.00 is tested. The paths that enter the second conditional split again on the innermost **if** statement. Overall, there are six ways to reach the **return** statement and, thus, six different collections of hypotheses that need to be considered when proving (Base_Tuition + Fees) − Grants is in range.

In the version of Compute_Bill shown earlier, there are several control structures in sequence, each multiplying the number of paths, until the total number of ways to reach the final **return** statement, and the Assert pragmas just before

it, is quite large. Regardless of the proof strategy used, this increases the computational burden of proving the subprogram.

One approach to dealing with this problem is to factor large subprograms into several smaller ones. The idea is to lift out relatively independent blocks of code from the large subprogram and transform those blocks into (probably local) helper subprograms. Because the SPARK tools do their analysis on a per-subprogram basis, the number of paths in the helper subprograms do not multiply each other. Instead, the effect of the helper subprograms, and whatever paths they contain internally, is summarized by their contracts.

Another approach, that we introduce here, is to add one or more *cut points* to the subprogram. A cut point is a place in the subprogram where all incoming paths terminate and from which a single new path is outgoing. All information gathered by the SPARK tools on the incoming paths is forgotten.

The Assert_And_Cut pragma works like the Assert pragma in that it creates a proof obligation and provides information that the SPARK tools can use on the outgoing paths. However, unlike Assert, the Assert_And_Cut pragma introduces a cut point. Only a single path leaves Assert_And_Cut. Furthermore, the only information known to the SPARK tools immediately after Assert_And_Cut is that which is specifically stated in the pragma.

As an example, here is a version of the Students package body using Assert_And_Cut in function Compute_Bill:

```
package body Students3
  with SPARK_Mode => On
is
  function Compute_Bill (Student : in Student_Record;
                         Base_Tuition : in Money_Type) return Money_Type is
     Tuition   : Money_Type;
     Fees      : Money_Type;
     Grants    : Money_Type := 0.00;
     Insurance : Money_Type := 0.00;
  begin
     Tuition := Base_Tuition;

     if not Student.In_State then
        -- Out of state tuition is 50% higher.
        Tuition := Tuition + Tuition / 2;
     end if;

     pragma Assert_And_Cut (Tuition in 0.00 .. 30_000.00);
```

```
-- Compute health insurance premium.
if not Student. Self_Insured then
   Insurance := 1_000.00;
end if;

pragma Assert_And_Cut ((Tuition   in 0.00 .. 30_000.00) and
                       (Insurance in 0.00 ..  1_000.00));

-- Compute base fees depending on full-time/part-time status.
if Student.Part_Time then
   Fees := 100.00;
else
   Fees := 500.00;
end if;

-- Room and board.
if Student.Resident then
   Fees := Fees + 4_000.00;  -- Room.
   case Student.Meal_Plan is
      when None      => null;
      when On_Demand => Fees := Fees + 100.00;
      when Basic     => Fees := Fees + 1_000.00;
      when Full      => Fees := Fees + 3_000.00;
   end case;
else
   -- Nonresident students getting a meal plan pay a premium.
   case Student.Meal_Plan is
      when None      => null;
      when On_Demand => Fees := Fees + 200.00;
      when Basic     => Fees := Fees + 1_500.00;
      when Full      => Fees := Fees + 4_500.00;
   end case;
end if;

pragma Assert_And_Cut ((Tuition   in   0.00 .. 30_000.00) and
                       (Insurance in   0.00 ..  1_000.00) and
                       (Fees      in 100.00 ..  7_500.00));

-- University policy: give high achieving students a break.
if Student.GPA >= 3.00 then
   Grants := Grants + 250.00;
```

```
         -- Special directive from the state for very high achieving women.
         if Student.GPA >= 3.75 and Student.Gender = Female then
            Grants := Grants + 250.00;
         end if;
      end if;

      pragma Assert_And_Cut ((Tuition   in   0.00 .. 30_000.00) and
                             (Insurance in   0.00 ..  1_000.00) and
                             (Fees      in 100.00 ..  7_500.00) and
                             (Grants    in   0.00 ..    500.00));

      return ((Tuition + Fees) - Grants) + Insurance;
   end Compute_Bill;

end Students3;
```

Here each intermediate result of interest is stated with Assert_And_Cut as soon as it is computed. The multiple paths generated by the preceding control structures are thus blocked, and the number of paths do not multiply as one goes down the function. This keeps the verification conditions simple or small in number depending on the proof strategy being used and speeds up the proving process. However, notice how it is necessary for each Assert_And_Cut to reassert any information that needs to be carried forward.

6.3.3 *Assume*

The Assume pragma is very similar to the Assert pragma in many respects.

1. Assume contains a boolean expression that is evaluated if the assertion policy is set to Check. If that expression returns false the Assertion_Error exception is raised.
2. The SPARK tools use the asserted condition in the hypotheses of verification conditions that follow the Assume.

However, unlike Assert, the Assume pragma does not create a proof obligation. Instead, the SPARK tools just take the assumed condition as a given. Thus, it is important for you to ensure that the assumed condition is true. However, because Assume is executable, like all assertions, a false assumption may be detected during testing by way of the exception raised when it is evaluated. This means the safety of the assumption depends entirely on code review and testing rather than on proof.

In general, you should use Assume only under special circumstances and only with great care. To illustrate the potential danger, consider the effect of an assumption that is blatantly false:

```
pragma Assume (0 = 1); -- Assume nothing is something.
```

Verification conditions following this assume will contain a false hypothesis. When conjoined with the other hypotheses, the result is a false antecedent. Because a verification condition is just an implication, a false antecendent allows the verification condition to be proved no matter what the consequent might be. You can prove anything from a contradiction. Thus, the preceding Assume allows all following verification conditions to be proved.

Of course this is a silly example. It would fail in testing immediately (provided the program was compiled with an assertion policy of Check). Also, it seems clear nobody would purposely write such an assumption. However, some contradictory assumptions may be less clear. For example, consider the following Assume, where A is an array:

```
pragma Assume ((for all J in A'Range => A(J) > 0) and (A(A'First) = -1));
```

Again, all verification conditions after this contradictory assumption would be provable – even verification conditions that had nothing to do with the array A.

Contradictory assumptions might evade detection during code review if they contain complicated conditions, but such an assumption would fail at runtime and so should be easily detectable during testing. The real danger is with assumptions that might only be false sometimes, as in this example:

```
pragma Assume (C > 0);
A := B / C;
```

Using the assumption, the SPARK tools successfully prove that B / C does not entail division by zero. Yet what if the assumption is wrong? If the Assume is changed to an Assert, the tools will try to prove that C > 0 is true in all cases. The Assume does not carry that requirement.

So what is the purpose of Assume? The pragma allows you to inject information into your program that you know to be true for reasons unrelated to the program's logic. Without this information the SPARK tools may require you to add error handling or do other processing that in the larger view of your system you know to be unnecessary. In effect, Assume allows you to encode information about the external world that the tools need to know but otherwise would not.

As an example, consider an embedded system using a 64-bit counter as a kind of clock. The following procedure Tick is called each millisecond to update the clock value and do other housekeeping:

```
pragma SPARK_Mode(On);
package body TickTock
   with Refined_State => (TickTock_State => Clock_Value)
is
   type Clock_Type is range 0 .. 2 ** 63 - 1;
   Clock_Value : Clock_Type := 0;

   procedure Tick
      with
         Refined_Global  => (In_Out => Clock_Value),
         Refined_Depends => (Clock_Value =>+ null)
   is
   begin
      Clock_Value := Clock_Value + 1;
      -- Other housekeeping...
   end Tick;

end TickTock;
```

In this simple program, the SPARK tools cannot prove the incrementing of Clock_Value will stay in range. However, if the system initializes at boot time and increments Clock_Value only once every millisecond, it would take more than 290 million years to reach its maximum value.

You could push the proofs through by adding error handling:

```
if Clock_Value = Clock_Type'Last then
   Restart_System;
else
   Clock_Value := Clock_Value + 1;
end if;
```

In the event that the system is still running when the next supercontinent forms, it now has the sense to reboot and re-initalize itself. More importantly, the SPARK tools are now convinced that Clock_Value + 1 is safe. However, adding error handling like this for a condition that will never arise in any realistic scenario is more obscure than useful.

Instead, this is an appropriate place to use Assume:

```
pragma Assume (Clock_Value < Clock_Type'Last);
Clock_Value := Clock_Value + 1;
```

Now the SPARK tools discharge the verification condition associated with incrementing Clock_Value without complaint. Furthermore, the assumption made to do so is documented in the code in an easy-to-find manner. The assumption will potentially even be checked at runtime. In effect, information about the external environment in which the program runs, namely that it will be rebooted at least every 290 million years, is being made known to the tools so they can account for that information in the proofs.

The preceding example seems compelling, but even here caution is necessary. Perhaps at a later time Clock_Type is changed to be

```
type Clock_Type is range 0 .. 2 ** 31 - 1;  -- 32 bits.
```

Now Clock_Value will reach its maximum value after only 24.8 days. It is very possible the system might run that long causing Assertion_Error when the assumption fails or Constraint_Error when Clock_Value overflows if assertion checking is off. The SPARK tools will not detect this problem because it is masked by the assumption. Of course after making such a change, all assumptions should be reviewed. Fortunately, the Assume pragmas stand out in the program making it easy to locate them.

We make use of the Assume pragma in Section 9.2 when using transitivity in the proof of the selection sort introduced in Chapter 1.

6.4 Loop Invariants

Loops are a problem. Each execution of a loop is a separate path through the subprogram being analyzed. Yet, in general, the number of times a loop executes is only discovered dynamically. As a result there are potentially an unknown number of paths around a loop and leading away from that loop.

Imagine unwinding a loop so that there are as many sequential executions of its body as there are loop passes. Constructs after the loop might be reachable after one unwinding or after two unwindings, or after any number of unwindings. Verification conditions generated for checks after the loop need to account for each of these potentially infinite number of possibilities. The same applies for constructs in the loop body itself that are repeatedly visited during the loop's execution. In this section we look at the Loop_Invariant pragma provided by SPARK for managing loops. We describe the Loop_Variant pragma used for proving loop termination in Section 6.5.

Using the Loop_Invariant pragma the programmer can assert a Boolean condition that must be true at a particular point in the loop whenever that point is reached. The SPARK tools generate verification conditions considering every

path that can reach the loop invariant. If these verification conditions can be discharged, the invariant will always be true. Like all assertions, loop invariants are also executable, depending on the assertion policy, and will raise Assertion_Error if they fail.

One special feature of the Loop_Invariant pragma is that it is a cut point as we described in Section 6.3.2. All paths that reach the invariant are considered terminated at that point. Only one path leaves the invariant. This behavior is essential to control the otherwise unbounded number of paths a program with a loop might contain.

The invariant splits the loop into three parts: the path that enters the loop for the first time and terminates on the invariant, the path that goes from the invariant around the loop and terminates on the invariant again, and finally, the path that goes from the invariant and leaves the loop. Of course each of these parts may entail multiple paths if there are other control structures before, after, or inside the loop. However, this approach fixes the number of paths to something the SPARK tools can know rather than having that number depend on the number of times the loop executes.

A consequence of the invariant being a cut point is that it must appear immediately inside the loop. That means it cannot be nested inside some other control structure such as an **if** statement within the loop. An invariant that was only executed conditionally would not cut the loop nor would it limit the number of paths the SPARK tools need to consider. Loop invariants are so essential that the SPARK tools will automatically generate one for each loop where you do not provide one. The generated invariant only asserts True. It is, thus, trivial to prove but not useful in following proofs.

You might suppose that a loop invariant, even the trivial one generated by the SPARK tools if needed, would block information gathered before the loop from reaching past the loop. This would be an expected consequence of the invariant being a cut point. However, the SPARK tools have special handling that allow them to convey information gathered before the loop about objects not modified in the loop past the cut point and to verification conditions beyond the loop. This simplifies the writing of loop invariants because it is not necessary to reassert information in the invariant about objects the loop does not modify.

Like the Assert and Assume pragmas, Loop_Invariant is not really contractual. It is needed to assist the SPARK tools in proving verification conditions in the face of a potentially unknown number of loop iterations. As the tools evolve, they may become better at generating loop invariants without assistance. You may find that you need to explicitly state fewer of them in the future than in the past.

As an example, consider a package that provides a buffer type as a kind of array of characters together with some subprograms for operating on buffers. A part of the specification of such a Buffers package might look like

```
package Buffers is

   Maximum_Buffer_Size : constant := 1024;
   subtype Buffer_Count_Type is Natural  range 0 .. Maximum_Buffer_Size;
   subtype Buffer_Index_Type is Positive range 1 .. Maximum_Buffer_Size;
   type    Buffer_Type       is array (Buffer_Index_Type) of Character;

   -- Returns the number of occurrences of Ch in Buffer.
   function Count_Character (Buffer : in Buffer_Type;
                             Ch     : in Character) return Buffer_Count_Type;
end Buffers;
```

Here is the body of function Count_Character:

```
   function Count_Character (Buffer : in Buffer_Type;
                             Ch     : in Character) return Buffer_Count_Type is
      Count : Buffer_Count_Type := 0;
   begin
      for Index in Buffer_Index_Type loop
         pragma Loop_Invariant (Count < Index);

         if Buffer (Index) = Ch then
            Count := Count + 1;
         end if;
      end loop;
      return Count;
   end Count_Character;
```

The SPARK tools are interested in showing, among other things, that the value of Count will not go out of range despite it being incremented inside the loop. This is a reasonable concern. If the loop runs an excessive number of times, Count could be incremented too often. Yet in this case, the function is fine. The value of Count is initialized to zero and it is incremented at most the number of times the loop runs, which is 1,024 passes. Thus, even if the inner conditional is true for every pass, Count would only be 1,024 and still in range at the end of the loop.

To convey this information to the tools, we add a Loop_Invariant pragma to the loop asserting that whenever that point is reached, the value of Count is always less than the loop parameter. The tools can easily show this is true. On entry to the loop, Count is zero and Index is one. Each time around the loop Index

is always incremented and Count is only sometimes incremented, depending on the path. Either way Count remains less than Index if it was so on the previous iteration. Finally, the tools use the information in the loop invariant to readily show that Count + 1 will never go out of the allowed range of Buffer_Count_Type.

As a second example, consider a procedure for copying an ordinary string value into a buffer. The declaration of that procedure might look like

```
procedure Copy_Into (Buffer : out Buffer_Type;
                     Source : in  String)
  with
    Depends => (Buffer => Source);
```

If the source string is too long, this procedure is intended to truncate that string and only copy the characters that will fit into the buffer. If the source string is too short, the buffer is to be padded with spaces. Here is an implementation of this procedure:

```
procedure Copy_Into (Buffer : out Buffer_Type;
                     Source : in  String) is
   Characters_To_Copy : Buffer_Count_Type := Maximum_Buffer_Size;
begin
   Buffer := (others => ' ');  -- initialize to all blanks
   if Source'Length < Characters_To_Copy then
      Characters_To_Copy := Source'Length;
   end if;
   for Index in Buffer_Count_Type range 1 .. Characters_To_Copy loop
      pragma Loop_Invariant
        (Characters_To_Copy <= Source'Length and
         Characters_To_Copy  = Characters_To_Copy'Loop_Entry);

      Buffer (Index) := Source (Source'First + (Index - 1));
   end loop;
end Copy_Into;
```

After determining how many characters actually need to be copied, a loop is used to do the copying one character at a time. The loop invariant asserts that the value of Characters_To_Copy does not change as the loop executes. It accomplishes this using the 'Loop_Entry attribute allowing you to refer to a value a variable has when the loop is first entered. The 'Loop_Entry attribute is, thus, similar to the 'Old attribute in that its use requires the compiler to maintain a copy of the variable's earlier value.

The current generation of the SPARK tools does not actually need the loop invariant we wrote in procedure Copy_Into. This relaxation is a result of the special handling afforded to values, such as Characters_To_Copy that do not change

inside the loop; it is not actually necessary to reassert information about them at the cut point created by the invariant. However, an earlier generation of the SPARK tools did require the invariant because the technology was less mature at that time. This illustrates the point that the number and nature of the non-contractual assertions required in your programs may change as the tools evolve. However, contractual assertions such as pre- and postconditions embody design information and are to a certain extent tool independent.

Finding an appropriate loop invariant requires a certain amount of practice and skill. You need to find a condition that describes the work of the loop in a nontrivial way, is easy for the tools to prove, and provides useful information for later verification conditions to use.

As a more complex example, consider a procedure that converts an IP version 4 (IPv4) address into a dotted decimal string suitable for display to humans.[11] This procedure is part of package Network.Addresses from the Thumper project that we describe in Section 8.5. Here the specification of the package is shown, in part. In this code the type Network.Octet is an 8-bit modular type holding values in the range of 0 to 255.

```
package Network.Addresses is

   type IPv4 is private;

   subtype Address_String_Index_Type is Positive range 1 .. 15;
   subtype Address_String_Type is String (Address_String_Index_Type);
   subtype Address_Length_Type is Natural range 7 .. 15;

   procedure To_IPv4_String (Address      : in  IPv4;
                             Text         : out Address_String_Type;
                             Character_Count : out Address_Length_Type)
      with
         Global => null,
         Depends => ((Text, Character_Count) => Address);

private
   subtype IPv4_Address_Index_Type is Integer range 1 .. 4;
   type IPv4 is array (IPv4_Address_Index_Type) of Network.Octet;

end Network.Addresses;
```

This package takes advantage of the fact that the text form IP addresses require at most fifteen characters. It thus defines a suitable subtype to express this limitation. However, because some IP addresses are shorter than fifteen

characters, the procedure To_IPv4_String also outputs a count of the number of characters that were actually required. The procedure pads the output string with spaces in that case. Here is one attempt at implementing this procedure:

```
subtype Digit_Type is Character range '0' .. '9';
subtype Value_Type is Network.Octet range 0 .. 9;
type    Value_To_Digit_Type is array(Value_Type) of Digit_Type;

Digit_Lookup_Table : constant Value_To_Digit_Type :=
  Value_To_Digit_Type'('0', '1', '2', '3', '4', '5', '6', '7', '8', '9');

procedure To_IPv4_String (Address         : in  IPv4;
                          Text            : out Address_String_Type;
                          Character_Count : out Address_Length_Type) is

   subtype Skip_Type is Positive range 1 .. 4;

   Index   : Address_String_Index_Type;
   Count   : Natural;
   Skip    : Skip_Type;
   Value   : Network.Octet;
   Digit_2 : Digit_Type;
   Digit_1 : Digit_Type;
   Digit_0 : Digit_Type;
begin
   Text  := Address_String_Type'(others => ' ');
   Count := 0;

   -- For each octet...
   for J in IPv4_Address_Index_Type loop

      -- Compute starting position in output string.
      Index := Text'First + Count;

      -- Compute the digit characters for this octet.
      Value   := Address (J);
      Digit_2 := Digit_Lookup_Table(Value / 100);
      Value   := Value rem 100;
      Digit_1 := Digit_Lookup_Table(Value /  10);
      Value   := Value rem 10;
      Digit_0 := Digit_Lookup_Table(Value);
```

```
      -- Output the digits appropriately.
      if Digit_2 /= '0' then
         Text (Index + 0) := Digit_2;
         Text (Index + 1) := Digit_1;
         Text (Index + 2) := Digit_0;
         Skip := 3;
      elsif Digit_1 /= '0' then
         Text (Index + 0) := Digit_1;
         Text (Index + 1) := Digit_0;
         Skip := 2;
      else
         Text (Index + 0) := Digit_0;
         Skip := 1;
      end if;

      -- Place the dot unless this is the last octet.
      if J /= IPv4_Address_Index_Type'Last then
         Text (Index + Skip) := '.';
         Skip := Skip + 1;
      end if;

      -- Update Count.
      Count := Count + Skip;
   end loop;

   Character_Count := Count;
end To_IPv4_String;
```

This procedure works by looping over each byte in the IP address and filling in appropriate text in the output string as it works. The amount of space in the output string used by each pass of the loop depends on the value of the address component being processed. Sometimes only one digit is needed, but sometimes up to three digits are required. The value of Skip records how much space was used in the current loop pass; that value is used to update the running total of the number of characters consumed so far.

The subprogram contains no flow errors, but the SPARK tools have difficulty proving that the various accesses of the array Text are in bounds. The tools do not "understand" that the loop will execute a limited number of times and never run Count and, hence, Index up to an excessive value.

To help the proofs succeed, we must add a loop invariant that explains to the SPARK tools that Count is suitably bounded. We must find a condition that is both

within the tools' ability to prove and yet also adds enough information to let the tools complete the proofs they are stuck on. To find an appropriate invariant, start by asking the question, How do we, as humans, know this code works? Answering this question gives us insight about what we must tell the SPARK tools. Furthermore, if the procedure is in fact faulty, attempting to explain to ourselves why it works will likely reveal the fault. This is the essence of how SPARK transforms incorrect programs into reliable ones.

If we study the procedure, we can see that each loop pass adds at most four characters to the output string (up to three digit characters and a dot). Thus, as a first attempt we add the following loop invariant immediately inside the for loop:

```
pragma Loop_Invariant (Count <= 4*(J-1));
```

This increases the number of verification conditions to be proved as the loop invariant adds additional proof obligations. However, this invariant does allow the tools to verify that the accesses to the Text array are all in bounds – a significant step forward. The only two remaining objections are to the statement

```
Count := Count + Skip;
```

at the end of the loop and to the final assignment to Character_Count before the procedure returns.

One problem is that the last loop pass is special but we do not convey any information about that to the tool. In particular, the dot character is not output in the last pass so only three characters at most are added to the output string. This is important because the size of the output string is exactly fifteen characters and not sixteen as would be required if four characters were output with every pass. To express this concept, we add the following assertion to the end of the loop immediately before the statement Count := Count + Skip:

```
pragma Assert (if J < IPv4_Address_Index_Type'Last
               then Skip <= 4
               else Skip <= 3);
```

This informs the tools that the skip distance is at most three during the last loop iteration and allows the tools to discharge the verification condition associated with updating Count.

The remaining issue is on the statement

```
Character_Count := Count;
```

The problem here is that Character_Count has type Address_Length_Type, which is constrained to the range 7 .. 15. The assertions so far only put an upper bound on the number of characters written, and the tools are having trouble showing

that at least seven characters are output. To address this, we change the loop invariant to

```
pragma Loop_Invariant (Count >= 2*(J-1) and Count <= 4*(J-1));
```

Now the tools are able to discharge all verification conditions in the subprogram, proving that it is free of any possibility of runtime error.

It might seem as if this process would have been easier if the types had not been so precisely defined. For example, if Address_Length_Type had a lower bound of one instead of seven, perhaps the final step would not have been necessary. However, loosening type definitions for the sake of making certain proofs easier is almost never the right approach. In fact, loosely specified types typically make proofs more difficult, if not in one place then in another. The tools might be able to use the tight constraint on Address_Length_Type to its advantage when proving verification conditions elsewhere related to IP addresses. Always strive to tighten type definitions; avoid loosening them. Embed as much information as you can into the program.

This example also illustrates the interplay between human reasoning and the SPARK tools. After convincing ourselves of the correctness of the code, we could have just, perhaps, recorded our reasoning in a comment and not bothered with finding suitable SPARK assertions. However, SPARK serves to check our work, which is valuable because humans are very error prone when trying to mentally manage the mass of details required while reasoning about programs. Also, the SPARK assertions are a form of machine readable documentation that can be checked automatically by other programmers less familiar with the code. If a change is made to the procedure, the SPARK tools will alert the programmer making the change to any potential runtime issues introduced.

As an example, consider the following code from the To_IPv4_String procedure:

```
-- Compute the digit characters for this octet.
Value   := Address (J);
Digit_2 := Digit_Lookup_Table (Value / 100);
Value   := Value rem 100;
Digit_1 := Digit_Lookup_Table (Value /  10);
Value   := Value rem 10;
Digit_0 := Digit_Lookup_Table (Value);
```

It is necessary that the values used to index the lookup table are all in the range 0 .. 9. No doubt the programmer considered that when writing the code initially. The SPARK tools also proved this without comment so the programmer did not need to spend time reviewing the code for that error. The tools can thus take care of many "simple" proofs and only require human assistance for the

more difficult cases. However, ultimately it is the human and not the tools that generated the "proofs" for the code, even if just mentally. The tools simply serve to check the human's work.

As a final example, consider this implementation of procedure Binary_Search that we specified on page 167:

```
package body Searchers2
  with SPARK_Mode => On
is

   procedure Binary_Search (Search_Item : in  Integer;
                            Items       : in  Array_Type;
                            Found       : out Boolean;
                            Result      : out Index_Type) is

      Low_Index  : Index_Type := Items'First;
      Mid_Index  : Index_Type;
      High_Index : Index_Type := Items'Last;
   begin
      Found := False;
      Result := Items'First;  -- Initialize Result to "not found" case.

      -- If the item is out of range, it is not found.
      if Search_Item < Items(Low_Index) or Items(High_Index) < Search_Item then
         return;
      end if;

      loop
         Mid_Index := (Low_Index + High_Index) / 2;
         if Search_Item = Items(Mid_Index) then
            Found := True;
            Result := Mid_Index;
            return;
         end if;

         exit when Low_Index = High_Index;

         pragma Loop_Invariant
           (Search_Item in Items(Low_Index) .. Items(High_Index));

         if Items(Mid_Index) < Search_Item then
            Low_Index := Mid_Index;
```

```
            else
               High_Index := Mid_Index;
            end if;

         end loop;
      end Binary_Search;

end Searchers2;
```

The implementation is non-recursive and relatively straightforward. To complete the proof, an appropriate loop invariant must be given. As usual finding the right loop invariant is the trickiest part of the problem. At the point where the invariant is given the search item has not yet been found and, if it exists in the array at all, it resides between position Low_Index and High_Index inclusive.

With the invariant shown, the preceding implementation of Binary_Search is proved. It will never exhibit any runtime error and it honors its strong postcondition.

Unfortunately, the preceding implementation contains a serious fault. Under certain circumstances the loop runs infinitely without ever making any progress. Consider the case where Low_Index and High_Index are adjacent and the array elements under consideration are the natural numbers 10 and 20. Suppose the search item is 20. The implementation computes a Mid_Index equal to Low_Index in this case. Because the item in the array at that position (10) is less than the search item (20), the implementation sets Low_Index to the value of Mid_Index, which does not change anything. The procedure loops forever.

Although the tools have proved the procedure honors its postcondition, that is only applicable if the procedure returns at all. We have not yet proved that the procedure actually terminates in every case. We discuss how to do that in Section 6.5.

6.5 Loop Variants

In the previous section we saw that it is possible for the SPARK tools to prove even strong postconditions about a subprogram and yet for there to still be serious faults in that subprogram. This unexpected effect occurs because the proof of a postcondition does not consider the possibility of nontermination. The subprogram may simply execute forever without returning. In general, nontermination can occur if the subprogram contains unbounded recursion or

if it contains loop statements. In this section we look at how we can prove that a subprogram has no infinite loops.

Many loops in SPARK subprograms obviously terminate and do not need any special handling. For example, a **for** loop runs its loop parameter over a range that is computed when the loop is first encountered. The rules of Ada prevent the loop parameter from being changed in the loop, so it can only advance across the specified range of values. Such loops are guaranteed to terminate once the loop parameter has exhausted its range. However, **while** loops and loops constructed without an iteration scheme (a bare **loop** statement) may run for an indeterminate number of passes. Proving that such loops terminate is often desirable and even necessary as part of a full proof of the enclosing subprogram's correctness.

Proving loop termination can be done with the help of the Loop_Variant pragma. The semantics of Loop_Variant are more complex than for the other assertions we have seen so far. Loop_Variant allows you to define an expression that either always increases (or always decreases) as the loop executes. If the value of the expression has a suitable upper (or lower) bound, then the loop must end because the value of the expression cannot increase (or decrease) infinitely without crossing the bound. The expression is not allowed to stay the same between loop iterations. The expression must "make progress" monotonically toward a bound. Otherwise, the loop might execute forever. For example, the Loop_Variant pragma

```
pragma Loop_Variant (Increases => A - B + C);
```

asserts that the expression A − B + C increases with each iteration of the loop.

The Loop_Variant pragma allows you to specify several expressions. Each expression is prefixed with a change direction of either Increases or Decreases. Here is an example using the integer variables, X, Y, and Z:

```
pragma Loop_Variant (Increases => X, Decreases => Y - Z);
```

This assertion states that during each loop iteration, either the value of X increases or the value of the expression Y − Z decreases. When there are multiple expressions in the pragma, any particular expression may stay the same during a particular iteration as long as one of the other expressions moves in the specified direction.

The order of the expressions in pragma Loop_Variant is significant. In each iteration, expressions are checked in textual order until either a change is found or all expressions have been checked. The assertion is true if the last expression checked moved in the specified direction and false otherwise. Any expressions

after the one that moved are ignored. So in our example, if X increases, what happens to Y − Z is not considered.

The expressions in a Loop_Variant pragma must have a discrete type. The domain of any discrete type consists of a *finite* set of ordered values. Therefore, the values of a discrete type are automatically bounded both above and below. In our examples, the expressions have type Integer and so are bounded by Integer'First and Integer'Last. It is not necessary for you to specify any other bounds.

Of course the loop may not run the expressions all the way to their bounds. That is not important. It is only necessary for the bound to exist and for the loop to increase (or decrease) the value of the expressions monotonically. That is sufficient to prove that the loop must eventually terminate.

Like all assertions, loop variants are executable under the control of the assertion policy. If they fail, Assertion_Error is raised as usual. Also, like all assertions, the SPARK tools will create a verification condition to prove that the variant never fails. Discharging that verification condition proves that the loop terminates.

In the Binary_Search example in Section 6.4, the procedure made use of a bare **loop** statement with an **exit** statement that ends the loop under certain conditions. Unlike **for** loops, a loop of this form may conceivably execute forever. It is thus appropriate in this case to use Loop_Variant to prove that will not happen.

To find an appropriate loop variant start by asking, What expression describes the progress the loop is making? In the case of Binary_Search one possibility is that High_Index and Low_Index always get closer together:

```
pragma Loop_Variant (Decreases => High_Index - Low_Index);
```

If each iteration of the loop reduces the distance between the two indicies, eventually the loop will end.

The subprogram exits the loop when the difference between the indicies is zero. However, you might wonder what would happen if High_Index − Low_Index skipped over zero and became negative.

The type of the loop variant expression in this case is Integer (the base type of Index_Type). In theory the smallest possible value of High_Index − Low_Index is -99. This could occur if the two index values were at their appropriate extremes. If the loop were truly infinite and yet the loop variant succeeded, then High_Index − Low_Index would decrease forever. Eventually, Constraint_Error would be raised when one of the two indices goes out of bounds or, if not that,

when the subtraction overflows the range of Integer. If the code proves free of runtime error, neither of these cases can occur; the program cannot have both an infinite loop and a satisfied loop variant at the same time.

Unfortunately, the implementation of Binary_Search in Section 6.4 does contain a possible infinite loop, and as you would expect, the loop variant fails to prove. To fix the subprogram, it is necessary to correct the error in such a way as to maintain the proofs of freedom from runtime error, the post-condition, and some suitable loop variant. An implementation that does so follows:

```
package body Searchers3
  with SPARK_Mode => On
is
   procedure Binary_Search (Search_Item : in  Integer;
                            Items       : in  Array_Type;
                            Found       : out Boolean;
                            Result      : out Index_Type) is
      Low_Index  : Index_Type := Items'First;
      Mid_Index  : Index_Type;
      High_Index : Index_Type := Items'Last;
   begin
      Found := False;
      Result := Items'First;  -- Initialize Result to "not found" case.

      -- If the item is out of range, it is not found.
      if Search_Item < Items(Low_Index) or Items(High_Index) < Search_Item then
         return;
      end if;

      loop
         Mid_Index := (Low_Index + High_Index) / 2;
         if Search_Item = Items(Mid_Index) then
            Found := True;
            Result := Mid_Index;
            return;
         end if;

         exit when Low_Index = High_Index;

         pragma Loop_Invariant (not Found);
         pragma Loop_Invariant (Mid_Index in Low_Index .. High_Index - 1);
         pragma Loop_Invariant (Items(Low_Index) <= Search_Item);
```

```
            pragma Loop_Invariant (Search_Item <= Items(High_Index));
            pragma Loop_Variant (Decreases => High_Index - Low_Index);

            if Items(Mid_Index) < Search_Item then
               if Search_Item < Items(Mid_Index + 1) then
                  return;
               end if;
               Low_Index := Mid_Index + 1;
            else
               High_Index := Mid_Index;
            end if;

         end loop;
      end Binary_Search;

end Searchers3;
```

The key idea is to assign Mid_Index + 1 to Low_Index to force Low_Index to advance in the case of a two-element subsequence as described previously. However, this now requires an extra test in case the extra advance skips past the search item's value in the array. These changes caused the SPARK tools to have trouble proving the postcondition in the case of the resulting early return. Adding **not** Found to the loop invariant clarified for the tools which part of the postcondition was relevant.

Finally, it was necessary to assert a loop invariant that described the relationship between Low_Index, Mid_Index, and High_Index so the effect of the assignments at the bottom of the loop could be tracked. With these changes the SPARK tools are able to prove that the subprogram works and never loops infinitely.

We should note that technically the SPARK tools only allow a single loop invariant in each loop, but as a convenience, it is permitted, as was done in the previous example, to use several Loop_Invariant pragmas in a row. The effect is to create an overall invariant that is the conjunction of the individually listed invariants.

Finally, it bears mentioning that technically proving a particular loop variant does not by itself prove a subprogram returns. A subprogram might contain multiple loops on various paths; proving that one loop terminates leaves open the possibility that a different loop might run infinitely. To ensure that a subprogram always returns, it is necessary to prove that all loops contained in the subprogram that can be feasibly reached terminate. In practice this is not normally an issue, but the SPARK tools by themselves do not provide any direct checking of this requirement.

6.6 Discriminants

Ada provides a way to parameterize record types using discriminants. The basics of this topic were briefly described in Section 2.3.6. In this section we provide a more detailed example of the use of discriminants and show how they interact with SPARK proofs.

A discriminated type is a kind of indefinite type similar in some respects to an unconstrained array type. To declare an object, it is necessary to provide a specific value for the discriminant. The value provided can be dynamically computed, giving you flexibility. However, different objects with different discriminants are still of the same type and thus can be, for example, passed to a single subprogram that accepts that type. In Section 6.7 we show an example of using SPARK with an alternative method of parameterizing types, namely, generics.

As a concrete example of using discriminated types, consider the problem of doing integer computations on very large values. The integer types built into your Ada compiler may support 64-bit computations and conceivably even larger sizes, but they will be limited by whatever is natural for your hardware. However, some applications have a need to do computations on extremely large integers with, for example, 1,024 bits or even more. Many cryptographic algorithms such as RSA or elliptic curve-based cryptosystems need to manipulate such *extended precision* integers.

It is natural to design a package supporting an extended precision integer type. In this example, we will call that type Very_Long. Following the common Ada convention of making the name of a type package plural, we will call the package that defines our type Very_Longs.

Unfortunately, different applications have different needs regarding the size of the numbers they must manipulate. We could design Very_Long to expand (and contract) dynamically as needed, but the natural way of doing this would entail the use of memory allocators, a feature not supported by SPARK. Alternatively, we could set a fixed size for Very_Long, picking a size large enough to support any conceivable application. However, this may still not be enough for some exceptional applications and will waste space and time in the majority of cases where a very large size is not needed.

The type Very_Long is thus a prime candidate for being discriminated with a value that gives the size of the integer. Different objects could thus have different sizes as needed and yet all be of the same type.

For cryptographic applications modeling signed integers, using the usual mathematical operations is not normally needed. Instead, unsigned, modular integers tend to be more useful where addition and subtraction "wrap around"

inside a value with a fixed number of bits without overflow. These are the kinds of integers we show in our example. The full specification of package Very_Longs with line numbers for reference is as follows:

```
1  pragma SPARK_Mode(On);
2
3  package Very_Longs is
4
5     -- Here "Digit" means a base 256 digit.
6     Maximum_Length : constant := 2**16;
7     type    Digit_Count_Type is new Natural range 0 .. Maximum_Length;
8     subtype Digit_Index_Type is Digit_Count_Type range 1 .. Digit_Count_Type'Last;
9     type    Very_Long (Length : Digit_Index_Type) is private;
10
11    -- Constructors.
12    function Make_From_Natural (Number : in Natural;
13                                Length : in Digit_Index_Type) return Very_Long
14       with Post => Make_From_Natural'Result.Length = Length;
15
16    procedure Make_From_Hex_String (Number : in String;
17                                    Result : out Very_Long;
18                                    Valid  : out Boolean)
19       with
20          Depends => ((Result, Valid) => (Number, Result)),
21          Pre => Number'Length = 2*Result.Length;
22
23    -- Relational operators. Only Very_Longs of equal size can be compared.
24    function "<" (L, R : in Very_Long) return Boolean
25       with Pre => L.Length = R.Length;
26
27    function "<=" (L, R : in Very_Long) return Boolean
28       with Pre => L.Length = R.Length;
29
30    function ">" (L, R : in Very_Long) return Boolean
31       with Pre => L.Length = R.Length;
32
33    function ">=" (L, R : in Very_Long) return Boolean
34      with Pre => L.Length = R.Length;
35
36    -- Returns True if Number is zero.
37    function Is_Zero (Number : in Very_Long) return Boolean;
38
39    -- Returns the number of significant digits in Number.
```

```
40  function Number_Of_Digits(Number : in Very_Long) return Digit_Count_Type;
41
42     -- Modular addition (modulo 256**Length).
43     function ModAdd (L, R : in Very_Long) return Very_Long
44        with
45           Pre  => L.Length = R.Length,
46           Post => ModAdd'Result.Length = L.Length;
47
48     -- Modular subtraction (modulo 256**Length).
49     function ModSubtract (L, R : in Very_Long) return Very_Long
50        with
51           Pre  => L.Length = R.Length,
52           Post => ModSubtract'Result.Length = L.Length;
53
54     -- Modular multiplication (modulo 256**Length).
55     function ModMultiply (L, R : in Very_Long) return Very_Long
56        with
57           Pre  => L.Length = R.Length,
58           Post => ModMultiply'Result.Length = L.Length;
59
60     -- Ordinary multiplication.
61     function "*" (L, R : in Very_Long) return Very_Long
62       with Post => "*"'Result.Length = L.Length + R.Length;
63
64     -- Division returns quotient and remainder.
65     procedure Divide (Dividend  : in  Very_Long;
66                       Divisor   : in  Very_Long;
67                       Quotient  : out Very_Long;
68                       Remainder : out Very_Long)
69        with
70           Depends => (Quotient =>+ (Dividend, Divisor),
71                       Remainder =>+ (Dividend, Divisor)),
72           Pre => (Number_Of_Digits (Divisor) > 1)     and
73                  (Divisor.Length  = Remainder.Length) and
74                  (Dividend.Length = Quotient.Length ) and
75                  (Dividend.Length = 2*Divisor.Length);
76
77  private
78     type Octet is mod 2**8;
79     type Double_Octet is mod 2**16;
80
81     type Digits_Array_Type is array (Digit_Index_Type range <>) of Octet;
82
```

```
83  -- The bytes are stored in little endian order.
84     type Very_Long (Length : Digit_Index_Type) is
85        record
86           Long_Digits : Digits_Array_Type (1 .. Length);
87        end record;
88
89     function Is_Zero (Number : in Very_Long) return Boolean is
90        (for all J in Number.Long_Digits'Range => Number.Long_Digits (J) = 0);
91
92  end Very_Longs;
```

The package starts by declaring three types. We regard a Very_Long as being expressed in the base 256, where each *extended digit* is an 8-bit value in the range of 0–255. In our discussion of this example, and in the code itself, we use the word *digit* to mean a base 256 extended digit.

Lines 6 and 7 introduce a type used for counting digits in a Very_Long. The maximum number of digits we choose to support is the somewhat arbitrary number 2^{16}. Yet each Very_Long is the length it needs to be; they do not all need to have 2^{16} digits. However, imposing a reasonable bound simplifies many of the proofs. If Digit_Count_Type did not apply any constraints on Natural, the code would be forced to deal with integers having potentially billions of digits. Certain computations on lengths would tend to overflow in this general case. Limiting the range of Digit_Count_Type allows those computations to complete successfully without intricate handling.

Although the limit of 2^{16} digits may seem arbitrary, it is no more arbitrary than a limit of $2^{31} - 1$ that would be typical of Natural'Last on a 32-bit system. If some arbitrary limit must be specified, why not choose one that gives enough "headroom" for doing simple calculations on lengths without overflowing? We note that if this code is compiled for a 16-bit system where Natural'Last is only $2^{15} - 1$, the code will either fail to compile outright or, at worst, fail to prove. Either way the potential problem will be caught during development.

Line 8 introduces a subtype used to index digits in a Very_Long. The indexing discipline assigns the least significant digit the index 1. Although it might seem more natural to start the indexing at zero, that turns out to be unworkable as a result of certain limitations on discriminants we describe shortly.

Line 9 introduces the discriminated Very_Long type itself as private. The discriminant specifies the number of digits in the value. For example, a 1024-bit integer would be $1024/8 = 128$ digits in length. Although the details of Very_Long are hidden from the user, the discriminant is not and, instead, behaves much like a public component of the type. As a result it can be used in pre- and postconditions on the public subprograms.

Notice also that these declarations use the Ada type system to enforce the restriction that zero length Very_Long objects are not allowed. The discriminant cannot take the value zero; every object must be at least one digit in length. Of course, all the digits of a Very_Long could be zero so the length of a Very_Long and the number of significant digits it contains are two separate matters.

There are two constructor subprograms provided on lines 12–21. They allow Very_Long objects to be created from ordinary natural numbers and also from strings of hexadecimal digits. The later subprogram is useful for initializing extremely large values. Because arbitrary strings may contain characters that are not hexadecimal digits, the subprogram returns a Valid parameter set to false if invalid characters are found. Checking this requires error handling at runtime. An alternative strategy would be to strengthen the precondition as follows:

```
Pre => Number'Length = 2 * Result.Length and
       (for all J in Number'Range => Is_Hex_Digit (J))
```

This assumes a function Is_Hex_Digit with the obvious meaning is available.

Although the second approach increases the burden of proof on all callers of the constructor, it allows the error detection code inside the procedure to be removed. It also allows the Valid parameter to be removed along with all the error handling code associated with checking it. Finally, it would allow the procedure to be converted into a function that, in turn, would allow it to be used in a declarative part to initialize a Very_Long as shown in the following example:

```
Some_Number : Very_Long := Make_From_Hex_String ("FFFFFFFF");
```

This example shows an interesting cascade effect arising from moving checks from the dynamic domain of program execution to the static domain of program verification. Furthermore, it could be argued that the modification described here is superior because callers are not likely to intentionally give Make_From_Hex_String an invalid string. The original design suffers from the problem that the flow analysis done by the SPARK tools will require Valid to be checked even in the case when the programmer knows the given string is fine.

Also notice that the precondition of Make_From_Hex_String uses the expression 2*Result.Length. This is an example of a computation that would have been problematic if Digit_Index_Type had the full range of Natural. For example, 2*Natural'Last would (likely) overflow. This is also an example of how the expressions in the assertions themselves are subject to checking by SPARK, as we discussed in Section 6.2.7.

One could allow the full range of Natural in this case by rewriting the precondition as

```
Pre => Number'Length mod 2 = 0 and
       Number'Length / 2 = Result.Length
```

The first condition ensures that the length of the given string is even. Such rewritings are commonly possible, but they can be obscure. It is often easier and better to just constrain the types involved to "reasonable" ranges.

Before leaving Make_From_Hex_String, we point out that the flow dependency contract uses Result as input:

```
Depends => ((Result, Valid) => (Number, Result))
```

This might seem surprising given that Result is an **out** parameter. However, similar to the bounds on parameters of unconstrained array types, the actual parameter used in the call has a specific value of the discriminant set by the caller. The value written to, for example, Result depends on this discriminant. If Result's length is large, the value written to it will be different than if Result's length is small. Thus the dependency as shown is correct.

Returning now to the listing of Very_Long's specification, lines 24–34 declare several relational operators. Private types can already be compared for equality, but it is natural to also have the other relational operators for a numeric type such as Very_Long. Notice that the preconditions require that both numbers being compared be the same size. Although it is mathematically logical to compare numbers with different sizes, the expected application domain (cryptography) does not normally require that. Furthermore, the restriction simplifies the implementation. Notice here that the precondition is being used to describe a relationship between the parameters; something Ada's type system cannot do by itself.

Line 37 declares a convenience function Is_Zero to test if a Very_Long is zero. This function was originally motivated for use in later assertions, but it also has usefulness to clients of the package.

Line 40 declares another convenience function Number_Of_Digits that returns the number of significant digits in a Very_Long. This is different than the Very_Long's length as leading zeros are not significant. In fact, if all the digits of the Very_Long are zero, then the Number_Of_Digits returns zero.

Lines 43–62 declare three arithmetic operators for Very_Long. Unlike the case with the relational operators, most of these functions are given names rather than overloaded operator symbols. This is because they do modular calculations in which the resulting carry is ignored without error. It is best to reserve the operator symbols for functions that follow the usual mathematical

behavior. The one exception is the second multiplication operator that does produce an extended result without overflow or loss of information.

The pre- and postconditions on the arithmetic operators assert that they only work on values that are the same size and produce values with specific sizes based on their inputs. The "*" operator function produces an extended result large enough to hold the largest possible value it might produce. Notice that leading zero bits are not stripped by any of the subprograms, for example, $01_{16} \times 01_{16} = 0001_{16}$ – that is, two 8-bit values multiplied by "*" always produces a 16-bit value.

A division procedure is declared on lines 65–75. Unlike the normal "/" operator, Divide returns both the quotient and the remainder. The precondition asserts several important relationships on the sizes of the numbers involved. In summary, it requires that a $2n$-bit dividend be divided by an n-bit divisor to yield a $2n$-bit quotient and an n-bit remainder.

The division algorithm used (Knuth, 1998) requires that the number of significant digits in the divisor be strictly greater than one.[12] This requirement is stated with the precondition

```
Number_Of_Digits (Divisor) > 1
```

using the previously declared convenience function.

Because the pre- and postconditions on the various arithmetic operations all make statements about the sizes of the numbers involved, using them together works smoothly. The outputs of one operation are verified by SPARK to be compatible with the inputs of the next operation. The implementations of the operations can be simplified (possibly giving improved performance) by taking advantage of the restrictions without concern that a misbehaving program might not follow them.

The postconditions on the arithmetic operations do not attempt to capture the actual mathematical result of each operation. Doing so is beyond the scope of this example, and this illustrates that postconditions are, in general, only partial statements of subprogram behavior. Although the SPARK tools will attempt to prove the postconditions as stated are always satisfied, more complete verification of these subprograms will, at the moment, require testing.

The private section of the specification is on lines 78–91. Here, the full view of the private type is provided. The size of the Long_Digits component holding the digits themselves is specified by the discriminant.

It might seem more natural to define the Digit_Index_Type as ranging from zero to Digit_Count_Type'Last − 1. This would allow the least significant digit to be at index zero in the Long_Digits array. However, doing so would require the full view of Very_Long to look like

```
type Very_Long (Length : Digit_Index_Type) is
   record
      Long_Digits : Digits_Array_Type(0 .. Length - 1); -- illegal
   end record;
```

Unfortunately, this is illegal in Ada because the value of the discriminant cannot be used as part of an expression inside the record.

The body of package Very_Longs is too long to display fully here but may be found on `http://www.cambridge.org/us/academic/subjects/computer-science/programming-languages-and-applied-logic/building-high-integrity-applications-spark`. However, it is instructive to look at Number_Of_Digits. That function is too complicated to easily implement as an expression function. Instead it must be implemented as an ordinary function in the package body as the following shows:

```
function Number_Of_Digits (Number : in Very_Long) return Digit_Count_Type
  with
    Refined_Post =>
      (Number_Of_Digits'Result <= Number.Length) and
      (if Number_Of_Digits'Result > 0 then
         Number.Long_Digits(Number_Of_Digits'Result) /= 0) and
      (for all J in (Number_Of_Digits'Result + 1) .. Number.Long_Digits'Last
         => Number.Long_Digits (J) = 0)
is
   Digit_Count : Digit_Count_Type := 0;
begin
   if not Is_Zero (Number) then
      for Index in Number.Long_Digits'Range loop
         if Number.Long_Digits (Index) /= 0 then
            Digit_Count := Index;
         end if;

         pragma Loop_Invariant
           ((if Digit_Count > 0 then
               (Number.Long_Digits (Digit_Count) /= 0 and
                  Digit_Count in 1 .. Index)) and
            (if Index > Digit_Count then
               (for all J in Digit_Count + 1 .. Index =>
                  Number.Long_Digits (J) = 0)));
      end loop;
   end if;
   return Digit_Count;
end Number_Of_Digits;
```

For SPARK to have any hope of proving the body of Divide free of runtime error, it will need to know how Number_Of_Digits works in terms of the full view

of Very_Long. This is because Number_Of_Digits is used in the precondition of Divide to express an important restriction ensuring Divide will not raise an exception. Thus, as described in Section 6.2.3, it is necessary to give Number_Of_Digits a refined postcondition that can be used during the analysis of Divide.

6.7 Generics

As described in Sections 2.4.2 and 3.3.3, Ada supports generic subprograms and generic packages that can be instantiated by the programmer in different ways. In this section we describe how generics are handled by SPARK and, in particular, some of the issues surrounding the proof of generic code.

The central idea with generics and SPARK is that each instantiation of a generic must be proved separately. It is not possible to prove the generic code itself because the specific details of the types and values used to instantiate the generic will affect the proofs. Although in principle it might be possible to prove certain aspects of the generic code once and for all, the SPARK tools currently do not attempt to do this.

Instead, the tools do flow analysis and generate verification conditions at the point where a generic is instantiated. Each instantiation is processed separately. It is possible for all proofs to succeed for one instantiation and yet not for others. In fact, it is possible for one instantiation to be "in SPARK" and others to be outside of the SPARK language entirely – another reason the SPARK tools do not analyze the generic code directly.

We do not normally give a SPARK_Mode to a generic unit. The mode is determined by the mode at the location of the instantiation of the generic unit. However, if the body of the generic unit contains non-SPARK code, that body should be explicitly marked with SPARK_Mode (Off). We do not want the SPARK tools to analyze that body at any instantiation of that generic unit.

As an example, consider the following abbreviated specification of a generic variable package that implements a doubly linked list. The objects in the list are of a type given by the generic parameter Element_Type.

```
generic
   type Element_Type is private;
   Max_Size : Natural;
   Default_Element : Element_Type;
package Double_List
  with
    Abstract_State     => Internal_List,
     Initializes       => Internal_List,
```

```
      Initial_Condition   => Size = 0
is
   type Status_Type is (Success, Invalid_Step,
                        Bad_Iterator, Insufficient_Space);
   type Iterator is private;

   procedure Clear
     with
       Global  => (Output => Internal_List),
       Depends => (Internal_List => null),
       Post    => Size = 0;

   procedure Insert_Before (It     : in Iterator;
                            Item   : in Element_Type;
                            Status : out Status_Type)
     with
       Global  => (In_Out => Internal_List),
       Depends => (Internal_List => + (It, Item),
                   Status => Internal_List);

   function Back return Iterator
     with
       Global => null;

   function Size return Natural
     with
       Global => (Input => Internal_List);

private
   -- Position zero is a sentinel node.
   type Iterator is new Natural range 0 .. Max_Size;
end Double_List;
```

The traditional way to implement a dynamic list is to use the heap to store nodes in the list. However, if any instantiations are to be analyzed by SPARK, that approach is not possible because SPARK does not support memory allocators or access types. Instead, this package stores the list in a fixed size structure making it more properly called a *bounded* doubly linked list. Although the list can change size dynamically, the region of memory reserved for it cannot. The maximum number of list elements allowed is given as the generic parameter Max_Size.

The package defines a private Iterator type. Objects of that type are used to "point" into the list. An iterator can point at any element in the list and also at

a special end-of-list element, also called a *sentinel* element, that conceptually exists just past the last valid element. The Back function returns an iterator to the sentinel; it is valid to call Back even for an empty list.

A more complete package would also include subprograms for moving iterators forward and backward over the list and for reading and writing list items through iterators. The generic parameter Default_Element is intended to be returned when one attempts to (incorrectly) read the list's sentinel. The generic package cannot know what value would be appropriate to return in that case so it relies on the instantiation to provide such a value.

The package specification has SPARK data and flow dependency contracts and a declaration of the abstract state held by the package. Here is the body of this abbreviated package:

```
package body Double_List
   with
      Refined_State => (Internal_List => (Memory, Count, Free_List, Free))
is
   subtype Index_Type is Iterator;

   type List_Node is
      record
         Value    : Element_Type;
         Next     : Index_Type;
         Previous : Index_Type;
      end record;

   type Node_Array is array (Index_Type) of List_Node;
   type Free_Array is array (Index_Type) of Index_Type;

   Memory    : Node_Array; -- Holds the list nodes.
   Count     : Index_Type; -- Number of items on the list.
   Free_List : Free_Array; -- Maps available nodes.
   Free      : Index_Type; -- Points at the head of the free list.

   procedure Clear
      with
         Refined_Global  => (Output => (Memory, Count, Free_List, Free)),
         Refined_Depends => ((Memory, Count, Free_List, Free) => null)
   is
   begin
      -- Make sure the entire array has some appropriate initial value.
      Memory := (others => (Default_Element, 0, 0));
      Count := 0;
```

```
   -- Prepare the free list.
   Free_List := (others => 0);
   Free := 1;
   for Index in Index_Type range 1 .. Index_Type'Last - 1 loop
      Free_List (Index) := Index + 1;
   end loop;
end Clear;

procedure Insert_Before (It     : in Iterator;
                         Item   : in Element_Type;
                         Status : out Status_Type)
   with
      Refined_Global  => (Input => Free_List,
                          In_Out => (Memory, Count, Free)),
      Refined_Depends => (Memory =>+ (Count, It, Item, Free),
                          (Count, Status) => Count,
                           Free           =>+ (Count, Free_List))
is
   New_Pointer : Index_Type;
begin
   if Count = Index_Type(Max_Size) then
      Status := Insufficient_Space;
   else
      Status := Success;

      -- Get an item from the free list.
      New_Pointer := Free;
      Free := Free_List (Free);

      -- Fill in the fields and link the new item into the list.
      Memory(New_Pointer) := (Item, It, Memory(It).Previous);
      Memory(Memory(It).Previous).Next := New_Pointer;
      Memory(It).Previous := New_Pointer;

      -- Adjust count.
      Count := Count + 1;
   end if;
end Insert_Before;

function Back return Iterator is
begin
   return 0;
end Back;
```

```
   function Size return Natural is (Natural(Count))
     with Refined_Global => (Input => Count);

begin
   -- Clear the list at package elaboration time.
   Clear;
end Double_List;
```

A List_Node type is defined containing a value of the list's Element_Type along with two "pointers" to the next and previous nodes in the list. Here, the pointers are implemented as indices into a suitably dimensioned array of list nodes. Each list node has a corresponding component in the Free_List array indicating if the node is available or not. The free list itself is a singly linked list with the head given by Free. The list node at position zero is the sentinel node.

The package body also has SPARK aspects as usual, including a refinement of the package's abstract state and refined data and flow dependency contracts. Although not used in this example, the body may contain other SPARK assertions such as loop invariants needed to prove instantiations of this generic unit.

Despite the SPARK aspects, the SPARK tools do not directly analyze the generic code. Instead, the aspects are used when SPARK analyzes each instantiation. Consider a simple package that provides higher level subprograms around a particular instantiation of a list of integers. For the sake of an example, this package is shown with a single procedure Append_Range that appends a given range of integers onto the list.

```
pragma SPARK_Mode(On);
package List_Handler -- To demonstrate instantiation of generic package.
   with
      Abstract_State => List,
       Initializes   => List
is
   procedure Append_Range (Lower, Upper : in Integer)
      with
         Global  => (In_Out => List),
         Depends => (List =>+ (Lower, Upper));
end List_Handler;
```

The package is given a SPARK specification by way of the SPARK_Mode pragma. It declares as abstract state the list it is managing. Here is the body of this package:

```
pragma SPARK_Mode(On);
with Double_List;  -- The generic doubly linked list package
pragma Elaborate_All(Double_List);
```

```
package body List_Handler
   with
      Refined_State => (List => Integer_List. Internal_List )
is
   package Integer_List is new Double_List (Element_Type    => Integer,
                                            Max_Size        => 128,
                                            Default_Element => 0);
   use type Integer_List .Status_Type;

   procedure Append_Range(Lower, Upper : in Integer)
      with
         Refined_Global  => (In_Out => Integer_List. Internal_List ),
         Refined_Depends => (Integer_List. Internal_List  =>+ (Lower, Upper))
   is
      Current : Integer := Lower;
      Status  : Integer_List .Status_Type;
   begin
      while Current <= Upper loop
         Integer_List . Insert_Before (It     => Integer_List.Back,
                                       Item   => Current,
                                       Status => Status);
         exit when Status /= Integer_List.Success or Current = Upper;
         Current := Current + 1;
      end loop;
   end Append_Range;
end List_Handler;
```

Notice the refined state clause at the beginning of this package body. The abstract state List is refined to the abstract state Internal_List of the instantiated package Integer_List. In effect, the instantiation is a kind of global variable inside package List_Handler.

The **pragma** Elaborate_All(Double_List) that appears at the top of List_Handler's body controls elaborate order. As we described in Section 3.6, dependencies between packages sometimes require special measures be taken to control the order in which they are elaborated to ensure no unelaborated units are used. Because the instantiation of Double_List occurs as a global variable in the body of a library level package, the SPARK tools require that Double_List's body, and transitively the bodies of all packages it depends on, be elaborated first. This is the effect of Elaborate_All. This ensures the elaboration of the instantiation succeeds because all library units required by it will be elaborated by then.

As the SPARK tools do flow analysis and generate verification conditions for the body of package List_Handler, they also do flow analysis and generate verification conditions for the particular instantiation of Double_List being used. If the proofs all succeed, as they do for this example, that only implies the proofs for the Integer_List instantiation succeeded. Other instantiations may have failing proofs and, conceivably, may even have serious errors that the Integer_List instantiation does not.

For example, Integer_List is an instantiation of Double_List with a Max_Size of 128 elements. Because the type of Max_Size is Natural, it is possible to instantiate Double_List with a Max_Size of zero. Conceivably that boundary case may contain runtime problems that the non-zero sized case may not contain. If so, the analysis done by the tools will fail for the problematic instantiation while still passing other, better behaved instantiations.

Some instantiations might not even be SPARK. Consider an instantiation such as

```
type Integer_Access is access Integer;
package Integer_Pointer_List is
        new Double_List (Element_Type    => Integer_Access,
                         Max_Size        => 128,
                         Default_Element => null);
```

This instantiation creates a list variable that holds integer access values. However, because access types are not allowed in SPARK, this instantiation can not appear in code where SPARK_Mode is on. It is not SPARK.

Double_List is a generic *variable* package. When we instantiate an actual package from it, that instance implements a single list variable. Because it is generic, it can be instantiated multiple times in a program. Thus, our generic variable package supports the creation of multiple list variables. However, the SPARK tools will repeat the proofs for each instantiation. Also, the "variables" created in this way cannot readily be copied or compared as they have, in effect, different types.

An alternative design would be to revise Double_List into a type package that is generic only in the list's element type. One could use a discriminant on the list type to specify the maximum size of each list object at declaration time. This approach has the advantage of allowing list objects to be assigned and compared provided they had the same size (and the SPARK tools would statically prove that was so). It would also mean the tools would only need to prove the Double_List code once for each element type and not once for each list variable. We leave the details of this alternative design as an exercise for the reader.

Ada also allows individual subprograms to be generic. As an example of this, consider the following specification of a package Generic_Searchers that contains a generic procedure that does a binary search of a sorted array. This is a generic version of the Binary_Search procedure described in Section 6.2.1.

```
pragma SPARK_Mode(On);
package Generic_Searchers is

   generic
      type Element_Type is private;
      type Index_Type   is range <>;
      type Array_Type   is array(Index_Type) of Element_Type;
      with function "<"(L, R : Element_Type) return Boolean is <>;
   procedure Binary_Search (Search_Item : in  Element_Type;
                            Items       : in  Array_Type;
                            Found       : out Boolean;
                            Result      : out Index_Type)
    with
      Pre  => (for all J in Items'Range =>
                  (for all K in J + 1 .. Items'Last => Items (J) < Items (K))),
      Post => (if Found then
                  Search_Item = Items (Result)
               else
                  (for all J in Items'Range => Items (J) /= Search_Item));

end Generic_Searchers;
```

The generic Binary_Search procedure is parameterized by an array type along with suitable types for the array indices and elements. Recall that declaring Element_Type as private in this context means that any type can be used as long as it can be copied and provides equality comparison operators. The binary search algorithm makes use of equality comparison of Element_Type objects.

The declaration of Index_Type as

```
type Index_Type is range <>;
```

means that Index_Type can be any signed integral type (or subtype). In contrast, if Index_Type had been declared as

```
type Index_Type is (<>);
```

this would allow any discrete type to be used to index the array including enumeration types. However, the binary search algorithm does computations on index values to find the midpoint between two locations in the array, and those operations are more difficult with enumeration types.

Notice that the declaration of the generic array type parameter requires instantiation with only fully constrained arrays. That is, each instantiation of Binary_Search only works with arrays of a particular size. One could generalize the procedure by allowing it to work with unconstrained array types by changing the declaration of the Array_Type parameter to

```
type Array_Type is array (Index_Type range <>) of Element_Type;
```

In that case, each instantiation could work with variable sized arrays of the given type. Although it is more general, this approach has other implications that we will discuss shortly.

Because the binary search algorithm requires that Element_Type have an ordering, the last generic parameter declares a function "<" that can be used to determine if one value comes before another. The generic code does not understand anything about this function other than what is declared in the generic parameter list. However, when the SPARK tools analyze an instantiation, the tools will know at that time precisely which function is actually being used.

For example, consider an instantiation of Binary_Search as follows:

```
subtype Index_Type is Positive range 1 .. 10;
type Natural_Array_Type is array(Index_Type) of Natural;

procedure Natural_Search is
        new Searchers.Binary_Search (Element_Type => Natural,
                                     Index_Type   => Index_Type,
                                     Array_Type
=> Natural_Array_Type);
```

Here, procedure Natural_Search does a binary search over arrays of exactly ten natural numbers. Because the "<" operator for type Natural is directly visible at the point of instantiation, the compiler will automatically provide it for the "<" generic parameter (this is the meaning of the "box" symbol, <>, at the end of the generic parameter declaration).

When the SPARK tools analyze the instantiation, they understand that "<" is the ordinary less than operation on natural numbers and make use of that information in the proofs. This is helpful because the tools have some built-in knowledge of the properties of fundamental arithmetic operators including relational operators. However, consider a different instantiation using the extended precision integers presented in Section 6.6:

```
subtype Index_Type is Positive range 1 .. 10;
subtype Big_Integer is Very_Longs.Very_Long (Length => 256/8);
type Big_Array_Type is array(Index_Type) of Big_Integer
```

```
procedure Big_Search is
          new Searchers.Binary_Search (Element_Type => Big_Integer,
                                       Index_Type   => Index_Type,
                                       Array_Type   => Big_Array_Type,
                                         "<"        => Very_Longs."<");
```

In this case the elements of the array are 256-bit integers. Notice that because the "<" operator of the Very_Long type is not directly visible, it must be explicitly named as a generic argument in the instantiation.

Unlike the earlier case, the SPARK tools have no built-in knowledge of the behavior of "<" for the Very_Long type. All they know about the operator is what they learn from the contract on the operator function provided by the package. That contract does not completely specify all the relevant properties of Very_Longs."<", and thus the tools may have more difficulties with the proofs for this instantiation. The necessary properties can be provided to the SPARK tools using an *external axiomatization*, an advanced technique we discuss very briefly in Section 9.4. However, this example shows the value of analyzing each instantiation separately. The tools are able to take advantage of whatever specific information is available for each instantiation. If the tools attempted to prove the generic once and for all, they would not be able to use specialized information to simplify the proofs when appropriate.

Let us now turn our attention to the pre- and postconditions on Binary_Search, repeated here as a convenience.

```
Pre =>
  (for all J in Items'Range =>
     (for all K in J + 1 .. Items'Last => Items (J) < Items (K))),
Post =>
  (if Found then
     Search_Item = Items(Result)
   else
     (for all J in Items'Range => Items (J) /= Search_Item));
```

The precondition is similar to the one given for the nongeneric version of the procedure described in Section 6.2.1. In particular, it asserts that the input array is sorted and also, in this case, that every element in the array is unique. Be aware, however, that such a conclusion relies on the "expected" properties of "<". While there is nothing in the generic code itself that guarantees "<" behaves as expected, the SPARK tools will use whatever interpretation is justified at the point of instantiation as described previously.

It might seem more natural and more general to use "<=" in the precondition as was done in the nongeneric example. Unfortunately, no "<=" operator is

available for Element_Type so such a modification does not compile. As stated previously, because assertions are executable, they must obey the same semantics and name resolution rules as ordinary Ada code. This could be worked around in several ways. You could add a "<=" function to the generic parameter list. Alternatively, you could write out the logic in the precondition itself using

```
Items (K) < Items (J) or Items (K) = Items (J)
```

This works because Element_Type is not a limited type and thus is guaranteed to have an "=" operator.

One could consider using the simpler precondition also mentioned in Section 6.2.1:

```
Pre => (for all J in Items'First .. Items'Last - 1 =>
          Items (J) <= Items (J+1))
```

However, it is potentially problematic. Consider the case in which the array contains only one element. In that situation Items'First = Items'Last. The computation Items'Last - 1 might overflow causing a Constraint_Error exception when the precondition is evaluated. Specifically consider an instantiation of Binary_Search as follows:

```
subtype Index_Type2 is Integer range Integer'First .. Integer'First;
type Natural_Array_Type2 is array(Index_Type2) of Natural;

procedure Natural_Search2 is
        new Searchers.Binary_Search (Element_Type => Natural,
                                     Index_Type   => Index_Type2,
                                     Array_Type   => Natural_Array_Type2);
```

The Natural_Search2 procedure only searches single element arrays that are indexed by the first (most negative) integer value supported. When the precondition is evaluated, the expression Integer'First - 1 is computed. This computation likely causes an exception as a result of overflow. The SPARK tools will detect this problem when they generate and attempt to prove the verification conditions associated with the assertion as described in Section 6.2.7. Unfortunately, the preceding instantiation of Natural_Search2 will include an unprovable verification condition associated with the precondition.

Notice that the original precondition suffers from a similar problem because of the computation of J + 1. You can work around this in a couple of ways. One is to change the overflow checking mode used by the tools. Another approach is

to make the precondition expression more complex to, for example, only apply the quantified expression in the case when the array has at least two elements (a single element array is already sorted). Alternatively, in this case, you could ignore the issue. Because the tools analyze each instantiation separately, an instantiation that does not have the problem will be successfully proved without incident. In this case, it would be rather silly to instantiate Binary_Search to search arrays of size one, so the failing instantiation would likely never be attempted anyway.

In the meantime, Ada's type system will prevent you from accidentally sending an array of size one to an instantiation expecting some other size. However, notice that more caution is needed if you create a generic procedure expecting an unconstrained array type.

6.8 Suppression of Checks

Once you have successfully completed all the proofs in a program unit, such as a package body, you may wish to suppress the checks ordinarily inserted by the Ada compiler. The SPARK tools will have shown that none of those checks can ever fail, so why suffer their overhead? This allows you to create a program that is both correct and efficient.

One demonstration of this effect is in the SPARK implementation of SPARK-Skein (Chapman, Botcazou, and Wallenburg, 2011), a secure hash algorithm.[13] The SPARK implementation was proved free of runtime errors and, in fact, helped discover a fault in the original reference implementation of the algorithm in C. Yet the SPARK implementation yielded performance, with all checks suppressed, essentially identical to the C implementation.

There are actually two classes of checks to consider, and they are handled differently. The first, which we will simply call "runtime checks," are added by the compiler automatically in accordance with the Ada language. If these checks fail, one of the predefined exceptions, usually Constraint_Error , is raised.

The other kind of checks are the assertions added by the programmer in the form of pre- and postconditions, Assert pragmas, loop invariants, and so forth. These differ from the runtime checks in that they do not exist at all unless the programmer writes them. Furthermore, the compiler is not obligated to execute the assertions by default; it depends on the implementation-defined default assertion policy. Finally, a failed assertion raises the Assertion_Error exception instead of one of the four previously discussed language-defined exceptions.

The methods for suppressing checks and the issues associated with doing so are somewhat different depending on the kind of check being suppressed. In the two sections that follow we describe these issues in more detail.

6.8.1 *Runtime Checks*

Runtime checks are inserted automatically by the compiler to test for situations where one of the predefined exceptions discussed in Section 6.1 should be raised. The Ada standard allows compilers to "optimize away" any runtime checks the compiler can determine will never fail. Because some compilers may analyze the code more deeply than others, there is no easy way to know precisely which runtime checks the compiler removes and which are left behind.

The analysis done by the SPARK tools can be seen as a deepening of the analysis already done by the compiler. The tools fully analyze all runtime checks of certain kinds; if the proofs succeed, none of those runtime checks are necessary. It is thus reasonable to direct the compiler to remove all runtime checks covered by SPARK in fully proved units.

One approach to removing runtime checks in a unit is to use pragma Suppress as follows to remove all checks:

```
package body Example is
  pragma Suppress (All_Checks);

  -- etc.
end Example;
```

However, this needs to be done with care. Certain runtime checks covered by All_Checks, in particular related to memory exhaustion, are not ruled out by SPARK alone. It is theoretically possible to prove a unit free of runtime errors and yet still experience a stack overflow in that unit during execution. With all runtime checks suppressed, such an event would cause the program to execute "erroneously" or, in other words, in an undefined manner instead of raising Storage_Error as usual. This is highly undesirable in a high-integrity context.

It should be mentioned, however, that the Ada standard does not require compilers to remove runtime checks that are mentioned in pragma Suppress. The pragma only grants permission to do so. For example, the GNAT compiler will still check for stack overflow if asked, despite the use of pragma Suppress (All_Checks) in the source code. See the *GNAT Reference Manual* (2015a) for more information.

As a result, the use of pragma Suppress may have less effect on the behavior and performance of your program than you might think, both because the

compiler may insert some runtime checks anyway and because the compiler may be optimizing away some runtime checks already. As always, when dealing with performance issues, you should carefully benchmark your program before and after making changes to ensure you are actually having a useful effect.

In any case, to suppress All_Checks safely, you may need to ensure code with runtime checks suppressed will not experience any memory problems, perhaps by using additional tools as described in Section 6.1. You should also review your compiler's documentation to understand what runtime checks, if any, are retained despite the use of pragma Suppress.

A potentially safer, if more tedious, way to suppress runtime checks is to explicitly suppress only the checks that are definitely covered by SPARK's analysis. This requires using multiple Suppress pragmas, one for each check. However, any checks you do not mention or are not aware of will still be checked. Although, this might cause some unnecessary checks to remain, it is safe. The example package body that follows shows this approach.

```
package body Example is
   -- See section 11.5 in the Ada Reference Manual.
   pragma Suppress (Discriminant_Check);
   pragma Suppress (Division_Check);
   pragma Suppress (Index_Check);
   pragma Suppress (Length_Check);
   pragma Suppress (Overflow_Check);
   pragma Suppress (Range_Check);
   pragma Suppress (Tag_Check);
   pragma Suppress (Elaboration_Check);

   -- etc.
end Example;
```

Certain checks mentioned in the *Ada Reference Manual* (2012) are not listed because they pertain to features that are not legal in a SPARK unit. In particular, the checks related to access types are not suppressed. If a future version of SPARK supports analysis of those things, the previous list of Suppress pragmas would be incomplete, but that would only mean some unnecessary checks might remain. The program would not become erroneous.

6.8.2 *Assertions*

Suppressing unnecessary assertions is potentially far more important than suppressing unnecessary runtime checks. This is because assertions can be very expensive to evaluate and can even change the asymptotic running time of

the subprograms to which they are attached. We noted this effect, for example, when discussing the Binary_Search example in Section 6.2.1. Consequently, assertions have the potential to slow down programs by a huge factor, making them thousands or even millions of times slower in some cases. For programs that rely on highly efficient algorithms, removing unnecessary assertions could make the difference between meeting performance goals and total unusability.

For this reason some compilers, such as GNAT, do not execute assertions by default. Instead they are only executed by explicitly setting the assertion policy to Check. This can be done for all assertions in the program using a compiler command line option or by setting the configuration pragma Assertion_Policy to the desired policy. Once the SPARK tools have proved that all assertions will never fail, you can disable them by explicitly setting the policy to Ignore, or in the case of GNAT, simply fall back to the default behavior. It is of course necessary to recompile the program if you change the assertion policy for the change to take effect.

Of particular interest are assertions that you might wish to ignore inside a package body but still enforce at the interface to the package. Suppose, for example, that you have successfully discharaged all verification conditions in a certain package body. That means, among other things, that the preconditions on internal subprograms are satisfied at every call site in that body. If you compile the body with the assertion policy set to Ignore, you will remove the overhead associated with those precondition checks. However, you might still want to have preconditions on the public subprograms checked, at least until you have shown that every call site in the entire program necessarily satisfies them. To do this, you can set the assertion policy on the specification where the public subprograms are declared to Check.

The rule is that the assertion policy used for a particular assertion is that which is in force at the point the assertion is defined. Thus, even though the compiler inserts pre- and postcondition checks into the body of the subprograms to which they apply, it is the assertion policy in effect in the specification that affects the pre- and postconditions of public subprograms.

For example, consider the function Search_For_Zero discussed on page 157. A utility package containing this function might have a specification, in part, as follows:

```
package Utility is
   pragma Assertion_Policy (Check);

   subtype Index_Type is Natural range 0 .. 1023;
   type Integer_Array is array(Index_Type) of Integer;
```

```
   function Search_For_Zero (Values : in Integer_Array) return Index_Type
     with
       Pre => (for some Index in Index_Type => Values (Index) = 0),
       Post => Values (Search_For_Zero'Result) = 0;

end Utility;
```

Here the assertion policy is explicitly set to Check causing the precondition (and postcondition) to be checked at runtime whenever the function is called. However, the body of this package could include

```
   pragma Assertion_Policy (Ignore);
```

This removes all assertion checking inside the body itself. Notice, however, that all calls to Search_For_Zero will have the precondition (and postcondition) checked even if those calls come from inside the package.

The checking of the postcondition in this case is somewhat unfortunate because discharging all verification conditions in the body will have shown the postcondition is always satisfied. However, with the GNAT compiler, it is possible to use the Assertion_Policy pragma to selectively control each kind of assertion individually. See the *GNAT Reference Manual* (2015a) for more information. For example, the specification could be written as follows.

```
package Utility is
   pragma Assertion_Policy (Pre => Check, Post => Ignore);

   subtype Index_Type is Natural range 0 .. 1023;
   type Integer_Array is array(Index_Type) of Integer;

   function Search_For_Zero (Values : in Integer_Array) return Index_Type
     with
       Pre => (for some Index in Index_Type => Values (Index) = 0),
       Post => Values (Search_For_Zero'Result) = 0;

end Utility;
```

Care is required whenever explicitly setting the assertion policy to Ignore. For example, if a change is made to the body of preceding package Utility and not all verification conditions are discharged after the change, you could easily end up with the case in which a postcondition is not satisfied and not checked. However, the SPARK tools always generate verification conditions and

attempt to discharge them regardless of the assertion policy setting. Thus, static verification is unaffected by the Assertion_Policy pragma.

Summary

- A logical error is an error in the program's logic as a result of programmer oversight. A runtime error is a special kind of logical error that is detected by Ada-mandated runtime checks.
- An assertion is a programmer-defined check. All assertions are executable under the control of the assertion policy in force when a unit is compiled. A policy of Check enables assertions. A policy of Ignore disables them. If an assertion fails during execution, Assertion_Error is raised.
- The SPARK tools attempt to prove that a program is free of runtime errors and, in addition, all programmer supplied assertions will always be satisfied.
- SPARK eliminates Program_Error and Tasking_Error by prohibiting the features that might raise them.
- SPARK does not eliminate the possibility of Storage_Error, but it simplifies the analysis required to eliminate it.
- SPARK eliminates Constraint_Error as part of proving freedom of runtime errors.
- At each point where a check is required, either as mandated by the Ada standard or as added by the programmer in the form of an assertion, the SPARK tools generate a verification condition that, if proved, shows the check will never fail.
- A precondition is an assertion attached to a subprogram that must hold whenever the subprogram is called. It can be used to constrain the inputs to a subprogram beyond the constraints imposed by the type system. In particular, preconditions can describe required relationships between inputs.
- A postcondition is an assertion attached to a subprogram that must hold whenever the subprogram returns. It describes the effect of the subprogram and is part of the subprogram's functional specification.
- In a postcondition, you can use the 'Old attribute to reference the value of an input when the subprogram is first entered. You can use the 'Result attribute to reference the return value of a function.
- The Initial_Condition aspect can be used on a package to specify the state the package has after elaboration. It serves as a kind of package-wide post-condition and is most useful for packages that have internal state.
- Assertions in the visible part of a package cannot directly reference information in the private section of a package. It is thus sometimes necessary to define and use public functions in such assertions.

- The Refined_Post aspect can be used in the body of a package to describe a subprogram's postcondition in internal terms. SPARK attempts to prove that the public postcondition, if any, follows from the subprogram's precondition and refined postcondition.
- Expression functions are automatically given a refined postcondition that asserts they return the value of the expression used to define them. As a result, expression functions can be thought of as pure specification.
- It is common to define functions in public assertions as inline expression functions in the private section of a package specification.
- A type invariant is an assertion attached to private type that must hold whenever a subprogram manipulating an object of that type returns. They can be approximately thought of as postconditions that are automatically applied to all subprograms manipulating the type.
- Type invariants are currently not supported by SPARK. However, they are part of Ada and can still be used. The SPARK tools will simply not (yet) attempt to prove they never fail.
- Subtype predicates allow you to specify complex constraints on a subtypes. They can be used to specify types more precisely. SPARK currently does not support dynamic predicates but does support static predicates.
- Contract_Cases gives you a way of specifying a collection of pre- and postconditions in a convenient, easy-to-maintain way. It is most applicable when the input domain of a subprogram can be partitioned into disjoint subdomains. SPARK proves that all the cases are mutually exclusive and that they cover the entire input domain.
- Because assertions are executable, there is a possibility they might cause a runtime error when evaluated. The SPARK tools also generate verification conditions to show this will not happen.
- Runtime errors in the assertions can be avoided by adjusting the overflow mode of the GNAT compiler.
- The Assert pragma allows you to inject arbitrary checks into the body of a subprogram. The SPARK tools attempt to prove every Assert is always true, and it uses the asserted information to help with following proofs. The Assert pragma lets you give "hints" to the tools to simplify later proofs.
- The Assert_And_Cut pragma is like the Assert pragma except that it also introduces a cut point. All paths that reach the Assert_And_Cut are terminated. Only a single path leaves the Assert_And_Cut. This is useful for reducing the total number of paths in a subprogram to simplify verification conditions and speed up processing.
- The Assume pragma introduces information for the SPARK tools to use but does not carry any proof obligations itself. It can be used to encode

information external to the program that the tools nevertheless need to complete reasonable proofs.

- The Assume pragma should be used carefully. If the assumption is false, the SPARK tools may end up proving false things. However, Assume is executed as usual so testing may uncover false assumptions.
- A loop invariant is an assertion added to the body of the loop that asserts a condition that is true for every loop iteration. It serves as a cut point and thus prevents the loop from creating a potentially infinite number of paths.
- Despite being a cut point, the SPARK tools give special handling to values that are not changed inside the loop. Information about those values do not need to be reasserted by the loop invariant.
- Choosing loop invariants can be tricky. You need to find a condition that is readily proved on loop entry and for each iteration of the loop and that also provides enough information to prove other verification conditions inside and beyond the loop.
- If the SPARK tools successfully prove a subprogram's postcondition, it only means the postcondition is honored if the subprogram actually returns.
- The Loop_Variant pragma allows you to specify one or more expressions that always increase or always decrease as a loop executes. The expressions must have a discrete type so if the loop variant is true it implies the loop must terminate eventually.
- It is appropriate to use Loop_Variant in **while** loops or in loops constructed with a bare **loop** reserved word.
- SPARK handles discriminated types similarly to the way it handles unconstrained array types. The tools generate verification conditions related to the discriminants as appropriate, and the discriminants can be used in assertions in a natural way.
- The SPARK tools do not analyze generic code directly but instead analyze each instantiation of a generic separately. Some instantiations might prove fully, whereas others might not even be legal SPARK.
- When the SPARK tools analyze a generic instantiation, they use information about the actual generic parameters involved. Some instantiations might be difficult to prove because of the limited information available about the types and operations used.
- In fully proved code, it may be desirable to suppress runtime checks and assertion checks to improve the efficiency of the program. Assertion checks, especially, may warrant suppression because assertions have the potential of slowing down execution asymptotically.

Exercises

6.1 Alice is interested in using a package Bob created. Which of them wants the public subprograms in that package to have strong preconditions? Which of them wants strong postconditions?

6.2 Suppose there is a function File_Exists that takes a string and returns True if a file of the given name is available to your program. Now consider a procedure that reads a configuration file declared as follows:

```
procedure Read_Configuration (String : in File_Name)
  with Pre => File_Exists (File_Name);
```

This is an suspicious use of contracts. Why?

6.3 What is a *cut point* and under what circumstances would it be appropriate to consider introducing one into your code? How would you add a cut point to your code?

6.4 Suppose you wanted to create a subprogram that accepted only even nonnegative integers. You could either use a precondition as follows,

```
procedure Example (X : Natural)
  with Pre => (X mod 2 = 0);
```

or you could use a subtype predicate as follows,

```
subtype Even is Natural
  with Dynamic_Predicate => Even mod 2 = 0;

procedure Example (X : Even);
```

Discuss the relative merits of these two approaches with respect to SPARK programming. Is your answer different if the subtype could be defined using a static predicate instead?

6.5 Type invariants are not supported by SPARK at the time of this writing. However, to a large degree this can be worked around. How?

6.6 Explain how contract cases can be considered a combination of preconditions and postconditions. Is the Contract_Cases aspect strictly necessary, or is it always possible to express contract cases using pre- and postconditions?

6.7 Write a procedure that increments every counter in an array of counters. Prove your procedure free of runtime errors. You will need a precondition to assert that all the counters are not yet at their maximal values. You may also need a suitable loop invariant.

6.8 If one says informally that condition C_1 is "stronger" than condition C_2 (where a condition is a boolean expression), what is the intended formal relationship between C_1 and C_2?

6.9 Consider the specification of package Buffers on page 203. Add the following subprograms to that package. For each subprogram write a suitable postcondition, implement the subprogram, and prove your implementation correct with respect to your postcondition.

a. Add a procedure Reverse_Buffer with a declaration as follows:

```
procedure Reverse_Buffer (Buffer : in out Buffer_Type);
```

The procedure reverses the contents of Buffer. For example, reversing the string "Hello" results in "olleH".

b. Add a procedure Rotate_Right with a declaration as follows:

```
procedure Rotate_Right (Buffer    : in out Buffer_Type;
                        Distance  : in     Buffer_Count_Type);
```

The procedure moves the contents of Buffer toward higher index values (to the right) by an amount Distance. Any elements that "fall off the end" are brought back to the beginning. For example, rotating the string "Hello" to the right by three results in "lloHe".

c. Add a function Search with a declaration as follows:

```
function Search (Haystack : Buffer_Type;
                 Needle   : String) return Buffer_Count_Type;
```

The function returns the index in Haystack where Needle first appears or zero if Needle does not appear. It is permitted for Needle to be longer than Haystack in which case it can never be found.

d. Add a procedure Count_And_Erase_Character with a declaration as follows:

```
procedure Count_And_Erase_Character
  (Buffer : in out Buffer_Type;
   Ch     : in     Character;
   Count  : out    Buffer_Count_Type)
```

The procedure returns in Count the number of times Ch occurs in Buffer. It also modifies Buffer so that each occurrence of Ch is replaced with a space.

e. Add a procedure Compact with a declaration as follows:

```
procedure Compact (Buffer          : in out Buffer_Type;
                   Erase_Character : in     Character;
                   Fill_Character  : in     Character;
                   Valid           :    out Buffer_Count_Type)
```

The procedure compacts Buffer by removing occurrences of Erase_Character. Free space opened at the end of the buffer is filled with instances of Fill_Character. On returning, the value of Valid is a count of the number of original characters that were not erased.

6.10 Let the specification of the Buffers package on page 203 be modified to use an unconstrained array type as follows:

```
package Buffers_Unconstrained is
   subtype Buffer_Count_Type is Natural;
   subtype Buffer_Index_Type is Positive;
   type Buffer_Type is array(Buffer_Index_Type range <>) of Character;
   ...
end Buffers_Unconstrained;
```

Repeat Exercise 6.9.

6.11 Loop variant pragmas are not needed for many loops in SPARK programs. Under what circumstances is it appropriate to consider using a loop variant?

6.12 When a loop variant pragma is used with multiple expressions, it is possible that later expressions might "go the wrong way" as long as an earlier expression changes in the right direction. In particular, a later expression might get reset to some initial value. Explain why the loop must still eventually terminate despite this behavior. How does this relate to nested loops?

6.13 The precondition of Binary_Search does not actually need to state that the input array is sorted. A weaker, but still adequate, precondition is that the array must be partitioned by both the expressions E < Search_Item and E <= Search_Item, where E is an array element. An array is partitioned by an expression if there exists an index i such that the expression is true for every element before i and false for every element at or above i. For example, if the search item is 10, the array containing (5, 3, 10, 10, 8, 6) is suitably partitioned.

Modify the precondition on Binary_Search shown on page 167 to use this weaker condition. Does the implementation given on page 214 still satisfy the postcondition?

6.14 Consider the In_Unit_Square function on page 185. Following the style of that function, show the specification of an In_Unit_Circle function taking Float parameters and that returns +1 if the given point is in a circle with radius one centered on the origin.

6.15 The generic package Double_List discussed in Section 6.7 is incomplete. Make the package usable by adding subprograms for iterator movement

and for reading and updating list elements via an iterator. Also add a Front function that returns an iterator to the first element of the list (or the sentinal if the list is empty). Add some procedures to package List_Handler in Section 6.7 to demonstrate these new capabilities. Ensure that SPARK proves your revised List_Handler free of runtime errors.

6.16 Extend Exercise 6.15 by revising package Double_List again so that it is a type package rather than a variable package. Let the maximum size of each list be given by a discriminant. Call your revised package Double_Lists with a private type List. Revise the List_Handler package to use your new package. Ensure that SPARK proves your revised List_Handler free of runtime errors.

6.17 Why is it more important to suppress assertions than runtime checks in deployed programs?