LEARN.
ADACORE.COM

Introduction to SPARK

Release 2020-05

Claire Dross and Yannick Moy

Oct 05, 2020

CONTENTS:

This tutorial is an interactive introduction to the SPARK programming language and its formal verification tools. You will learn the difference between Ada and SPARK and how to use the various analysis tools that come with SPARK.

This document was prepared by Claire Dross and Yannick Moy.

[1] http://creativecommons.org/licenses/by-sa/4.0

CHAPTER

ONE

SPARK OVERVIEW

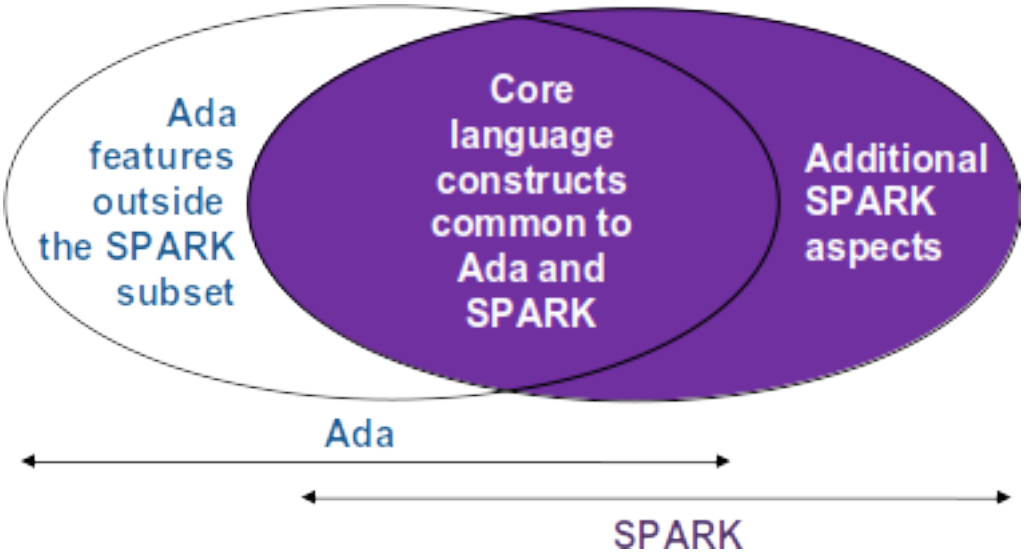
This tutorial is an introduction to the SPARK programming language and its formal verification tools. You need not know any specific programming language (although going over the Introduction to Ada course[2] first may help) or have experience in formal verification.

1.1 What is it?

SPARK refers to two different things:

- a programming language targeted at functional specification and static verification, and
- a set of development and verification tools for that language.

The SPARK language is based on a subset of the Ada language. Ada is particularly well suited to formal verification since it was designed for critical software development. SPARK builds on that foundation.



Version 2012 of Ada introduced the use of *aspects*, which can be used for subprogram contracts, and version 2014 of SPARK added its own aspects to further aid static analysis.

1.2 What do the tools do?

We start by reviewing static verification of programs, which is verification of the source code performed without compiling or executing it. Verification uses tools that perform static analysis. These

[2] https://learn.adacore.com/courses/intro-to-ada/index.html

can take various forms. They include tools that check types and enforce visibility rules, such as the compiler, in addition to those that perform more complex reasoning, such as abstract interpretation, as done by a tool like CodePeer[3] from AdaCore. The tools that come with SPARK perform two different forms of static analysis:

- *flow analysis* is the fastest form of analysis. It checks initializations of variables and looks at data dependencies between inputs and outputs of subprograms. It can also find unused assignments and unmodified variables.
- *proof* checks for the absence of runtime errors as well as the conformance of the program with its specifications.

1.3 Key Tools

The tool for formal verification of the SPARK language is called *GNATprove*. It checks for conformance with the SPARK subset and performs flow analysis and proof of the source code. Several other tools support the SPARK language, including both the GNAT compiler[4] and the GPS integrated development environment[5].

1.4 A trivial example

We start with a simple example of a subprogram in Ada that uses SPARK aspects to specify verifiable subprogram contracts. The subprogram, called `Increment`, adds 1 to the value of its parameter X:

```
procedure Increment
  (X : in out Integer)
with
  Global  => null,
  Depends => (X => X),
  Pre     => X < Integer'Last,
  Post    => X = X'Old + 1;
```

```
procedure Increment
  (X : in out Integer)
is
begin
  X := X + 1;
end Increment;
```

The contracts are written using the Ada *aspect* feature and those shown specify several properties of this subprogram:

- The SPARK Global aspect says that `Increment` does not read or write any global variables.
- The SPARK Depend aspect is especially interesting for security: it says that the value of the parameter X after the call depends only on the (previous) value of X.
- The `Pre` and `Post` aspects of Ada specify functional properties of `Increment`:
  - `Increment` is only allowed to be called if the value of X prior to the call is less than `Integer'Last`. This ensures that the addition operation performed in the subprogram body doesn't overflow.

[3] https://www.adacore.com/codepeer
[4] https://www.adacore.com/gnatpro
[5] https://www.adacore.com/gnatpro/toolsuite/gps

- `Increment` does indeed perform an increment of X: the value of X after a call is one greater than its value before the call.

GNATprove can verify all of these contracts. In addition, it verifies that no error can be raised at runtime when executing `Increment`'s body.

1.5 The Programming Language

It's important to understand why there are differences between the SPARK and Ada languages. The aim when designing the SPARK subset of Ada was to create the largest possible subset of Ada that was still amenable to simple specification and sound verification.

The most notable restrictions from Ada are related to exceptions and access types, both of which are known to considerably increase the amount of user-written annotations required for full support. Goto statements and controlled types are also not supported since they introduce non-trivial control flow. The two remaining restrictions relate to side-effects in expressions and aliasing of names, which we now cover in more detail.

1.6 Limitations

1.6.1 No side-effects in expressions

The SPARK language doesn't allow side-effects in expressions. In other words, evaluating a SPARK expression must not update any object. This limitation is necessary to avoid unpredictable behavior that depends on order of evaluation, parameter passing mechanisms, or compiler optimizations. The expression for `Dummy` below is non-deterministic due to the order in which the two calls to F are evaluated. It's therefore not legal SPARK.

```
procedure Show_Illegal_Ada_Code is

   function F (X : in out Integer) return Integer is
      Tmp : constant Integer := X;
   begin
      X := X + 1;
      return Tmp;
   end F;

   Dummy : Integer := 0;

begin
   Dummy := F (Dummy) - F (Dummy); -- ??
end Show_Illegal_Ada_Code;
```

In fact, the code above is not even legal Ada, so the same error is generated by the GNAT compiler. But SPARK goes further and GNATprove also produces an error for the following equivalent code that is accepted by the Ada compiler:

```
procedure Show_Illegal_SPARK_Code is

   Dummy : Integer := 0;

   function F return Integer is
      Tmp : constant Integer := Dummy;
   begin
      Dummy := Dummy + 1;
      return Tmp;
```

(continues on next page)

(continued from previous page)

```
   end F;

begin
   Dummy := F - F; -- ??
end Show_Illegal_SPARK_Code;
```

The SPARK languages enforces the lack of side-effects in expressions by forbidding side-effects in functions, which include modifications to either parameters or global variables. As a consequence, SPARK forbids functions with `out` or `in out` parameters in addition to functions modifying a global variable. Function `F` below is illegal in SPARK, while Function `Incr` might be legal if it doesn't modify any global variables and function `Incr_And_Log` might be illegal if it modifies global variables to perform logging.

```
function F (X : in out Integer) return Integer;      -- Illegal

function Incr (X : Integer) return Integer;          -- OK?

function Incr_And_Log (X : Integer) return Integer; -- OK?
```

In most cases, you can easily replace these functions by procedures with an `out` parameter that returns the computed value.

When it has access to function bodies, GNATprove verifies that those functions are indeed free from side-effects. Here for example, the two functions `Incr` and `Incr_And_Log` have the same signature, but only `Incr` is legal in SPARK. `Incr_And_Log` isn't: it attempts to update the global variable `Call_Count`.

```
package Side_Effects is

   function Incr (X : Integer) return Integer;          -- OK?

   function Incr_And_Log (X : Integer) return Integer; -- OK?

end Side_Effects;
```

```
package body Side_Effects is

   function Incr (X : Integer) return Integer
   is (X + 1); -- OK

   Call_Count : Natural := 0;

   function Incr_And_Log (X : Integer) return Integer is
   begin
      Call_Count := Call_Count + 1; -- Illegal
      return X + 1;
   end Incr_And_Log;

end Side_Effects;
```

1.6.2 No aliasing of names

Another restriction imposed by the SPARK subset concerns aliasing[6]. We say that two names are *aliased* if they refer to the same object. There are two reasons why aliasing is forbidden in SPARK:

- It makes verification more difficult because it requires taking into account the fact that modifications to variables with different names may actually update the same object.

[6] https://en.wikipedia.org/wiki/Aliasing_(computing)

- Results may seem unexpected from a user point of view. The results of a subprogram call may depend on compiler-specific attributes, such as parameter passing mechanisms, when its parameters are aliased.

Aliasing can occur as part of the parameter passing that occurs in a subprogram call. Functions have no side-effects in SPARK, so aliasing of parameters in function calls isn't problematic; we need only consider procedure calls. When a procedure is called, SPARK verifies that no `out` or `in out` parameter is aliased with either another parameter of the procedure or a global variable modified in the procedure's body.

Procedure `Move_To_Total` is an example where the possibility of aliasing wasn't taken into account by the programmer:

```
procedure No_Aliasing is

   Total : Natural := 0;

   procedure Move_To_Total (Source : in out Natural)
     with Post => Total = Total'Old + Source'Old and Source = 0
   is
   begin
      Total  := Total + Source;
      Source := 0;
   end Move_To_Total;

   X : Natural := 3;

begin
   Move_To_Total (X);          -- OK
   pragma Assert (Total = 3); -- OK
   Move_To_Total (Total);      -- flow analysis error
   pragma Assert (Total = 6); -- runtime error
end No_Aliasing;
```

`Move_To_Total` adds the value of its input parameter `Source` to the global variable `Total` and then resets `Source` to 0. The programmer has clearly not taken into account the possibility of an aliasing between `Total` and `Source`. (This sort of error is quite common.)

This procedure itself is valid SPARK. When doing verification, GNATprove assumes, like the programmer did, that there's no aliasing between `Total` and `Source`. To ensure this assumption is valid, GNATprove checks for possible aliasing on every call to `Move_To_Total`. Its final call in procedure `No_Aliasing` violates this assumption, which produces both a message from GNATprove and a runtime error (an assertion violation corresponding to the expected change in `Total` from calling `Move_To_Total`). Note that the postcondition of `Move_To_Total` is not violated on this second call since integer parameters are passed by copy and the postcondition is checked before the copy-back from the formal parameters to the actual arguments.

Aliasing can also occur as a result of using access types (pointers[7] in Ada). These are restricted in SPARK so that only benign aliasing is allowed, when both names are only used to read the data. In particular, assignment between access objects operates a transfer of ownership, where the source object loses its permission to read or write the underlying allocated memory.

Procedure `Ownership_Transfer` is an example of code that is legal in Ada but rejected in SPARK due to aliasing:

```
procedure Ownership_Transfer is
   type Int_Ptr is access Integer;
   X     : Int_Ptr;
   Y     : Int_Ptr;
   Dummy : Integer;
begin
```

(continues on next page)

[7] https://en.m.wikipedia.org/wiki/Pointer_(computer_programming)

(continued from previous page)

```
   X     := new Integer'(1);
   X.all := X.all + 1;
   Y     := X;
   Y.all := Y.all + 1;
   X.all := X.all + 1;  --  illegal
   X.all := 1;          --  illegal
   Dummy := X.all;      --  illegal
end Ownership_Transfer;
```

After the assignment of X to Y, variable X cannot be used anymore to read or write the underlying allocated memory.

1.7 Designating SPARK Code

Since the SPARK language is restricted to only allow easily specifiable and verifiable constructs, there are times when you can't or don't want to abide by these limitations over your entire code base. Therefore, the SPARK tools only check conformance to the SPARK subset on code which you identify as being in SPARK.

You do this by using an aspect named SPARK_Mode. If you don't explicitly specify otherwise, SPARK_Mode is *Off*, meaning you can use the complete set of Ada features in that code and that it should not be analyzed by GNATprove. You can change this default either selectively (on some units or subprograms or packages inside units) or globally (using a configuration pragma, which is what we're doing in this tutorial). To allow simple reuse of existing Ada libraries, entities declared in imported units with no explicit SPARK_Mode can still be used from SPARK code. The tool only checks for SPARK conformance on the declaration of those entities which are actually used within the SPARK code.

Here's a common case of using the SPARK_Mode aspect:

```
package P
  with SPARK_Mode => On
is
   -- package spec is IN SPARK, so can be used by SPARK clients
end P;

package body P
  with SPARK_Mode => Off
is
   -- body is NOT IN SPARK, so is ignored by GNATprove
end P;
```

The package P only defines entities whose specifications are in the SPARK subset. However, it wants to use all Ada features in its body. Therefore the body should not be analyzed and has its SPARK_Mode aspect set to *Off*.

You can specify SPARK_Mode in a fine-grained manner on a per-unit basis. An Ada package has four different components: the visible and private parts of its specification and the declarative and statement parts of its body. You can specify SPARK_Mode as being either *On* or *Off* on any of those parts. Likewise, a subprogram has two parts: its specification and its body.

A general rule in SPARK is that once SPARK_Mode has been set to *Off*, it can never be switched *On* again in the same part of a package or subprogram. This prevents setting SPARK_Mode to *On* for subunits of a unit with SPARK_Mode *Off* and switching back to SPARK_Mode *On* for a part of a given unit where it was set fo *Off* in a previous part.

1.8 Code Examples / Pitfalls

1.8.1 Example #1

Here's a package defining an abstract stack type (defined as a private type in SPARK) of `Element` objects along with some subprograms providing the usual functionalities of stacks. It's marked as being in the SPARK subset.

```
package Stack_Package
  with SPARK_Mode => On
is
   type Element is new Natural;
   type Stack is private;

   function Empty return Stack;
   procedure Push (S : in out Stack; E : Element);
   function Pop (S : in out Stack) return Element;

private
   type Stack is record
      Top : Integer;
      --  ...
   end record;

end Stack_Package;
```

Side-effects in expressions are not allowed in SPARK. Therefore, `Pop` is not allowed to modify its parameter `S`.

1.8.2 Example #2

Let's turn to an abstract state machine version of a stack, where the unit provides a single instance of a stack. The content of the stack (global variables `Content` and `Top`) is not directly visible to clients. In this stripped-down version, only the function `Pop` is available to clients. The package spec and body are marked as being in the SPARK subset.

```
package Global_Stack
  with SPARK_Mode => On
is
   type Element is new Integer;

   function Pop return Element;

end Global_Stack;
```

```
package body Global_Stack
  with SPARK_Mode => On
is
   Max : constant Natural := 100;
   type Element_Array is array (1 .. Max) of Element;

   Content : Element_Array;
   Top     : Natural;

   function Pop return Element is
      E : constant Element := Content (Top);
   begin
      Top := Top - 1;
```

(continues on next page)

(continued from previous page)

```
      return E;
   end Pop;

end Global_Stack;
```

As above, functions should be free from side-effects. Here, `Pop` updates the global variable `Top`, which is not allowed in SPARK.

1.8.3 Example #3

We now consider two procedures: `Permute` and `Swap`. `Permute` applies a circular permutation to the value of its three parameters. `Swap` then uses `Permute` to swap the value of X and Y.

```
package P
  with SPARK_Mode => On
is
   procedure Permute (X, Y, Z : in out Positive);
   procedure Swap (X, Y : in out Positive);
end P;
```

```
package body P
  with SPARK_Mode => On
is
   procedure Permute (X, Y, Z : in out Positive) is
      Tmp : constant Positive := X;
   begin
      X := Y;
      Y := Z;
      Z := Tmp;
   end Permute;

   procedure Swap (X, Y : in out Positive) is
   begin
      Permute (X, Y, Y);
   end Swap;
end P;
```

```
with P; use P;

procedure Test_Swap
  with SPARK_Mode => On
is
   A : Integer := 1;
   B : Integer := 2;
begin
   Swap (A, B);
end Test_Swap;
```

Here, the values for parameters Y and Z are aliased in the call to `Permute`, which is not allowed in SPARK. In fact, in this particular case, this is even a violation of Ada rules so the same error is issued by the Ada compiler.

In this example, we see the reason why aliasing is not allowed in SPARK: since Y and Z are `Positive`, they are passed by copy and the result of the call to `Permute` depends on the order in which they're copied back after the call.

1.8.4 Example #4

Here, the Swap procedure is used to swap the value of the two record components of R.

```
package P
  with SPARK_Mode => On
is
   type Rec is record
      F1 : Positive;
      F2 : Positive;
   end record;

   procedure Swap_Fields (R : in out Rec);
   procedure Swap (X, Y : in out Positive);
end P;
```

```
package body P
  with SPARK_Mode => On
is
   procedure Swap (X, Y : in out Positive) is
      Tmp : constant Positive := X;
   begin
      X := Y;
      Y := Tmp;
   end Swap;

   procedure Swap_Fields (R : in out Rec) is
   begin
      Swap (R.F1, R.F2);
   end Swap_Fields;

end P;
```

This code is correct. The call to Swap is safe: two different components of the same record can't refer to the same object.

1.8.5 Example #5

Here's a slight modification of the previous example using an array instead of a record: Swap_Indexes calls Swap on values stored in the array A.

```
package P
  with SPARK_Mode => On
is
   type P_Array is array (Natural range <>) of Positive;

   procedure Swap_Indexes (A : in out P_Array; I, J : Natural);
   procedure Swap (X, Y : in out Positive);
end P;
```

```
package body P
  with SPARK_Mode => On
is
   procedure Swap (X, Y : in out Positive) is
      Tmp : constant Positive := X;
   begin
      X := Y;
      Y := Tmp;
   end Swap;
```

(continues on next page)

(continued from previous page)

```
   procedure Swap_Indexes (A : in out P_Array; I, J : Natural) is
   begin
      Swap (A (I), A (J));
   end Swap_Indexes;

end P;
```

GNATprove detects a possible case of aliasing. Unlike the previous example, it has no way of knowing that the two elements `A (I)` and `A (J)` are actually distinct when we call `Swap`. GNATprove issues a check message here instead of an error, giving you the possibility of justifying the message after review (meaning that you've verified manually that this can't, in fact, occur).

1.8.6 Example #6

We now consider a package declaring a type `Dictionary`, an array containing a word per letter. The procedure `Store` allows us to insert a word at the correct index in a dictionary.

```
package P
  with SPARK_Mode => On
is
   subtype Letter is Character range 'a' .. 'z';
   type String_Access is access all String;
   type Dictionary is array (Letter) of String_Access;

   procedure Store (D : in out Dictionary; W : String);
end P;
```

```
package body P
  with SPARK_Mode => On
is
   procedure Store (D : in out Dictionary; W : String) is
      First_Letter : constant Letter := W (W'First);
   begin
      D (First_Letter) := new String'(W);
   end Store;
end P;
```

This code is not correct: general access types are not part of the SPARK subset. Note that we could use here a pool-specific access type for `String_Access` by removing the keyword `all` in its definition. In the case where it's necessary to keep a general access type (for example to be able to store pointers to variables on the stack), another solution here is to use `SPARK_Mode` to separate the definition of `String_Access` from the rest of the code in a fine grained manner.

1.8.7 Example #7

Here's a new version of the previous example, which we've modified to hide the general access type inside the private part of package `P`, using `pragma SPARK_Mode (Off)` at the start of the private part.

```
package P
  with SPARK_Mode => On
is
   subtype Letter is Character range 'a' .. 'z';
   type String_Access is private;
   type Dictionary is array (Letter) of String_Access;
```

(continues on next page)

(continued from previous page)

```
   function New_String_Access (W : String) return String_Access;

   procedure Store (D : in out Dictionary; W : String);

private
   pragma SPARK_Mode (Off);

   type String_Access is access all String;

   function New_String_Access (W : String) return String_Access is
     (new String'(W));
end P;
```

Since the general access type is defined and used inside of a part of the code ignored by GNATprove, this code is correct.

1.8.8 Example #8

Let's put together the new spec for package P with the body of P seen previously.

```
package P
  with SPARK_Mode => On
is
   subtype Letter is Character range 'a' .. 'z';
   type String_Access is private;
   type Dictionary is array (Letter) of String_Access;

   function New_String_Access (W : String) return String_Access;

   procedure Store (D : in out Dictionary; W : String);

private
   pragma SPARK_Mode (Off);

   type String_Access is access all String;

   function New_String_Access (W : String) return String_Access is
     (new String'(W));
end P;
```

```
package body P
  with SPARK_Mode => On
is
   procedure Store (D : in out Dictionary; W : String) is
      First_Letter : constant Letter := W (W'First);
   begin
      D (First_Letter) := New_String_Access (W);
   end Store;
end P;
```

The body of `Store` doesn't actually use any construct that's not in the SPARK subset, but we nevertheless can't set `SPARK_Mode` to `On` for P's body because it has visibility to P's private part, which is not in SPARK, even if we don't use it.

1.8.9 Example #9

Next, we moved the declaration and the body of the procedure `Store` to another package named Q.

```
package P
  with SPARK_Mode => On
is
   subtype Letter is Character range 'a' .. 'z';
   type String_Access is private;
   type Dictionary is array (Letter) of String_Access;

   function New_String_Access (W : String) return String_Access;

private
   pragma SPARK_Mode (Off);

   type String_Access is access all String;

   function New_String_Access (W : String) return String_Access is
     (new String'(W));
end P;
```

```
with P; use P;
package Q
  with SPARK_Mode => On
is
   procedure Store (D : in out Dictionary; W : String);
end Q;
```

```
package body Q
  with SPARK_Mode => On
is
   procedure Store (D : in out Dictionary; W : String)  is
      First_Letter : constant Letter := W (W'First);
   begin
      D (First_Letter) := New_String_Access (W);
   end Store;
end Q;
```

And now everything is fine: we've managed to retain the use of the access type while having most of our code in the SPARK subset so GNATprove is able to analyze it.

1.8.10 Example #10

Our final example is a package with two functions to search for the value 0 inside an array A. The first raises an exception if 0 isn't found in A while the other simply returns 0 in that case.

```
package P
  with SPARK_Mode => On
is
   type N_Array is array (Positive range <>) of Natural;
   Not_Found : exception;

   function Search_Zero_P (A : N_Array) return Positive;

   function Search_Zero_N (A : N_Array) return Natural;
end P;
```

```
package body P
  with SPARK_Mode => On
is
   function Search_Zero_P (A : N_Array) return Positive is
   begin
      for I in A'Range loop
         if A (I) = 0 then
            return I;
         end if;
      end loop;
      raise Not_Found;
   end Search_Zero_P;

   function Search_Zero_N (A : N_Array) return Natural
     with SPARK_Mode => Off is
   begin
      return Search_Zero_P (A);
   exception
      when Not_Found => return 0;
   end Search_Zero_N;
end P;
```

This code is perfectly correct, despite the use of exception handling, because we've carefully isolated this non-SPARK feature in a function body marked with a `SPARK_Mode` of `Off` so it's ignored by GNATprove. However, GNATprove tries to show that `Not_Found` is never raised in `Search_Zero_P`, producing a message about a possible exception being raised. Looking at `Search_Zero_N`, it's indeed likely that an exception is meant to be raised in some cases, which means you need to verify that `Not_Found` is only raised when appropriate using other methods such as peer review or testing.

CHAPTER

TWO

FLOW ANALYSIS

In this section we present the flow analysis capability provided by the GNATprove tool, a critical tool for using SPARK.

2.1 What does flow analysis do?

Flow analysis concentrates primarily on variables. It models how information flows through them during a subprogram's execution, connecting the final values of variables to their initial values. It analyzes global variables declared at library level, local variables, and formal parameters of subprograms.

Nesting of subprograms creates what we call *scope variables*: variables declared locally to an enclosing unit. From the perspective of a nested subprogram, scope variables look very much like global variables

Flow analysis is usually fast, roughly as fast as compilation. It detects various types of errors and finds violations of some SPARK legality rules, such as the absence of aliasing and freedom of expressions from side-effects. We discussed these rules in the *SPARK Overview* (page 3).

Flow analysis is *sound*: if it doesn't detect any errors of a type it's supposed to detect, we know for sure there are no such errors.

2.2 Errors Detected

2.2.1 Uninitialized Variables

We now present each class of errors detected by flow analysis. The first is the reading of an uninitialized variable. This is nearly always an error: it introduces non-determinism and breaks the type system because the value of an uninitialized variable may be outside the range of its subtype. For these reasons, SPARK requires every variable to be initialized before being read.

Flow analysis is responsible for ensuring that SPARK code always fulfills this requirement. For example, in the function `Max_Array` shown below, we've neglected to initialize the value of `Max` prior to entering the loop. As a consequence, the value read by the condition of the if statement may be uninitialized. Flow analysis detects and reports this error.

```
package Show_Uninitialized is

   type Array_Of_Naturals is array (Integer range <>) of Natural;

   function Max_Array (A : Array_Of_Naturals) return Natural;

end Show_Uninitialized;
```

```
package body Show_Uninitialized is

   function Max_Array (A : Array_Of_Naturals) return Natural is
      Max : Natural;
   begin
      for I in A'Range loop
         if A (I) > Max then -- Here Max may not be initialized
            Max := A (I);
         end if;
      end loop;
      return Max;
   end Max_Array;

end Show_Uninitialized;
```

2.2.2 Ineffective Statements

Ineffective statements are different than dead code: they're executed, and often even modify the value of variables, but have no effect on any of the subprogram's visible outputs: parameters, global variables or the function result. Ineffective statements should be avoided because they make the code less readable and more difficult to maintain.

More importantly, they're often caused by errors in the program: the statement may have been written for some purpose, but isn't accomplishing that purpose. These kinds of errors can be difficult to detect in other ways.

For example, the subprograms `Swap1` and `Swap2` shown below don't properly swap their two parameters `X` and `Y`. This error caused a statement to be ineffective. That ineffective statement is not an error in itself, but flow analysis produces a warning since it can be indicative of an error, as it is here.

```
package Show_Ineffective_Statements is

   type T is new Integer;

   procedure Swap1 (X, Y : in out T);
   procedure Swap2 (X, Y : in out T);

end Show_Ineffective_Statements;
```

```
package body Show_Ineffective_Statements is

   procedure Swap1 (X, Y : in out T) is
      Tmp : T;
   begin
      Tmp := X; -- This statement is ineffective
      X   := Y;
      Y   := X;
   end Swap1;

   Tmp : T := 0;

   procedure Swap2 (X, Y : in out T) is
      Temp : T := X; -- This variable is unused
   begin
      X := Y;
      Y := Tmp;
   end Swap2;

end Show_Ineffective_Statements;
```

So far, we've seen examples where flow analysis warns about ineffective statements and unused variables.

2.2.3 Incorrect Parameter Mode

Parameter modes are an important part of documenting the usage of a subprogram and affect the code generated for that subprogram. Flow analysis checks that each specified parameter mode corresponds to the usage of that parameter in the subprogram's body. It checks that an `in` parameter is never modified, either directly or through a subprogram call, checks that the initial value of an `out` parameter is never read in the subprogram (since it may not be defined on subprogram entry), and warn whens an `in out` parameter isn't modified or when its initial value isn't used. All of these may be signs of an error.

We see an example below. The subprogram `Swap` is incorrect and GNATprove warns about an input which isn't read:

```
package Show_Incorrect_Param_Mode is

   type T is new Integer;

   procedure Swap (X, Y : in out T);

end Show_Incorrect_Param_Mode;
```

```
package body Show_Incorrect_Param_Mode is

   procedure Swap (X, Y : in out T) is
      Tmp : T := X;
   begin
      Y := X;   -- The initial value of Y is not used
      X := Tmp; -- Y is computed to be an out parameter
   end Swap;

end Show_Incorrect_Param_Mode;
```

In SPARK, unlike Ada, you should declare an `out` parameter to be `in out` if it's not modified on every path, in which case its value may depend on its initial value. SPARK is stricter than Ada to allow more static detection of errors. This table summarizes SPARK's valid parameter modes as a function of whether reads and writes are done to the parameter.

Initial value read	Written on some path	Written on every path	Parameter mode
X			`in`
X	X		`in out`
X		X	`in out`
	X		`in out`
		X	`out`

2.3 Additional Verifications

2.3.1 Global Contracts

So far, none of the verifications we've seen require you to write any additional annotations. However, flow analysis also checks flow annotations that you write. In SPARK, you can specify the set of global and scoped variables accessed or modified by a subprogram. You do this using a contract named `Global`.

When you specify a `Global` contract for a subprogram, flow analysis checks that it's both correct and complete, meaning that no variables other than those stated in the contract are accessed or modified, either directly or through a subprogram call, and that all those listed are accessed or modified. For example, we may want to specify that the function `Get_Value_Of_X` reads the value of the global variable X and doesn't access any other global variable. If we do this through a comment, as is usually done in other languages, GNATprove can't verify that the code complies with this specification:

```
package Show_Global_Contracts is

   X : Natural := 0;

   function Get_Value_Of_X return Natural;
   --  Get_Value_Of_X reads the value of the global variable X

end Show_Global_Contracts;
```

You write global contracts as part of the subprogram specification. In addition to their value in flow analysis, they also provide useful information to users of a subprogram. The value you specify for the `Global` aspect is an aggregate-like list of global variable names, grouped together according to their mode.

In the example below, the procedure `Set_X_To_Y_Plus_Z` reads both Y and Z. We indicate this by specifying them as the value for `Input`. It also writes X, which we specify using `Output`. Since `Set_X_To_X_Plus_Y` both writes X and reads its initial value, X's mode is `In_Out`. Like parameters, if no mode is specified in a `Global` aspect, the default is `Input`. We see this in the case of the declaration of `Get_Value_Of_X`. Finally, if a subprogram, such as `Incr_Parameter_X`, doesn't reference any global variables, you set the value of the global contract to `null`.

```
package Show_Global_Contracts is

   X, Y, Z : Natural := 0;

   procedure Set_X_To_Y_Plus_Z with
     Global => (Input  => (Y, Z), --  reads values of Y and Z
                Output => X);     --  modifies value of X

   procedure Set_X_To_X_Plus_Y with
     Global => (Input  => Y,  --  reads value of Y
                In_Out => X); --  modifies value of X and
                              --  also reads its initial value

   function Get_Value_Of_X return Natural with
     Global => X;  -- reads the value of the global variable X

   procedure Incr_Parameter_X (X : in out Natural) with
     Global => null; -- do not reference any global variable

end Show_Global_Contracts;
```

2.3.2 Depends Contracts

You may also supply a `Depends` contract for a subprogram to specify dependencies between its inputs and outputs. These dependencies include not only global variables but also parameters and the function's result. When you supply a `Depends` contract for a subprogram, flow analysis checks that it's correct and complete, that is, for each dependency you list, the variable depends on those listed and on no others.

For example, you may want to say that the new value of each parameter of Swap, shown below, depends only on the initial value of the other parameter and that the value of X after the return of

`Set_X_To_Zero` doesn't depend on any global variables. If you indicate this through a comment, as you often do in other languages, GNATprove can't verify that this is actually the case.

```
package Show_Depends_Contracts is

   type T is new Integer;

   procedure Swap (X, Y : in out T);
   --  The value of X (resp. Y) after the call depends only
   --  on the value of Y (resp. X) before the call

   X : Natural;
   procedure Set_X_To_Zero;
   --  The value of X after the call depends on no input

end Show_Depends_Contracts;
```

Like `Global` contracts, you specify a `Depends` contract in subprogram declarations using an aspect. Its value is a list of one or more dependency relations between the outputs and inputs of the subprogram. Each relation is represented as two lists of variable names separated by an arrow. On the left of each arrow are variables whose final value depends on the initial value of the variables you list on the right.

For example, here we indicate that the final value of each parameter of `Swap` depends only on the initial value of the other parameter. If the subprogram is a function, we list its result as an output, using the `Result` attribute, as we do for `Get_Value_Of_X` below.

```
package Show_Depends_Contracts is

   type T is new Integer;

   X, Y, Z : T := 0;

   procedure Swap (X, Y : in out T) with
     Depends => (X => Y,
                 --  X depends on the initial value of Y
                 Y => X);
                 --  Y depends on the initial value of X

   function Get_Value_Of_X return T with
     Depends => (Get_Value_Of_X'Result => X);
                 --  result depends on the initial value of X

   procedure Set_X_To_Y_Plus_Z with
     Depends => (X => (Y, Z));
                 --  X depends on the initial values of Y and Z

   procedure Set_X_To_X_Plus_Y with
     Depends => (X =>+ Y);
              --  X depends on Y and X's initial value

   procedure Do_Nothing (X : T) with
     Depends => (null => X);
                 --  no output is affected by X

   procedure Set_X_To_Zero with
     Depends => (X => null);
                 --  X depends on no input

end Show_Depends_Contracts;
```

Often, the final value of a variable depends on its own initial value. You can specify this in a concise way using the + character, as we did in the specification of `Set_X_To_X_Plus_Y` above. If there's

more than one variable on the left of the arrow, a + means each variables depends on itself, not that they all depend on each other. You can write the corresponding dependency with (=> +) or without (=>+) whitespace.

If you have a program where an input isn't used to compute the final value of any output, you express that by writting `null` on the left of the dependency relation, as we did for the `Do_Nothing` subprogram above. You can only write one such dependency relation, which lists all unused inputs of the subprogram, and it must be written last. Such an annotation also silences flow analysis' warning about unused parameters. You can also write `null` on the right of a dependency relation to indicate that an output doesn't depend on any input. We do that above for the procedure `Set_X_To_Zero`.

2.4 Shortcomings

2.4.1 Modularity

Flow analysis is sound, meaning that if it doesn't output a message on some analyzed SPARK code, you can be assured that none of the errors it tests for can occur in that code. On the other hand, flow analysis often issues messages when there are, in fact, no errors. The first, and probably most common reason for this relates to modularity.

To scale flow analysis to large projects, verifications are usually done on a per-subprogram basis, including detection of uninitialized variables. To analyze this modularly, flow analysis needs to assume the initialization of inputs on subprogram entry and modification of outputs during subprogram execution. Therefore, each time a subprogram is called, flow analysis checks that global and parameter inputs are initialized and each time a subprogram returns, it checks that global and parameter outputs were modified.

This can produce error messages on perfectly correct subprograms. An example is `Set_X_To_Y_Plus_Z` below, which only sets its `out` parameter X when `Overflow` is `False`.

```
procedure Set_X_To_Y_Plus_Z
  (Y, Z     :     Natural;
   X        : out Natural;
   Overflow : out Boolean)
is
begin
   if Natural'Last - Z < Y then
      Overflow := True; -- X should be initialized on every path
   else
      Overflow := False;
      X := Y + Z;
   end if;
end Set_X_To_Y_Plus_Z;
```

The message means that flow analysis wasn't able to verify that the program didn't read an uninitialized variable. To solve this problem, you can either set X to a dummy value when there's an overflow or manually verify that X is never used after a call to `Set_X_To_Y_Plus_Z` that returned `True` as the value of `Overflow`.

2.4.2 Composite Types

Another common cause of false alarms is caused by the way flow analysis handles composite types. Let's start with arrays.

Flow analysis treats an entire array as single object instead of one object per element, so it considers modifying a single element to be a modification of the array as a whole. Obviously, this

makes reasoning about which global variables are accessed less precise and hence the dependencies of those variables are also less precise. This also affects the ability to accurately detect reads of uninitialized data.

It's sometimes impossible for flow analysis to determine if an entire array object has been initialized. For example, after we write code to initialize every element of an unconstrained array A in chunks, we may still receive a message from flow analysis claiming that the array isn't initialized. To resolve this issue, you can either use a simpler loop over the full range of the array, or (even better) an aggregate assignment, or, if that's not possible, verify initialization of the object manually.

```
package Show_Composite_Types_Shortcoming is

   type T is array (Natural range <>) of Integer;

   procedure Init_Chunks (A : out T);
   procedure Init_Loop (A : out T);
   procedure Init_Aggregate (A : out T);

end Show_Composite_Types_Shortcoming;
```

```
package body Show_Composite_Types_Shortcoming is

   procedure Init_Chunks (A : out T) is
   begin
      A (A'First) := 0;
      for I in A'First + 1 .. A'Last loop
         A (I) := 0;
      end loop;
      --  flow analysis doesn't know that A is initialized
   end Init_Chunks;

   procedure Init_Loop (A : out T) is
   begin
      for I in A'Range loop
         A (I) := 0;
      end loop;
      --  flow analysis knows that A is initialized
   end Init_Loop;

   procedure Init_Aggregate (A : out T) is
   begin
      A := (others => 0);
      --  flow analysis knows that A is initialized
   end Init_Aggregate;

end Show_Composite_Types_Shortcoming;
```

Flow analysis is more precise on record objects because it tracks the value of each component of a record separately within a single subprogram. So when a record object is initialized by successive assignments of its components, flow analysis knows that the entire object is initialized. However, record objects are still treated as single objects when analyzed as an input or output of a subprogram.

```
package Show_Record_Flow_Analysis is

   type Rec is record
      F1 : Natural;
      F2 : Natural;
   end record;

   procedure Init (R : out Rec);
```

(continues on next page)

(continued from previous page)

```
end Show_Record_Flow_Analysis;
```

```
package body Show_Record_Flow_Analysis is

   procedure Init (R : out Rec) is
   begin
      R.F1 := 0;
      R.F2 := 0;
      --  R is initialized
   end Init;

end Show_Record_Flow_Analysis;
```

Flow analysis complains when a procedure call initializes only some components of a record object. It'll notify you of uninitialized components, as we see in subprogram `Init_F2` below.

```
package Show_Record_Flow_Analysis is

   type Rec is record
      F1 : Natural;
      F2 : Natural;
   end record;

   procedure Init (R : out Rec);
   procedure Init_F2 (R : in out Rec);

end Show_Record_Flow_Analysis;
```

```
package body Show_Record_Flow_Analysis is

   procedure Init_F2
     (R : in out Rec) is
   begin
      R.F2 := 0;
   end Init_F2;

   procedure Init (R : out Rec) is
   begin
      R.F1 := 0;
      Init_F2 (R); -- R should be initialized before this call
   end Init;

end Show_Record_Flow_Analysis;
```

2.4.3 Value Dependency

Flow analysis is not value-dependent: it never reasons about the values of expressions, only whether they have been set to some value or not. As a consequence, if some execution path in a subprogram is impossible, but the impossibility can only be determined by looking at the values of expressions, flow analysis still considers that path feasible and may emit messages based on it believing that execution along such a path is possible.

For example, in the version of `Absolute_Value` below, flow analysis computes that R is uninitialized on a path that enters neither of the two conditional statements. Because it doesn't consider values of expressions, it can't know that such a path is impossible.

```
procedure Absolute_Value
  (X :     Integer;
```

(continues on next page)

(continued from previous page)

```
   R : out Natural)
is
begin
   if X < 0 then
      R := -X;
   end if;
   if X >= 0 then
      R := X;
   end if;
   --  flow analysis doesn't know that R is initialized
end Absolute_Value;
```

To avoid this problem, you should make the control flow explicit, as in this second version of `Absolute_Value`:

```
procedure Absolute_Value
  (X :     Integer;
   R : out Natural)
is
begin
   if X < 0 then
      R := -X;
   else
      R := X;
   end if;
   --  flow analysis knows that R is initialized
end Absolute_Value;
```

2.4.4 Contract Computation

The final cause of unexpected flow messages that we'll discuss also comes from inaccuracy in computations of contracts. As we explained earlier, both `Global` and `Depends` contracts are optional, but GNATprove uses their data for some of its analysis.

For example, flow analysis can't detect reads from uninitialized variables without knowing the set of variables accessed. It needs to analyze and check both the `Depends` contracts you wrote for a subprogram and those you wrote for callers of that subprogram. Since each flow contract on a subprogram depends on the flow contracts of all the subprograms called inside its body, this computation can often be quite time-consuming. Therefore, flow analysis sometimes trades-off the precision of this computation against the time a more precise computation would take.

This is the case for `Depends` contracts, where flow analysis simply assumes the worst, that each subprogram's output depends on all of that subprogram's inputs. To avoid this assumption, all you have to do is supply contracts when default ones are not precise enough. You may also want to supply `Global` contracts to further speed up flow analysis on larger programs.

2.5 Code Examples / Pitfalls

2.5.1 Example #1

The procedure `Search_Array` searches for an occurrence of element `E` in an array `A`. If it finds one, it stores the index of the element in `Result`. Otherwise, it sets `Found` to `False`.

```
package Show_Search_Array is

```

(continues on next page)

(continued from previous page)

```
   type Array_Of_Positives is array (Natural range <>) of Positive;

   procedure Search_Array
     (A      :     Array_Of_Positives;
      E      :     Positive;
      Result : out Integer;
      Found  : out Boolean);

end Show_Search_Array;
```

```
package body Show_Search_Array is

   procedure Search_Array
     (A      :     Array_Of_Positives;
      E      :     Positive;
      Result : out Integer;
      Found  : out Boolean) is
   begin
      for I in A'Range loop
         if A (I) = E then
            Result := I;
            Found  := True;
            return;
         end if;
      end loop;
      Found := False;
   end Search_Array;

end Show_Search_Array;
```

GNATprove produces a message saying that `Result` is possibly uninitialized on return. There are perfectly legal uses of the function `Search_Array`, but flow analysis detects that `Result` is not initialized on the path that falls through from the loop. Even though this program is correct, you shouldn't ignore the message: it means flow analysis cannot guarantee that `Result` is always initialized at the call site and so assumes any read of `Result` at the call site will read initialized data. Therefore, you should either initialize `Result` when `Found` is false, which silences flow analysis, or verify this assumption at each call site by other means.

2.5.2 Example #2

To avoid the message previously issued by GNATprove, we modify `Search_Array` to raise an exception when `E` isn't found in `A`:

```
package Show_Search_Array is

   type Array_Of_Positives is array (Natural range <>) of Positive;

   Not_Found : exception;

   procedure Search_Array
     (A      :     Array_Of_Positives;
      E      :     Positive;
      Result : out Integer);
end Show_Search_Array;
```

```
package body Show_Search_Array is

   procedure Search_Array
```

(continues on next page)

(continued from previous page)

```
     (A      :     Array_Of_Positives;
      E      :     Positive;
      Result : out Integer) is
   begin
      for I in A'Range loop
         if A (I) = E then
            Result := I;
            return;
         end if;
      end loop;
      raise Not_Found;
   end Search_Array;

end Show_Search_Array;
```

Flow analysis doesn't emit any messages in this case, meaning it can verify that `Result` can't be read in SPARK code while uninitialized. But why is that, since `Result` is still not initialized when `E` is not in `A`? This is because the exception, `Not_Found`, can never be caught within SPARK code (SPAK doesn't allow exception handlers). However, the GNATprove tool also tries to ensure the absence of runtime errors in SPARK code, so tries to prove that `Not_Found` is never raised. When it can't do that here, it produces a different message.

2.5.3 Example #3

In this example, we're using a discriminated record for the result of `Search_Array` instead of conditionally raising an exception. By using such a structure, the place to store the index at which `E` was found exists only when `E` was indeed found. So if it wasn't found, there's nothing to be initialized.

```
package Show_Search_Array is

   type Array_Of_Positives is array (Natural range <>) of Positive;

   type Search_Result (Found : Boolean := False) is record
      case Found is
         when True =>
            Content : Integer;
         when False => null;
      end case;
   end record;

   procedure Search_Array
     (A      :     Array_Of_Positives;
      E      :     Positive;
      Result : out Search_Result)
   with Pre => not Result'Constrained;

end Show_Search_Array;
```

```
package body Show_Search_Array is

   procedure Search_Array
     (A      :     Array_Of_Positives;
      E      :     Positive;
      Result : out Search_Result) is
   begin
      for I in A'Range loop
         if A (I) = E then
```

(continues on next page)

(continued from previous page)

```
            Result := (Found   => True,
                       Content => I);
            return;
         end if;
      end loop;
      Result := (Found => False);
   end Search_Array;

end Show_Search_Array;
```

This example is correct and flow analysis doesn't issue any message: it can verify both that no uninitialized variables are read in `Search_Array`'s body, and that all its outputs are set on return. We've used the attribute `Constrained` in the precondition of `Search_Array` to indicate that the value of the `Result` in argument can be set to any variant of the record type `Search_Result`, specifically to either the variant where `E` was found and where it wasn't.

2.5.4 Example #4

The function `Size_Of_Biggest_Increasing_Sequence` is supposed to find all sequences within its parameter `A` that contain elements with increasing values and returns the length of the longest one. To do this, it calls a nested procedure `Test_Index` iteratively on all the elements of `A`. `Test_Index` checks if the sequence is still increasing. If so, it updates the largest value seen so far in this sequence. If not, it means it's found the end of a sequence, so it computes the size of that sequence and stores it in `Size_Of_Seq`.

```
package Show_Biggest_Increasing_Sequence is

   type Array_Of_Positives is array (Integer range <>) of Positive;

   function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
      return Natural;

end Show_Biggest_Increasing_Sequence;
```

```
package body Show_Biggest_Increasing_Sequence is

   function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
      return Natural
   is
      Max         : Natural;
      End_Of_Seq  : Boolean;
      Size_Of_Seq : Natural;
      Beginning   : Integer;

      procedure Test_Index (Current_Index : Integer) is
      begin
         if A (Current_Index) >= Max then
            Max := A (Current_Index);
            End_Of_Seq := False;
         else
            Max         := 0;
            End_Of_Seq  := True;
            Size_Of_Seq := Current_Index - Beginning;
            Beginning   := Current_Index;
         end if;
      end Test_Index;

      Biggest_Seq : Natural := 0;
```

(continues on next page)

(continued from previous page)

```
   begin
      for I in A'Range loop
         Test_Index (I);
         if End_Of_Seq then
            Biggest_Seq := Natural'Max (Size_Of_Seq, Biggest_Seq);
         end if;
      end loop;
      return Biggest_Seq;
   end Size_Of_Biggest_Increasing_Sequence;

end Show_Biggest_Increasing_Sequence;
```

However, this example is not correct. Flow analysis emits messages for `Test_Index` stating that `Max`, `Beginning`, and `Size_Of_Seq` should be initialized before being read. Indeed, when you look carefully, you see that both `Max` and `Beginning` are missing initializations because they are read in `Test_Index` before being written. As for `Size_Of_Seq`, we only read its value when `End_Of_Seq` is true, so it actually can't be read before being written, but flow analysis isn't able to verify its initialization by using just flow information.

The call to `Test_Index` is automatically inlined by GNATprove, which leads to another messages above. If GNATprove couldn't inline the call to `Test_Index`, for example if it was defined in another unit, the same messages would be issued on the call to `Test_Index`.

2.5.5 Example #5

In the following example, we model permutations as arrays where the element at index `I` is the position of the `I`'th element in the permutation. The procedure `Init` initializes a permutation to the identity, where the `I`'th elements is at the `I`'th position. `Cyclic_Permutation` calls `Init` and then swaps elements to construct a cyclic permutation.

```
package Show_Permutation is

   type Permutation is array (Positive range <>) of Positive;

   procedure Swap (A    : in out Permutation;
                   I, J : Positive);

   procedure Init (A : out Permutation);

   function Cyclic_Permutation (N : Natural) return Permutation;

end Show_Permutation;
```

```
package body Show_Permutation is

   procedure Swap (A    : in out Permutation;
                   I, J : Positive)
   is
      Tmp : Positive := A (I);
   begin
      A (I) := A (J);
      A (J) := Tmp;
   end Swap;

   procedure Init (A : out Permutation) is
   begin
      A (A'First) := A'First;
```

(continues on next page)

(continued from previous page)

```
      for I in A'First + 1 .. A'Last loop
         A (I) := I;
      end loop;
   end Init;

   function Cyclic_Permutation (N : Natural) return Permutation is
      A : Permutation (1 .. N);
   begin
      Init (A);
      for I in A'First .. A'Last - 1 loop
         Swap (A, I, I + 1);
      end loop;
      return A;
   end Cyclic_Permutation;

end Show_Permutation;
```

This program is correct. However, flow analysis will nevertheless still emit messages because it can't verify that every element of `A` is initialized by the loop in `Init`. This message is a false alarm. You can either ignore it or justify it safely.

2.5.6 Example #6

This program is the same as the previous one except that we've changed the mode of `A` in the specification of `Init` to `in out` to avoid the message from flow analysis on array assignment.

```
package Show_Permutation is

   type Permutation is array (Positive range <>) of Positive;

   procedure Swap (A    : in out Permutation;
                   I, J : Positive);

   procedure Init (A : in out Permutation);

   function Cyclic_Permutation (N : Natural) return Permutation;

end Show_Permutation;
```

```
package body Show_Permutation is

   procedure Swap (A    : in out Permutation;
                   I, J : Positive)
   is
      Tmp : Positive := A (I);
   begin
      A (I) := A (J);
      A (J) := Tmp;
   end Swap;

   procedure Init (A : in out Permutation) is
   begin
      A (A'First) := A'First;
      for I in A'First + 1 .. A'Last loop
         A (I) := I;
      end loop;
   end Init;

   function Cyclic_Permutation (N : Natural) return Permutation is
```

(continues on next page)

(continued from previous page)

```
      A : Permutation (1 .. N);
   begin
      Init (A);
      for I in A'First .. A'Last - 1 loop
         Swap (A, I, I + 1);
      end loop;
      return A;
   end Cyclic_Permutation;

end Show_Permutation;
```

This program is not correct. Changing the mode of a parameter that should really be `out` to `in out` to silence a false alarm is not a good idea. Not only does this obfuscate the specification of `Init`, but flow analysis emits a message on the procedure where `A` is not initialized, as shown by the message in `Cyclic_Permutation`.

2.5.7 Example #7

`Incr_Step_Function` takes an array `A` as an argument and iterates through `A` to increment every element by the value of `Increment`, saturating at a specified threshold value. We specified a `Global` contract for `Incr_Until_Threshold`.

```
package Show_Increments is

   type Array_Of_Positives is array (Natural range <>) of Positive;

   Increment : constant Natural := 10;

   procedure Incr_Step_Function (A : in out Array_Of_Positives);

end Show_Increments;
```

```
package body Show_Increments is

   procedure Incr_Step_Function (A : in out Array_Of_Positives) is

      Threshold : Positive := Positive'Last;

      procedure Incr_Until_Threshold (I : Integer) with
        Global => (Input  => Threshold,
                   In_Out => A);

      procedure Incr_Until_Threshold (I : Integer) is
      begin
         if Threshold - Increment <= A (I) then
            A (I) := Threshold;
         else
            A (I) := A (I) + Increment;
         end if;
      end Incr_Until_Threshold;

   begin
      for I in A'Range loop
         if I > A'First then
            Threshold := A (I - 1);
         end if;
         Incr_Until_Threshold (I);
      end loop;
```

(continues on next page)

(continued from previous page)

```
   end Incr_Step_Function;

end Show_Increments;
```

Everything is fine here. Specifically, the `Global` contract is correct. It mentions both `Threshold`, which is read but not written in the procedure, and `A`, which is both read and written. The fact that `A` is a parameter of an enclosing unit doesn't prevent us from using it inside the `Global` contract; it really is global to `Incr_Until_Threshold`. We didn't mention `Increment` since it's a static constant.

2.5.8 Example #8

We now go back to the procedure `Test_Index` from *Example #4* (page 28) and correct the missing initializations. We want to know if the `Global` contract of `Test_Index` is correct.

```
package Show_Biggest_Increasing_Sequence is

   type Array_Of_Positives is array (Integer range <>) of Positive;

   function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
      return Natural;

end Show_Biggest_Increasing_Sequence;
```

```
package body Show_Biggest_Increasing_Sequence is

   function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
      return Natural
   is
      Max         : Natural := 0;
      End_Of_Seq  : Boolean;
      Size_Of_Seq : Natural := 0;
      Beginning   : Integer := A'First - 1;

      procedure Test_Index (Current_Index : Integer) with
        Global => (In_Out => (Beginning, Max, Size_Of_Seq),
                   Output => End_Of_Seq,
                   Input  => Current_Index)
      is
      begin
         if A (Current_Index) >= Max then
            Max := A (Current_Index);
            End_Of_Seq := False;
         else
            Max         := 0;
            End_Of_Seq  := True;
            Size_Of_Seq := Current_Index - Beginning;
            Beginning   := Current_Index;
         end if;
      end Test_Index;

      Biggest_Seq : Natural := 0;

   begin
      for I in A'Range loop
         Test_Index (I);
         if End_Of_Seq then
            Biggest_Seq := Natural'Max (Size_Of_Seq, Biggest_Seq);
```

(continues on next page)

(continued from previous page)

```
         end if;
      end loop;
      return Biggest_Seq;
   end Size_Of_Biggest_Increasing_Sequence;

end Show_Biggest_Increasing_Sequence;
```

The contract in this example is not correct: `Current_Index` is a parameter of `Test_Index`, so we shouldn't reference it as a global variable. Also, we should have listed variable `A` from the outer scope as an `Input` in the `Global` contract.

2.5.9 Example #9

Next, we change the `Global` contract of `Test_Index` into a `Depends` contract. In general, we don't need both contracts because the set of global variables accessed can be deduced from the `Depends` contract.

```
package Show_Biggest_Increasing_Sequence is

   type Array_Of_Positives is array (Integer range <>) of Positive;

   function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
      return Natural;

end Show_Biggest_Increasing_Sequence;
```

```
package body Show_Biggest_Increasing_Sequence is

   function Size_Of_Biggest_Increasing_Sequence (A : Array_Of_Positives)
      return Natural
   is
      Max         : Natural := 0;
      End_Of_Seq  : Boolean;
      Size_Of_Seq : Natural := 0;
      Beginning   : Integer := A'First - 1;

      procedure Test_Index (Current_Index : Integer) with
        Depends => ((Max, End_Of_Seq)        => (A, Current_Index, Max),
                    (Size_Of_Seq, Beginning) =>
                       + (A, Current_Index, Max, Beginning))
      is
      begin
         if A (Current_Index) >= Max then
            Max := A (Current_Index);
            End_Of_Seq := False;
         else
            Max         := 0;
            End_Of_Seq  := True;
            Size_Of_Seq := Current_Index - Beginning;
            Beginning   := Current_Index;
         end if;
      end Test_Index;

      Biggest_Seq : Natural := 0;

   begin
      for I in A'Range loop
         Test_Index (I);
```

(continues on next page)

(continued from previous page)

```
         if End_Of_Seq then
            Biggest_Seq := Natural'Max (Size_Of_Seq, Biggest_Seq);
         end if;
      end loop;
      return Biggest_Seq;
   end Size_Of_Biggest_Increasing_Sequence;

end Show_Biggest_Increasing_Sequence;
```

This example is correct. Some of the dependencies, such as `Size_Of_Seq` depending on `Beginning`, come directly from the assignments in the subprogram. Since the control flow influences the final value of all of the outputs, the variables that are being read, `A`, `Current_Index`, and `Max`, are present in every dependency relation. Finally, the dependencies of `Size_Of_Eq` and `Beginning` on themselves are because they may not be modified by the subprogram execution.

2.5.10 Example #10

The subprogram `Identity` swaps the value of its parameter two times. Its `Depends` contract says that the final value of `X` only depends on its initial value and likewise for `Y`.

```
package Show_Swap is

   procedure Swap (X, Y : in out Positive);

   procedure Identity (X, Y : in out Positive) with
     Depends => (X => X,
                 Y => Y);

end Show_Swap;
```

```
package body Show_Swap is

   procedure Swap (X, Y : in out Positive) is
      Tmp : constant Positive := X;
   begin
      X := Y;
      Y := Tmp;
   end Swap;

   procedure Identity (X, Y : in out Positive) is
   begin
      Swap (X, Y);
      Swap (Y, X);
   end Identity;

end Show_Swap;
```

This code is correct, but flow analysis can't verify the `Depends` contract of `Identity` because we didn't supply a `Depends` contract for `Swap`. Therefore, flow analysis assumes that all outputs of `Swap`, `X` and `Y`, depend on all its inputs, both `X` and `Y`'s initial values. To prevent this, we should manually specify a `Depends` contract for `Swap`.

CHAPTER

THREE

PROOF OF PROGRAM INTEGRITY

This section presents the proof capability of GNATprove, a major tool for the SPARK language. We focus here on the simpler proofs that you'll need to write to verify your program's integrity. The primary objective of performing proof of your program's integrity is to ensure the absence of runtime errors during its execution.

The analysis steps discussed here are only sound if you've previously performed *Flow Analysis* (page 17). You shouldn't proceed further if there you still have unjustified flow analysis messages for your program.

3.1 Runtime Errors

There's always the potential for errors that aren't detected during compilation to occur during a program's execution. These errors, called runtime errors, are those targeted by GNATprove.

There are various kinds of runtime errors, the most common being references that are out of the range of an array (buffer overflow[8] in Ada), subtype range violations, overflows in computations, and divisions by zero. The code below illustrates many examples of possible runtime errors, all within a single statement. Look at the assignment statement setting the `I + J`'th cell of an array `A` to the value `P / Q`.

```
package Show_Runtime_Errors is

   type Nat_Array is array (Integer range <>) of Natural;

   procedure Update (A : in out Nat_Array; I, J, P, Q : Integer);

end Show_Runtime_Errors;
```

```
package body Show_Runtime_Errors is

   procedure Update (A : in out Nat_Array; I, J, P, Q : Integer) is
   begin
      A (I + J) := P / Q;
   end Update;

end Show_Runtime_Errors;
```

There are quite a number of errors that may occur when executing this code. If we don't know anything about the values of `I`, `J`, `P`, and `Q`, we can't rule out any of those errors.

First, the computation of `I + J` can overflow, for example if `I` is `Integer'Last` and `J` is positive.

```
A (Integer'Last + 1) := P / Q;
```

[8] https://en.wikipedia.org/wiki/Buffer_overflow

Next, the sum, which is used as an array index, may not be in the range of the index of the array.

```
A (A'Last + 1) := P / Q;
```

On the other side of the assignment, the division may also overflow, though only in the very special case where `P` is `Integer'First` and `Q` is -1 because of the asymmetric range of signed integer types.

```
A (I + J) := Integer'First / -1;
```

The division is also not allowed if `Q` is 0.

```
A (I + J) := P / 0;
```

Finally, since the array contains natural numbers, it's also an error to store a negative value in it.

```
A (I + J) := 1 / -1;
```

The compiler generates checks in the executable code corresponding to each of those runtime errors. Each check raises an exception if it fails. For the above asssignment statement, we can see examples of exceptions raised due to failed checks for each of the different cases above.

```
A (Integer'Last + 1) := P / Q;
-- raised CONSTRAINT_ERROR : overflow check failed

A (A'Last + 1) := P / Q;
-- raised CONSTRAINT_ERROR : index check failed

A (I + J) := Integer'First / (-1);
-- raised CONSTRAINT_ERROR : overflow check failed

A (I + J) := 1 / (-1);
-- raised CONSTRAINT_ERROR : range check failed

A (I + J) := P / 0;
-- raised CONSTRAINT_ERROR : divide by zero
```

These runtime checks are costly, both in terms of program size and execution time. It may be appropriate to remove them if we can statically ensure they aren't needed at runtime, in other words if we can prove that the condition tested for can never occur.

This is where the analysis done by GNATprove comes in. It can be used to demonstrate statically that none of these errors can ever occur at runtime. Specifically, GNATprove logically interprets the meaning of every instruction in the program. Using this interpretation, GNATprove generates a logical formula called a *verification condition* for each check that would otherwise be required by the Ada (and hence SPARK) language.

```
A (Integer'Last + 1) := P / Q;
-- medium: overflow check might fail

A (A'Last + 1) := P / Q;
-- medium: array index check might fail

A (I + J) := Integer'First / (-1);
-- medium: overflow check might fail

A (I + J) := 1 / (-1);
-- medium: range check might fail

A (I + J) := P / 0;
-- medium: divide by zero might fail
```

GNATprove then passes these verification conditions to an automatic prover, stated as conditions that must be true to avoid the error. If every such condition can be validated by a prover (meaning that it can be mathematically shown to always be true), we've been able to prove that no error can ever be raised at runtime when executing that program.

3.2 Modularity

To scale to large programs, GNATprove performs proofs on a per-subprogram basis by relying on preconditions and postconditions to properly summarize the input and output state of each subprogram. More precisely, when verifying the body of a subprogram, GNATprove assumes it knows nothing about the possible initial values of its parameters and of the global variables it accesses except what you state in the subprogram's precondition. If you don't specify a precondition, it can't make any assumptions.

For example, the following code shows that the body of `Increment` can be successfully verified: its precondition constrains the value of its parameter `X` to be less than `Integer'Last` so we know the overflow check is always false.

In the same way, when a subprogram is called, GNATprove assumes its `out` and `in out` parameters and the global variables it writes can be modified in any way compatible with their postconditions. For example, since `Increment` has no postcondition, GNATprove doesn't know that the value of `X` after the call is always less than `Integer'Last`. Therefore, it can't prove that the addition following the call to `Increment` can't overflow.

```
procedure Show_Modularity is

   procedure Increment (X : in out Integer) with
     Pre => X < Integer'Last is
   begin
      X := X + 1;
      --  info: overflow check proved
   end Increment;

   X : Integer;
begin
   X := Integer'Last - 2;
   Increment (X);
   --  After the call, GNATprove no longer knows the value of X

   X := X + 1;
   --  medium: overflow check might fail
end Show_Modularity;
```

3.2.1 Exceptions

There are two cases where GNATprove doesn't require modularity and hence doesn't make the above assumptions. First, local subprograms without contracts can be inlined if they're simple enough and are neither recursive nor have multiple return points. If we remove the contract from `Increment`, it fits the criteria for inlining.

```
procedure Show_Modularity is

   procedure Increment (X : in out Integer) is
   begin
      X := X + 1;
      --  info: overflow check proved, in call inlined at...
   end Increment;
```

(continues on next page)

(continued from previous page)

```
   X : Integer;
begin
   X := Integer'Last - 2;
   Increment (X);
   X := X + 1;
   --  info: overflow check proved
end Show_Modularity;
```

GNATprove now sees the call to `Increment` exactly as if the increment on X was done outside that call, so it can successfully verify that neither addition can overflow.

The other case involves functions. If we define a function as an expression function, with or without contracts, GNATprove uses the expression itself as the postcondition on the result of the function.

In our example, replacing `Increment` with an expression function allows GNATprove to successfully verify the overflow check in the addition.

```
procedure Show_Modularity is

   function Increment (X : Integer) return Integer is
     (X + 1)
     --  info: overflow check proved
     with Pre => X < Integer'Last;

   X : Integer;
begin
   X := Integer'Last - 2;
   X := Increment (X);
   X := X + 1;
   --  info: overflow check proved
end Show_Modularity;
```

3.3 Contracts

Ada contracts are perfectly suited for formal verification, but are primarily designed to be checked at runtime. When you specify the `-gnata` switch, the compiler generates code that verifies the contracts at runtime. If an Ada contract isn't satisfied for a given subprogram call, the program raises the `Assert_Failure` exception. This switch is particularly useful during development and testing, but you may also retain run-time execution of assertions, and specifically preconditions, during the program's deployment to avoid an inconsistent state.

Consider the incorrect call to `Increment` below, which violates its precondition. One way to detect this error is by compiling the function with assertions enabled and testing it with inputs that trigger the violation. Another way, one that doesn't require guessing the needed inputs, is to run GNATprove.

```
procedure Show_Precondition_Violation is

    procedure Increment (X : in out Integer) with
      Pre => X < Integer'Last  is
    begin
       X := X + 1;
    end Increment;

    X : Integer;

 begin
```

(continues on next page)

(continued from previous page)

```
   X := Integer'Last;
   Increment (X);
end Show_Precondition_Violation;
```

Similarly, consider the incorrect implementation of function `Absolute` below, which violates its postcondition. Likewise, one way to detect this error is by compiling the function with assertions enabled and testing with inputs that trigger the violation. Another way, one which again doesn't require finding the inputs needed to demonstrate the error, is to run GNATprove.

```
procedure Show_Postcondition_Violation is

   procedure Absolute (X : in out Integer) with
     Post => X >= 0 is
   begin
      if X > 0 then
         X := -X;
      end if;
   end Absolute;

   X : Integer;

begin
   X := 1;
   Absolute (X);
end Show_Postcondition_Violation;
```

The benefits of dynamically checking contracts extends beyond making testing easier. Early failure detection also allows an easier recovery and facilitates debugging, so you may want to enable these checks at runtime to terminate execution before some damaging or hard-to-debug action occurs.

GNATprove statically analyses preconditions and postcondition. It verifies preconditions every time a subprogram is called, which is the runtime semantics of contracts. Postconditions, on the other hand, are verified once as part of the verification of the subprogram's body. For example, GNATprove must wait until `Increment` is improperly called to detect the precondition violation, since a precondition is really a contract for the caller. On the other hand, it doesn't need `Absolute` to be called to detect that its postcondition doesn't hold for all its possible inputs.

3.3.1 Executable Semantics

Expressions in Ada contracts have the same semantics as Boolean expressions elsewhere, so runtime errors can occur during their computation. To simplify both debugging of assertions and combining testing and static verification, the same semantics are used by GNATprove.

While proving programs, GNATprove verifies that no error can ever be raised during the execution of the contracts. However, you may sometimes find those semantics too heavy, in particular with respect to overflow checks, because they can make it harder to specify an appropriate precondition. We see this in the function `Add` below.

```
procedure Show_Executable_Semantics
   with SPARK_Mode => On
is
   function Add (X, Y : Integer) return Integer is (X + Y)
     with Pre => X + Y in Integer;

   X : Integer;
begin
   X := Add (Integer'Last, 1);
end Show_Executable_Semantics;
```

GNATprove issues a message on this code warning about a possible overflow when computing the sum of X and Y in the precondition. Indeed, since expressions in assertions have normal Ada semantics, this addition can overflow, as you can easily see by compiling and running the code that calls Add with arguments `Integer'Last` and 1.

On the other hand, you sometimes may prefer GNATprove to use the mathematical semantics of addition in contracts while the generated code still properly verifies that no error is ever raised at runtime in the body of the program. You can get this behavior by using the compiler switch `-gnato??` (for example `-gnato13`), which allows you to independently set the overflow mode in code (the first digit) and assertions (the second digit). For both, you can either reduce the number of overflow checks (the value 2), completely eliminate them (the value 3), or preserve the default Ada semantics (the value 1).

3.3.2 Additional Assertions and Contracts

As we've seen, a key feature of SPARK is that it allows us to state properties to check using assertions and contracts. SPARK supports preconditions and postconditions as well as assertions introduced by the `Assert` pragma.

The SPARK language also includes new contract types used to assist formal verification. The new pragma `Assume` is treated as an assertion during execution but introduces an assumption when proving programs. Its value is a Boolean expression which GNATprove assumes to be true without any attempt to verify that it's true. You'll find this feature useful, but you must use it with great care. Here's an example of using it.

```
procedure Incr (X : in out Integer) is
begin
   pragma Assume (X < Integer'Last);
   X := X + 1;
end Incr;
```

The `Contract_Cases` aspect is another construct introduced for GNATprove, but which also acts as an assertion during execution. It allows you to specify the behavior of a subprogram using a disjunction of cases. Each element of a `Contract_Cases` aspect is a *guard*, which is evaluated before the call and may only reference the subprogram's inputs, and a *consequence*. At each call of the subprogram, one and only one guard is permitted to evaluate to `True`. The consequence of that case is a contract that's required to be satisfied when the subprogram returns.

```
procedure Absolute (X : in out Integer) with
  Pre            =>  X > Integer'First,
  Contract_Cases => (X <  0 => X = -X'Old,
                     X >= 0 => X =  X'Old)
is
begin
   if X < 0 then
      X := -X;
   end if;
end Absolute;
```

Similarly to how it analyzes a subprogram's precondition, GNATprove verifies the `Contract_Cases` only once. It verifies the validity of each consequence (given the truth of its guard) and the disjointness and completeness of the guard conditions (meaning that exactly one guard must be true for each possible set of input values).

3.4 Debugging Failed Proof Attempts

GNATprove may report an error while verifying a program for any of the following reasons:

- there might be an error in the program; or
- the property may not be provable as written because more information is required; or
- the prover used by GNATprove may be unable to prove a perfectly valid property.

We spend the remainder of this section discussing the sometimes tricky task of debugging failed proof attempts.

3.4.1 Debugging Errors in Code or Specification

First, let's discuss the case where there's indeed an error in the program. There are two possibilities: the code may be incorrect or, equally likely, the specification may be incorrect. As an example, there's an error in our procedure `Incr_Until` below which makes its `Contract_Cases` unprovable.

```
package Show_Failed_Proof_Attempt is

   Incremented : Boolean := False;

   procedure Incr_Until (X : in out Natural) with
     Contract_Cases =>
       (Incremented => X > X'Old,
        others      => X = X'Old);

end Show_Failed_Proof_Attempt;
```

```
package body Show_Failed_Proof_Attempt is

   procedure Incr_Until (X : in out Natural) is
   begin
      if X < 1000 then
         X := X + 1;
         Incremented := True;
      else
         Incremented := False;
      end if;
   end Incr_Until;

end Show_Failed_Proof_Attempt;
```

Since this is an assertion that can be executed, it may help you find the problem if you run the program with assertions enabled on representative sets of inputs. This allows you to find bugs in both the code and its contracts. In this case, testing `Incr_Until` with an input greater than 1000 raises an exception at runtime.

```
package Show_Failed_Proof_Attempt is

   Incremented : Boolean := False;

   procedure Incr_Until (X : in out Natural) with
     Contract_Cases =>
       (Incremented => X > X'Old,
        others      => X = X'Old);

end Show_Failed_Proof_Attempt;
```

```
package body Show_Failed_Proof_Attempt is

   procedure Incr_Until (X : in out Natural) is
```

(continues on next page)

(continued from previous page)

```
   begin
      if X < 1000 then
         X := X + 1;
         Incremented := True;
      else
         Incremented := False;
      end if;
   end Incr_Until;

end Show_Failed_Proof_Attempt;
```

```
with Show_Failed_Proof_Attempt; use Show_Failed_Proof_Attempt;

procedure Main is
   Dummy : Integer;
begin
   Dummy := 0;
   Incr_Until (Dummy);

   Dummy := 1000;
   Incr_Until (Dummy);
end Main;
```

The error message shows that the first contract case is failing, which means that `Incremented` is `True`. However, if we print the value of `Incremented` before returning, we see that it's `False`, as expected for the input we provided. The error here is that guards of contract cases are evaluated before the call, so our specification is wrong! To correct this, we should either write `X < 1000` as the guard of the first case or use a standard postcondition with an if-expression.

3.4.2 Debugging Cases where more Information is Required

Even if both the code and the assertions are correct, GNATprove may still report that it can't prove a verification condition for a property. This can happen for two reasons:

- The property may be unprovable because the code is missing some assertion. One category of these cases is due to the modularity of the analysis which, as we disussed above, means that GNATprove only knows about the properties of your subprograms that you have explicitly written.
- There may be some information missing in the logical model of the program used by GNATprove.

Let's look at the case where the code and the specification are correct but there's some information missing. As an example, GNATprove finds the postcondition of `Increase` to be unprovable.

```
package Show_Failed_Proof_Attempt is

   C : Natural := 100;

   procedure Increase (X : in out Natural) with
      Post => (if X'Old < C then X > X'Old else X = C);

end Show_Failed_Proof_Attempt;
```

```
package body Show_Failed_Proof_Attempt is

   procedure Increase (X : in out Natural) is
   begin
      if X < 90 then
```

(continues on next page)

(continued from previous page)

```
         X := X + 10;
      elsif X >= C then
         X := C;
      else
         X := X + 1;
      end if;
   end Increase;

end Show_Failed_Proof_Attempt;
```

This postcondition is a conditional. It says that if the parameter (X) is less than a certain value (C), its value will be increased by the procedure while if it's greater, its value will be set to C (saturated). When C has the value 100, the code of `Increases` adds 10 to the value of X if it was initially less than 90, increments X by 1 if it was between 90 and 99, and sets X to 100 if it was greater or equal to 100. This behavior does satisfy the postcondition, so why is the postcondition not provable?

The values in the counterexample returned by GNATprove in its message gives us a clue: `C = 0 and X = 10 and X'Old = 0`. Indeed, if C is not equal to 100, our reasoning above is incorrect: the values of 0 for C and X on entry indeed result in X being 10 on exit, which violates the postcondition!

We probably didn't expect the value of C to change, or at least not to go below 90. But, in that case, we should have stated so by either declaring C to be constant or by adding a precondition to the `Increase` subprogram. If we do either of those, GNATprove is able to prove the postcondition.

3.4.3 Debugging Prover Limitations

Finally, there are cases where GNATprove provides a perfectly valid verification condition for a property, but it's neverthless not proved by the automatic prover that runs in the later stages of the tool's execution. This is quite common. Indeed, GNATprove produces its verification conditions in first-order logic, which is not decidable, especially in combination with the rules of arithmetic. Sometimes, the automatic prover just needs more time. Other times, the prover will abandon the search almost immediately or loop forever without reaching a conclusive answer (either a proof or a counterexample).

For example, the postcondition of our GCD function below — which calculates the value of the GCD of two positive numbers using Euclide's algorithm — can't be verified with GNATprove's default settings.

```
package Show_Failed_Proof_Attempt is

   function GCD (A, B : Positive) return Positive with
     Post =>
       A mod GCD'Result = 0
       and B mod GCD'Result = 0;

end Show_Failed_Proof_Attempt;
```

```
package body Show_Failed_Proof_Attempt is

   function GCD (A, B : Positive) return Positive is
   begin
      if A > B then
         return GCD (A - B, B);
      elsif B > A then
         return GCD (A, B - A);
      else
         return A;
```

(continues on next page)

(continued from previous page)

```
      end if;
   end GCD;

end Show_Failed_Proof_Attempt;
```

The first thing we try is increasing the amount of time the prover is allowed to spend on each verification condition using the `--timeout` option of GNATprove (e.g., by using the dialog box in GPS). In this example, increasing it to one minute, which is relatively high, doesn't help. We can also specify an alternative automatic prover — if we have one — using the option `--prover` of GNATprove (or the dialog box). For our postcondition, we tried Alt-Ergo, CVC4, and Z3 without any luck.

```
package Show_Failed_Proof_Attempt is

   function GCD (A, B : Positive) return Positive with
     Post =>
       A mod GCD'Result = 0
       and B mod GCD'Result = 0;

end Show_Failed_Proof_Attempt;
```

```
package body Show_Failed_Proof_Attempt is

   function GCD (A, B : Positive) return Positive
   is
      Result : Positive;
   begin
      if A > B then
         Result := GCD (A - B, B);
         pragma Assert ((A - B) mod Result = 0);
         --  info: assertion proved
         pragma Assert (B mod Result = 0);
         --  info: assertion proved
         pragma Assert (A mod Result = 0);
         --  medium: assertion might fail
      elsif B > A then
         Result := GCD (A, B - A);
         pragma Assert ((B - A) mod Result = 0);
         --  info: assertion proved
      else
         Result := A;
      end if;
      return Result;
   end GCD;

end Show_Failed_Proof_Attempt;
```

To better understand the reason for the failure, we added intermediate assertions to simplify the proof and pin down the part that's causing the problem. Adding such assertions is often a good idea when trying to understand why a property is not proved. Here, provers can't verify that if both A - B and B can be divided by `Result` so can A. This may seem surprising, but non-linear arithmetic, involving, for example, multiplication, modulo, or exponentiation, is a difficult topic for provers and is not handled very well in practice by any of the general-purpose ones like Alt-Ergo, CVC4, or Z3.

3.5 Code Examples / Pitfalls

We end with some code examples and pitfalls.

3.5.1 Example #1

The package `Lists` defines a linked-list data structure. We call `Link(I,J)` to make a link from index `I` to index `J` and call `Goes_To(I,J)` to determine if we've created a link from index `I` to index `J`. The postcondition of `Link` uses `Goes_To` to state that there must be a link between its arguments once `Link` completes.

```
package Lists with SPARK_Mode is

   type Index is new Integer;

   function Goes_To (I, J : Index) return Boolean;

   procedure Link (I, J : Index) with Post => Goes_To (I, J);

private

   type Cell (Is_Set : Boolean := True) is record
      case Is_Set is
         when True =>
            Next : Index;
         when False =>
            null;
      end case;
   end record;

   type Cell_Array is array (Index) of Cell;

   Memory : Cell_Array;

end Lists;
```

```
package body Lists with SPARK_Mode is

   function Goes_To (I, J : Index) return Boolean is
   begin
      if Memory (I).Is_Set then
         return Memory (I).Next = J;
      end if;
      return False;
   end Goes_To;

   procedure Link (I, J : Index) is
   begin
      Memory (I) := (Is_Set => True, Next => J);
   end Link;

end Lists;
```

This example is correct, but can't be verified by GNATprove. This is because `Goes_To` itself has no postcondition, so nothing is known about its result.

3.5.2 Example #2

We now redefine `Goes_To` as an expression function.

```
package Lists with SPARK_Mode is

   type Index is new Integer;
```

(continues on next page)

(continued from previous page)

```
   function Goes_To (I, J : Index) return Boolean;

   procedure Link (I, J : Index) with Post => Goes_To (I, J);

private

   type Cell (Is_Set : Boolean := True) is record
      case Is_Set is
      when True =>
         Next : Index;
      when False =>
         null;
      end case;
   end record;

   type Cell_Array is array (Index) of Cell;

   Memory : Cell_Array;

   function Goes_To (I, J : Index) return Boolean is
     (Memory (I).Is_Set and then Memory (I).Next = J);

end Lists;
```

```
package body Lists with SPARK_Mode is

   procedure Link (I, J : Index) is
   begin
      Memory (I) := (Is_Set => True, Next => J);
   end Link;

end Lists;
```

GNATprove can fully prove this version: `Goes_To` is an expression function, so its body is available for proof (specifically, for creating the postcondition needed for the proof).

3.5.3 Example #3

The package `Stacks` defines an abstract stack type with a `Push` procedure that adds an element at the top of the stack and a function `Peek` that returns the content of the element at the top of the stack (without removing it).

```
package Stacks with SPARK_Mode is

   type Stack is private;

   function  Peek (S : Stack) return Natural;
   procedure Push (S : in out Stack; E : Natural) with
     Post => Peek (S) = E;

private

   Max : constant := 10;

   type Stack_Array is array (1 .. Max) of Natural;

   type Stack is record
      Top     : Positive;
      Content : Stack_Array;
```

(continues on next page)

(continued from previous page)

```
   end record;

   function Peek (S : Stack) return Natural is
     (if S.Top in S.Content'Range then S.Content (S.Top) else 0);

end Stacks;
```

```
package body Stacks with SPARK_Mode is

   procedure Push (S : in out Stack; E : Natural) is
   begin
      if S.Top >= Max then
         return;
      end if;

      S.Top := S.Top + 1;
      S.Content (S.Top) := E;
   end Push;

end Stacks;
```

This example isn't correct. The postcondition of Push is only satisfied if the stack isn't full when we call Push.

3.5.4 Example #4

We now change the behavior of Push so it raises an exception when the stack is full instead of returning.

```
package Stacks with SPARK_Mode is

   type Stack is private;

   Is_Full_E : exception;

   function  Peek (S : Stack) return Natural;
   procedure Push (S : in out Stack; E : Natural) with
     Post => Peek (S) = E;

private

   Max : constant := 10;

   type Stack_Array is array (1 .. Max) of Natural;

   type Stack is record
      Top     : Positive;
      Content : Stack_Array;
   end record;

   function Peek (S : Stack) return Natural is
     (if S.Top in S.Content'Range then S.Content (S.Top) else 0);

end Stacks;
```

```
package body Stacks with SPARK_Mode is

   procedure Push (S : in out Stack; E : Natural) is
```

(continues on next page)

(continued from previous page)

```
   begin
      if S.Top >= Max then
         raise Is_Full_E;
      end if;

      S.Top := S.Top + 1;
      S.Content (S.Top) := E;
   end Push;

end Stacks;
```

The postcondition of `Push` is now proved because GNATprove only considers execution paths leading to normal termination. But it issues a message warning that exception `Is_Full_E` may be raised at runtime.

3.5.5 Example #5

Let's add a precondition to `Push` stating that the stack shouldn't be full.

```
package Stacks with SPARK_Mode is

   type Stack is private;

   Is_Full_E : exception;

   function  Peek (S : Stack) return Natural;
   function  Is_Full (S : Stack) return Boolean;
   procedure Push (S : in out Stack; E : Natural) with
     Pre  => not Is_Full (S),
     Post => Peek (S) = E;

private

   Max : constant := 10;

   type Stack_Array is array (1 .. Max) of Natural;

   type Stack is record
      Top     : Positive;
      Content : Stack_Array;
   end record;

   function Peek (S : Stack) return Natural is
     (if S.Top in S.Content'Range then S.Content (S.Top) else 0);
   function Is_Full (S : Stack) return Boolean is (S.Top >= Max);

end Stacks;
```

```
package body Stacks with SPARK_Mode is

   procedure Push (S : in out Stack; E : Natural) is
   begin
      if S.Top >= Max then
         raise Is_Full_E;
      end if;
      S.Top := S.Top + 1;
      S.Content (S.Top) := E;
   end Push;
```

(continues on next page)

(continued from previous page)

```
end Stacks;
```

This example is correct. With the addition of the precondition, GNATprove can now verify that `Is_Full_E` can never be raised at runtime.

3.5.6 Example #6

The package `Memories` defines a type `Chunk` that models chunks of memory. Each element of the array, represented by its index, corresponds to one data element. The procedure `Read_Record` reads two pieces of data starting at index `From` out of the chunk represented by the value of `Memory`.

```
package Memories is

   type Chunk is array (Integer range <>) of Integer
     with Predicate => Chunk'Length >= 10;

   function Is_Too_Coarse (V : Integer) return Boolean;

   procedure Treat_Value (V : out Integer);

end Memories;
```

```
with Memories; use Memories;

procedure Read_Record (Memory : Chunk; From : Integer)
  with SPARK_Mode => On,
       Pre => From in Memory'First .. Memory'Last - 2
is
   function Read_One (First : Integer; Offset : Integer) return Integer
     with Pre => First + Offset in Memory'Range
   is
      Value : Integer := Memory (First + Offset);
   begin
      if Is_Too_Coarse (Value) then
         Treat_Value (Value);
      end if;
      return Value;
   end Read_One;

   Data1, Data2 : Integer;

begin
   Data1 := Read_One (From, 1);
   Data2 := Read_One (From, 2);
end Read_Record;
```

This example is correct, but it can't be verified by GNATprove, which analyses `Read_One` on its own and notices that an overflow may occur in its precondition in certain contexts.

3.5.7 Example #7

Let's rewrite the precondition of `Read_One` to avoid any possible overflow.

```
package Memories is

   type Chunk is array (Integer range <>) of Integer
     with Predicate => Chunk'Length >= 10;

   function Is_Too_Coarse (V : Integer) return Boolean;

   procedure Treat_Value (V : out Integer);

end Memories;
```

```
with Memories; use Memories;

procedure Read_Record (Memory : Chunk; From : Integer)
  with SPARK_Mode => On,
       Pre => From in Memory'First .. Memory'Last - 2
is
   function Read_One (First : Integer; Offset : Integer) return Integer
     with Pre => First >= Memory'First
           and then Offset in 0 .. Memory'Last - First
   is
      Value : Integer := Memory (First + Offset);
   begin
      if Is_Too_Coarse (Value) then
         Treat_Value (Value);
      end if;
      return Value;
   end Read_One;

   Data1, Data2 : Integer;

begin
   Data1 := Read_One (From, 1);
   Data2 := Read_One (From, 2);
end Read_Record;
```

This example is also not correct: unfortunately, our attempt to correct `Read_One`'s precondition failed. For example, an overflow will occur at runtime if `First` is `Integer'Last` and `Memory'Last` is negative. This is possible here because type `Chunk` uses `Integer` as base index type instead of `Natural` or `Positive`.

3.5.8 Example #8

Let's completely remove the precondition of `Read_One`.

```
package Memories is

   type Chunk is array (Integer range <>) of Integer
     with Predicate => Chunk'Length >= 10;

   function Is_Too_Coarse (V : Integer) return Boolean;

   procedure Treat_Value (V : out Integer);

end Memories;
```

```
with Memories; use Memories;

procedure Read_Record (Memory : Chunk; From : Integer)
```

(continues on next page)

(continued from previous page)

```
  with SPARK_Mode => On,
       Pre => From in Memory'First .. Memory'Last - 2
is
   function Read_One (First : Integer; Offset : Integer) return Integer is
      Value : Integer := Memory (First + Offset);
   begin
      if Is_Too_Coarse (Value) then
         Treat_Value (Value);
      end if;
      return Value;
   end Read_One;

   Data1, Data2 : Integer;

begin
   Data1 := Read_One (From, 1);
   Data2 := Read_One (From, 2);
end Read_Record;
```

This example is correct and fully proved. We could have fixed the contract of `Read_One` to correctly handle both positive and negative values of `Memory'Last`, but we found it simpler to let the function be inlined for proof by removing its precondition.

3.5.9 Example #9

The procedure `Compute` performs various computations on its argument. The computation performed depends on its input range and is reflected in its contract, which we express using a `Contract_Cases` aspect.

```
procedure Compute (X : in out Integer) with
  Contract_Cases => ((X in -100 .. 100) => X = X'Old * 2,
                     (X in    0 .. 199) => X = X'Old + 1,
                     (X in -199 .. 0)   => X = X'Old - 1,
                      X >=  200          => X =  200,
                      others             => X = -200)
is
begin
   if X in -100 .. 100 then
      X := X * 2;
   elsif X in 0 .. 199 then
      X := X + 1;
   elsif X in -199 .. 0 then
      X := X - 1;
   elsif X >= 200 then
      X := 200;
   else
      X := -200;
   end if;
end Compute;
```

This example isn't correct. We duplicated the content of `Compute`'s body in its contract. This is incorrect because the semantics of `Contract_Cases` require disjoint cases, just like a case statement. The counterexample returned by GNATprove shows that `X = 0` is covered by two different case-guards (the first and the second).

3.5.10 Example #10

Let's rewrite the contract of `Compute` to avoid overlapping cases.

```
procedure Compute (X : in out Integer) with
  Contract_Cases => ((X in    0 ..  199) => X >= X'Old,
                     (X in -199 ..   -1) => X <= X'Old,
                      X >=  200          => X =  200,
                      X <  -200          => X = -200)
is
begin
   if X in -100 .. 100 then
      X := X * 2;
   elsif X in 0 .. 199 then
      X := X + 1;
   elsif X in -199 .. 0 then
      X := X - 1;
   elsif X >= 200 then
      X := 200;
   else
      X := -200;
   end if;
end Compute;
```

This example is still not correct. GNATprove can successfully prove the different cases are disjoint and also successfully verify each case individually. This isn't enough, though: a `Contract_Cases` must cover all cases. Here, we forgot the value -200, which is what GNATprove reports in its counterexample.

CHAPTER

FOUR

STATE ABSTRACTION

Abstraction is a key concept in programming that can drastically simplify both the implementation and maintenance of code. It's particularly well suited to SPARK and its modular analysis. This section explains what state abstraction is and how you use it in SPARK. We explain how it impacts GNATprove's analysis both in terms of information flow and proof of program properties.

State abstraction allows us to:

- express dependencies that wouldn't otherwise be expressible because some data that's read or written isn't visible at the point where a subprogram is declared — examples are dependencies on data, for which we use the `Global` contract, and on flow, for which we use the `Depends` contract.
- reduce the number of variables that need to be considered in flow analysis and proof, a reduction which may be critical in order to scale the analysis to programs with thousands of global variables.

4.1 What's an Abstraction?

Abstraction is an important part of programming language design. It provides two views of the same object: an abstract one and a refined one. The abstract one — usually called *specification* — describes what the object does in a coarse way. A subprogram's specification usually describes how it should be called (e.g., parameter information such as how many and of what types) as well as what it does (e.g., returns a result or modifies one or more of its parameters).

Contract-based programming, as supported in Ada, allows contracts to be added to a subprogram's specification. You use contracts to describe the subprogram's behavior in a more fine-grained manner, but all the details of how the subprogram actually works are left to its refined view, its implementation.

Take a look at the example code shown below.

```
procedure Increase (X : in out Integer) with
  Global => null,
  Pre    => X <= 100,
  Post   => X'Old < X;
```

```
procedure Increase (X : in out Integer) is
begin
   X := X + 1;
end Increase;
```

We've written a specification of the subprogram `Increase` to say that it's called with a single argument, a variable of type `Integer` whose initial value is less than 100. Our contract says that the only effect of the subprogram is to increase the value of its argument.

4.2 Why is Abstraction Useful?

A good abstraction of a subprogram's implementation is one whose specification precisely and completely summarizes what its callers can rely on. In other words, a caller of that subprogram shouldn't rely on any behavior of its implementation if that behavior isn't documented in its specification.

For example, callers of the subprogram `Increase` can assume that it always strictly increases the value of its argument. In the code snippet shown below, this means the loop must terminate.

```
procedure Increase (X : in out Integer) with
  Global => null,
  Pre    => X <= 100,
  Post   => X'Old < X;
```

```
with Increase;
procedure Client is
   X : Integer := 0;
begin
   while X <= 100 loop      --  The loop will terminate
      Increase (X);         --  Increase can be called safely
   end loop;
   pragma Assert (X = 101); --  Will this hold?
end Client;
```

Callers can also assume that the implementation of `Increase` won't cause any runtime errors when called in the loop. On the other hand, nothing in the specification guarantees that the assertion show above is correct: it may fail if `Increase`'s implementation is changed.

If you follow this basic principle, abstraction can bring you significant benefits. It simplifies both your program's implementation and verification. It also makes maintenance and code reuse much easier since changes to the implementation of an object shouldn't affect the code using this object. Your goal in using it is that it should be enough to understand the specification of an object in order to use that object, since understanding the specification is usually much simpler than understanding the implementation.

GNATprove relies on the abstraction defined by subprogram contracts and therefore doesn't prove the assertion after the loop in `Client` above.

4.3 Abstraction of a Package's State

Subprograms aren't the only objects that benefit from abstraction. The state of a package — the set of persistent variables defined in it — can also be hidden from external users. You achieve this form of abstraction — called *state abstraction* — by defining variables in the body or private part of a package so they can only be accessed through subprogram calls. For example, our `Stack` package shown below provides an abstraction for a `Stack` object which can only be modified using the `Pop` and `Push` procedures.

```
package Stack is
   procedure Pop  (E : out Element);
   procedure Push (E : in  Element);
end Stack;

package body Stack is
   Content : Element_Array (1 .. Max);
   Top     : Natural;
   ...
end Stack;
```

The fact that we implemented it using an array is irrelevant to the caller. We could change that without impacting our callers' code.

4.4 Declaring a State Abstraction

Hidden state influences a program's behavior, so SPARK allows that state to be declared. You can use the `Abstract_State` aspect, an abstraction that names a state, to do this, but you aren't required to use it even for a package with hidden state. You can use several state abstractions to declare the hidden state of a single package or you can use it for a package with no hidden state at all. However, since SPARK doesn't allow aliasing, different state abstractions must always refer to disjoint sets of variables. A state abstraction isn't a variable: it doesn't have a type and can't be used inside expressions, either those in bodies or contracts.

As an example of the use of this aspect, we can optionally define a state abstraction for the entire hidden state of the `Stack` package like this:

```
package Stack with
  Abstract_State => The_Stack
is
  ...
```

Alternatively, we can define a state abstraction for each hidden variable:

```
package Stack with
  Abstract_State => (Top_State, Content_State)
is
  ...
```

Remember: a state abstraction isn't a variable (it has no type) and can't be used inside expressions. For example:

```
pragma Assert (Stack.Top_State = ...);
-- compilation error: Top_State is not a variable
```

4.5 Refining an Abstract State

Once you've declared an abstract state in a package, you must refine it into its constituents using a `Refined_State` aspect. You must place the `Refined_State` aspect on the package body even if the package wouldn't otherwise have required a body. For each state abstraction you've declared for the package, you list the set of variables represented by that state abstraction in its refined state.

If you specify an abstract state for a package, it must be complete, meaning you must have listed every hidden variable as part of some state abstraction. For example, we must add a `Refined_State` aspect on our `Stack` package's body linking the state abstraction (`The_Stack`) to the entire hidden state of the package, which consists of both `Content` and `Top`.

```
package Stack with
  Abstract_State => The_Stack
is
   type Element is new Integer;

   procedure Pop  (E : out Element);
   procedure Push (E : Element);

end Stack;
```

```
package body Stack with
  Refined_State => (The_Stack => (Content, Top))
is
   Max : constant := 100;

   type Element_Array is array (1 .. Max) of Element;

   Content : Element_Array := (others => 0);
   Top     : Natural range 0 .. Max := 0;
   --  Both Content and Top must be listed in the list of
   --  constituents of The_Stack

   procedure Pop (E : out Element) is
   begin
      E   := Content (Top);
      Top := Top - 1;
   end Pop;

   procedure Push (E : Element) is
   begin
      Top           := Top + 1;
      Content (Top) := E;
   end Push;

end Stack;
```

4.6 Representing Private Variables

You can refine state abstractions in the package body, where all the variables are visible. When only the package's specification is available, you need a way to specify which state abstraction each private variable belongs to. You do this by adding the `Part_Of` aspect to the variable's declaration.

`Part_Of` annotations are mandatory: if you gave a package an abstract state annotation, you must link all the hidden variables defined in its private part to a state abstraction. For example:

```
package Stack with
  Abstract_State => The_Stack
is
   type Element is new Integer;

   procedure Pop  (E : out Element);
   procedure Push (E : Element);

private

   Max : constant := 100;

   type Element_Array is array (1 .. Max) of Element;

   Content : Element_Array          with Part_Of => The_Stack;
   Top     : Natural range 0 .. Max with Part_Of => The_Stack;

end Stack;
```

Since we chose to define `Content` and `Top` in `Stack`'s private part instead of its body, we had to add a `Part_Of` aspect to both of their declarations, associating them with the state abstraction `The_Stack`, even though it's the only state abstraction. However, we still need to list them in the `Refined_State` aspect in `Stack`'s body.

```
package body Stack with
  Refined_State => (The_Stack => (Content, Top))
```

4.7 Additional State

4.7.1 Nested Packages

So far, we've only discussed hidden variables. But variables aren't the only component of a package's state. If a package P contains a nested package, the nested package's state is also part of P's state. If the nested package is hidden, its state is part of P's hidden state and must be listed in P's state refinement.

We see this in the example below, where the package `Hidden_Nested`'s hidden state is part of P's hidden state.

```
package P with
   Abstract_State => State
is
   package Visible_Nested with
      Abstract_State => Visible_State
   is
      procedure Get (E : out Integer);
   end Visible_Nested;
end P;
```

```
package body P with
   Refined_State => (State => Hidden_Nested.Hidden_State)
is
   package Hidden_Nested with
      Abstract_State => Hidden_State,
      Initializes    => Hidden_State
   is
      function Get return Integer;
   end Hidden_Nested;

   package body Hidden_Nested with
      Refined_State => (Hidden_State => Cnt)
   is
      Cnt : Integer := 0;

      function Get return Integer is (Cnt);
   end Hidden_Nested;

   package body Visible_Nested with
      Refined_State => (Visible_State => Checked)
   is
      Checked : Boolean := False;

      procedure Get (E : out Integer) is
      begin
         Checked := True;
         E := Hidden_Nested.Get;
      end Get;
   end Visible_Nested;
end P;
```

Any visible state of `Hidden_Nested` would also have been part of P's hidden state. However, if P contains a visible nested package, that nested package's state isn't part of P's hidden state.

Instead, you should declare that package's hidden state in a separate state abstraction on its own declaration, like we did above for `Visible_Nested`.

4.7.2 Constants that Depend on Variables

Some constants are also possible components of a state abstraction. These are constants whose value depends either on a variable or a subprogram parameter. They're handled as variables during flow analysis because they participate in the flow of information between variables throughout the program. Therefore, GNATprove considers these constants to be part of a package's state just like it does for variables.

If you've specified a state abstraction for a package, you must list such hidden constants declared in that package in the state abstraction refinement. However, constants that don't depend on variables don't participate in the flow of information and must not appear in a state refinement.

Let's look at this example.

```
package Stack with
  Abstract_State => The_Stack
is
   type Element is new Integer;

   procedure Pop  (E : out Element);
   procedure Push (E : Element);
end Stack;
```

```
package Configuration with
  Initializes => External_Variable
is
   External_Variable : Positive with Volatile;
end Configuration;
```

```
with Configuration;
pragma Elaborate (Configuration);

package body Stack with
  Refined_State => (The_Stack => (Content, Top, Max))
  --  Max has variable inputs. It must appear as a
  --  constituent of The_Stack
is
   Max : constant Positive := Configuration.External_Variable;

   type Element_Array is array (1 .. Max) of Element;

   Content : Element_Array := (others => 0);
   Top     : Natural range 0 .. Max := 0;

   procedure Pop (E : out Element) is
   begin
      E   := Content (Top);
      Top := Top - 1;
   end Pop;

   procedure Push (E : Element) is
   begin
      Top           := Top + 1;
      Content (Top) := E;
   end Push;

end Stack;
```

Here, Max — the maximum number of elements that can be stored in the stack — is initialized from a variable in an external package. Because of this, we must include Max as part of the state abstraction The_Stack.

4.8 Subprogram Contracts

4.8.1 Global and Depends

Hidden variables can only be accessed through subprogram calls, so you document how state abstractions are modified during the program's execution via the contracts of those subprograms. You use Global and Depends contracts to specify which of the state abstractions are used by a subprogram and how values flow through the different variables. The Global and Depends contracts that you write when referring to state abstractions are often less precise than contracts referring to visible variables since the possibly different dependencies of the hidden variables contained within a state abstraction are collapsed into a single dependency.

Let's add Global and Depends contracts to the Pop procedure in our stack.

```
package Stack with
   Abstract_State => (Top_State, Content_State)
is
   type Element is new Integer;

   procedure Pop (E : out Element) with
     Global  => (Input  => Content_State,
                 In_Out => Top_State),
     Depends => (Top_State => Top_State,
                 E         => (Content_State, Top_State));

end Stack;
```

In this example, the Pop procedure only modifies the value of the hidden variable Top, while Content is unchanged. By using distinct state abstractions for the two variables, we're able to preserve this semantic in the contract.

Let's contrast this example with a different representation of Global and Depends contracts, this time using a single abstract state.

```
package Stack with
  Abstract_State => The_Stack
is
   type Element is new Integer;

   procedure Pop  (E : out Element) with
     Global  => (In_Out => The_Stack),
     Depends => ((The_Stack, E) => The_Stack);

end Stack;
```

Here, Top_State and Content_State are merged into a single state abstraction, The_Stack. By doing so, we've hidden the fact that Content isn't modified (though we're still showing that Top may be modified). This loss in precision is reasonable here, since it's the whole point of the abstraction. However, you must be careful not to aggregate unrelated hidden state because this risks their annotations becoming meaningless.

Even though imprecise contracts that consider state abstractions as a whole are perfectly reasonable for users of a package, you should write Global and Depends contracts that are as precise as possible within the package body. To allow this, SPARK introduces the notion of *refined contracts*, which are precise contracts specified on the bodies of subprograms where state refinements are

visible. These contracts are the same as normal `Global` and `Depends` contracts except they refer directly to the hidden state of the package.

When a subprogram is called inside the package body, you should write refined contracts instead of the general ones so that the verification can be as precise as possible. However, refined `Global` and `Depends` are optional: if you don't specify them, GNATprove will compute them to check the package's implementation.

For our `Stack` example, we could add refined contracts as shown below.

```
package Stack with
  Abstract_State => The_Stack
is
   type Element is new Integer;

   procedure Pop  (E : out Element) with
     Global  => (In_Out => The_Stack),
     Depends => ((The_Stack, E) => The_Stack);

   procedure Push (E : Element) with
     Global  => (In_Out    => The_Stack),
     Depends => (The_Stack => (The_Stack, E));

end Stack;
```

```
package body Stack with
  Refined_State => (The_Stack => (Content, Top))
is
   Max : constant := 100;

   type Element_Array is array (1 .. Max) of Element;

   Content : Element_Array := (others => 0);
   Top     : Natural range 0 .. Max := 0;

   procedure Pop (E : out Element) with
     Refined_Global  => (Input  => Content,
                         In_Out => Top),
     Refined_Depends => (Top => Top,
                         E   => (Content, Top))
   is
   begin
      E   := Content (Top);
      Top := Top - 1;
   end Pop;

   procedure Push (E : Element) with
     Refined_Global  => (In_Out => (Content, Top)),
     Refined_Depends => (Content =>+ (Content, Top, E),
                         Top     => Top) is
   begin
     Top := Top + 1;
     Content (Top) := E;
   end Push;

end Stack;
```

4.8.2 Preconditions and Postconditions

We mostly express functional properties of subprograms using preconditions and postconditions. These are standard Boolean expressions, so they can't directly refer to state abstractions. To work

around this restriction, we can define functions to query the value of hidden variables. We then use these functions in place of the state abstraction in the contract of other subprograms.

For example, we can query the state of the stack with functions `Is_Empty` and `Is_Full` and call these in the contracts of procedures `Pop` and `Push`:

```
package Stack is
   type Element is new Integer;

   function Is_Empty return Boolean;
   function Is_Full  return Boolean;

   procedure Pop (E : out Element) with
     Pre  => not Is_Empty,
     Post => not Is_Full;

   procedure Push (E : Element) with
     Pre  => not Is_Full,
     Post => not Is_Empty;

end Stack;
```

```
package body Stack is

   Max : constant := 100;

   type Element_Array is array (1 .. Max) of Element;

   Content : Element_Array := (others => 0);
   Top     : Natural range 0 .. Max := 0;

   function Is_Empty return Boolean is (Top = 0);
   function Is_Full  return Boolean is (Top = Max);

   procedure Pop (E : out Element) is
   begin
      E   := Content (Top);
      Top := Top - 1;
   end Pop;

   procedure Push (E : Element) is
   begin
      Top           := Top + 1;
      Content (Top) := E;
   end Push;

end Stack;
```

Just like we saw for `Global` and `Depends` contracts, you may often find it useful to have a more precise view of functional contracts in the context where the hidden variables are visible. You do this using expression functions in the same way we did for the functions `Is_Empty` and `Is_Full` above. As expression function, bodies act as contracts for GNATprove, so they automatically give a more precise version of the contracts when their implementation is visible.

You may often need a more constraining contract to verify the package's implementation but want to be less strict outside the abstraction. You do this using the `Refined_Post` aspect. This aspect, when placed on a subprogram's body, provides stronger guaranties to internal callers of a subprogram. If you provide one, the refined postcondition must imply the subprogram's postcondition. This is checked by GNATprove, which reports a failing postcondition if the refined postcondition is too weak, even if it's actually implied by the subprogram's body. SPARK doesn't peform a similar verification for normal preconditions.

For example, we can refine the postconditions in the bodies of `Pop` and `Push` to be more detailed

than what we wrote for them in their specification.

```
package Stack is
   type Element is new Integer;

   function Is_Empty return Boolean;
   function Is_Full  return Boolean;

   procedure Pop (E : out Element) with
     Pre  => not Is_Empty,
     Post => not Is_Full;

   procedure Push (E : Element) with
     Pre  => not Is_Full,
     Post => not Is_Empty;

end Stack;
```

```
package body Stack is

   Max : constant := 100;

   type Element_Array is array (1 .. Max) of Element;

   Content : Element_Array := (others => 0);
   Top     : Natural range 0 .. Max := 0;

   function Is_Empty return Boolean is (Top = 0);
   function Is_Full  return Boolean is (Top = Max);

   procedure Pop (E : out Element) with
     Refined_Post => not Is_Full and E = Content (Top)'Old
   is
   begin
      E   := Content (Top);
      Top := Top - 1;
   end Pop;

   procedure Push (E : Element) with
     Refined_Post => not Is_Empty and E = Content (Top)
   is
   begin
      Top           := Top + 1;
      Content (Top) := E;
   end Push;

end Stack;
```

4.9 Initialization of Local Variables

As part of flow analysis, GNATprove checks for the proper initialization of variables. Therefore, flow analysis needs to know which variables are initialized during the package's elaboration.

You can use the `Initializes` aspect to specify the set of visible variables and state abstractions that are initialized during the elaboration of a package. An `Initializes` aspect can't refer to a variable that isn't defined in the unit since, in SPARK, a package can only initialize variables declared immediately within the package.

`Initializes` aspects are optional. If you don't supply any, they'll be derived by GNATprove.

For our `Stack` example, we could add an `Initializes` aspect.

```
package Stack with
  Abstract_State => The_Stack,
  Initializes    => The_Stack
is
   type Element is new Integer;

   procedure Pop  (E : out Element);

end Stack;
```

```
package body Stack with
  Refined_State => (The_Stack => (Content, Top))
is
   Max : constant := 100;

   type Element_Array is array (1 .. Max) of Element;

   Content : Element_Array := (others => 0);
   Top     : Natural range 0 .. Max := 0;

   procedure Pop (E : out Element) is
   begin
      E   := Content (Top);
      Top := Top - 1;
   end Pop;

end Stack;
```

Flow analysis also checks for dependencies between variables, so it must be aware of how information flows through the code that performs the initialization of states. We discussed one use of the `Initializes` aspect above. But you also can use it to provide flow information. If the initial value of a variable or state abstraction is dependent on the value of another visible variable or state abstraction from another package, you must list this dependency in the `Initializes` contract. You specify the list of entities on which a variable's initial value depends using an arrow following that variable's name.

Let's look at this example:

```
package Q is
   External_Variable : Integer := 2;
end Q;
```

```
with Q;
package P with
  Initializes => (V1, V2 => Q.External_Variable)
is
   V1 : Integer := 0;
   V2 : Integer := Q.External_Variable;
end P;
```

Here we indicated that `V2`'s initial value depends on the value of `Q.External_Variable` by including that dependency in the `Initializes` aspect of `P`. We didn't list any dependency for `V1` because its initial value doesn't depend on any external variable. We could also have stated that lack of dependency explicitly by writing `V1 => null`.

GNATprove computes dependencies of initial values if you don't supply an `Initializes` aspect. However, if you do provide an `Initializes` aspect for a package, it must be complete: you must list every initialized state of the package, along with all its external dependencies.

4.10 Code Examples / Pitfalls

This section contains some code examples to illustrate potential pitfalls.

4.10.1 Example #1

Package `Communication` defines a hidden local package, `Ring_Buffer`, whose capacity is initialized from an external configuration during elaboration.

```
package Configuration is

   External_Variable : Natural := 1;

end Configuration;
```

```
with Configuration;

package Communication with
  Abstract_State => State,
  Initializes    => (State => Configuration.External_Variable)
is
   function Get_Capacity return Natural;

private

   package Ring_Buffer with
     Initializes => (Capacity => Configuration.External_Variable)
   is
      Capacity : constant Natural := Configuration.External_Variable;
   end Ring_Buffer;

end Communication;
```

```
package body Communication with
  Refined_State => (State => Ring_Buffer.Capacity)
is

   function Get_Capacity return Natural is
   begin
      return Ring_Buffer.Capacity;
   end Get_Capacity;

end Communication;
```

This example isn't correct. `Capacity` is declared in the private part of `Communication`. Therefore, we should have linked it to `State` by using the `Part_Of` aspect in its declaration.

4.10.2 Example #2

Let's add `Part_Of` to the state of hidden local package `Ring_Buffer`, but this time we hide variable `Capacity` inside the private part of `Ring_Buffer`.

```
package Configuration is

   External_Variable : Natural := 1;

end Configuration;
```

```
with Configuration;

package Communication with
  Abstract_State => State
is
private

   package Ring_Buffer with
     Abstract_State => (B_State with Part_Of => State),
     Initializes    => (B_State => Configuration.External_Variable)
   is
      function Get_Capacity return Natural;
   private
      Capacity : constant Natural := Configuration.External_Variable
        with Part_Of => B_State;
   end Ring_Buffer;

end Communication;
```

```
package body Communication with
  Refined_State => (State => Ring_Buffer.B_State)
is

   package body Ring_Buffer with
      Refined_State => (B_State => Capacity)
   is
      function Get_Capacity return Natural is (Capacity);
   end Ring_Buffer;

end Communication;
```

This program is correct and GNATprove is able to verify it.

4.10.3 Example #3

Package `Counting` defines two counters: `Black_Counter` and `Red_Counter`. It provides separate initialization procedures for each, both called from the main procedure.

```
package Counting with
  Abstract_State => State
is
   procedure Reset_Black_Count;
   procedure Reset_Red_Count;
end Counting;
```

```
package body Counting with
  Refined_State => (State => (Black_Counter, Red_Counter))
is
   Black_Counter, Red_Counter : Natural;

   procedure Reset_Black_Count is
   begin
      Black_Counter := 0;
   end Reset_Black_Count;

   procedure Reset_Red_Count is
   begin
      Red_Counter := 0;
   end Reset_Red_Count;
end Counting;
```

```
with Counting; use Counting;

procedure Main is
begin
   Reset_Black_Count;
   Reset_Red_Count;
end Main;
```

This program doesn't read any uninitialized data, but GNATprove fails to verify that. This is because we provided a state abstraction for package `Counting`, so flow analysis computes the effects of subprograms in terms of this state abstraction and thus considers `State` to be an in-out global consisting of both `Reset_Black_Counter` and `Reset_Red_Counter`. So it issues the message requiring that `State` be initialized after elaboration as well as the warning that no procedure in package `Counting` can initialize its state.

4.10.4 Example #4

Let's remove the abstract state on package `Counting`.

```
package Counting is
   procedure Reset_Black_Count;
   procedure Reset_Red_Count;
end Counting;
```

```
package body Counting is
   Black_Counter, Red_Counter : Natural;

   procedure Reset_Black_Count is
   begin
      Black_Counter := 0;
   end Reset_Black_Count;

   procedure Reset_Red_Count is
   begin
      Red_Counter := 0;
   end Reset_Red_Count;
end Counting;
```

```
with Counting; use Counting;

procedure Main is
begin
   Reset_Black_Count;
   Reset_Red_Count;
end Main;
```

This example is correct. Because we didn't provide a state abstraction, GNATprove reasons in terms of variables, instead of states, and proves data initialization without any problem.

4.10.5 Example #5

Let's restore the abstract state to package `Counting`, but this time provide a procedure `Reset_All` that calls the initialization procedures `Reset_Black_Counter` and `Reset_Red_Counter`.

```
package Counting with
  Abstract_State => State
is
```

(continues on next page)

(continued from previous page)

```
   procedure Reset_Black_Count with Global => (In_Out => State);
   procedure Reset_Red_Count   with Global => (In_Out => State);
   procedure Reset_All         with Global => (Output => State);
end Counting;
```

```
package body Counting with
  Refined_State => (State => (Black_Counter, Red_Counter))
is
   Black_Counter, Red_Counter : Natural;

   procedure Reset_Black_Count is
   begin
      Black_Counter := 0;
   end Reset_Black_Count;

   procedure Reset_Red_Count is
   begin
      Red_Counter := 0;
   end Reset_Red_Count;

   procedure Reset_All is
   begin
      Reset_Black_Count;
      Reset_Red_Count;
   end Reset_All;
end Counting;
```

This example is correct. Flow analysis computes refined versions of `Global` contracts for internal calls and uses these to verify that `Reset_All` indeed properly initializes `State`. The `Refined_Global` and `Global` annotations are not mandatory and can be computed by GNATprove.

4.10.6 Example #6

Let's consider yet another version of our abstract stack unit.

```
package Stack with
  Abstract_State => The_Stack
is
   pragma Unevaluated_Use_Of_Old (Allow);

   type Element is new Integer;

   type Element_Array is array (Positive range <>) of Element;
   Max : constant Natural := 100;
   subtype Length_Type is Natural range 0 .. Max;

   procedure Push (E : Element) with
     Post =>
       not Is_Empty and
       (if Is_Full'Old then The_Stack = The_Stack'Old else Peek = E);

   function Peek     return Element with Pre => not Is_Empty;
   function Is_Full  return Boolean;
   function Is_Empty return Boolean;
end Stack;
```

```
package body Stack with
  Refined_State => (The_Stack => (Top, Content))
is
   Top     : Length_Type := 0;
   Content : Element_Array (1 .. Max) := (others => 0);

   procedure Push (E : Element) is
   begin
      Top           := Top + 1;
      Content (Top) := E;
   end Push;

   function  Peek     return Element is (Content (Top));
   function  Is_Full  return Boolean is (Top >= Max);
   function  Is_Empty return Boolean is (Top = 0);
end Stack;
```

This example isn't correct. There's a compilation error in `Push`'s postcondition: `The_Stack` is a state abstraction, not a variable, and therefore can't be used in an expression.

4.10.7 Example #7

In this version of our abstract stack unit, a copy of the stack is returned by function `Get_Stack`, which we call in the postcondition of `Push` to specify that the stack shouldn't be modified if it's full. We also assert that after we push an element on the stack, either the stack is unchanged (if it was already full) or its top element is equal to the element just pushed.

```
package Stack with
  Abstract_State => The_Stack
is
   pragma Unevaluated_Use_Of_Old (Allow);

   type Stack_Model is private;

   type Element is new Integer;
   type Element_Array is array (Positive range <>) of Element;
   Max : constant Natural := 100;
   subtype Length_Type is Natural range 0 .. Max;

   function Peek      return Element with Pre => not Is_Empty;
   function Is_Full   return Boolean;
   function Is_Empty  return Boolean;
   function Get_Stack return Stack_Model;

   procedure Push (E : Element) with
     Post => not Is_Empty and
       (if Is_Full'Old then Get_Stack = Get_Stack'Old else Peek = E);

private

   type Stack_Model is record
      Top     : Length_Type := 0;
      Content : Element_Array (1 .. Max) := (others => 0);
   end record;

end Stack;
```

```
package body Stack with
  Refined_State => (The_Stack => (Top, Content))
```

(continues on next page)

(continued from previous page)

```
is
   Top     : Length_Type := 0;
   Content : Element_Array (1 .. Max) := (others => 0);

   procedure Push (E : Element) is
   begin
      if Top >= Max then
         return;
      end if;
      Top             := Top + 1;
      Content (Top) := E;
   end Push;

   function Peek     return Element is (Content (Top));
   function Is_Full  return Boolean is (Top >= Max);
   function Is_Empty return Boolean is (Top = 0);

   function Get_Stack return Stack_Model is (Stack_Model'(Top, Content));

end Stack;
```

```
with Stack; use Stack;

procedure Use_Stack (E : Element) with
  Pre => not Is_Empty
is
   F : Element := Peek;
begin
   Push (E);
   pragma Assert (Peek = E or Peek = F);
end Use_Stack;
```

This program is correct, but GNATprove can't prove the assertion in `Use_Stack`. Indeed, even if `Get_Stack` is an expression function, its body isn't visible outside of `Stack`'s body, where it's defined.

4.10.8 Example #8

Let's move the definition of `Get_Stack` and other expression functions inside the private part of the spec of `Stack`.

```
package Stack with
  Abstract_State => The_Stack
is
   pragma Unevaluated_Use_Of_Old (Allow);

   type Stack_Model is private;

   type Element is new Integer;
   type Element_Array is array (Positive range <>) of Element;
   Max : constant Natural := 100;
   subtype Length_Type is Natural range 0 .. Max;

   function Peek      return Element with Pre => not Is_Empty;
   function Is_Full   return Boolean;
   function Is_Empty  return Boolean;
   function Get_Stack return Stack_Model;

   procedure Push (E : Element) with
```

(continues on next page)

(continued from previous page)

```
     Post => not Is_Empty and
       (if Is_Full'Old then Get_Stack = Get_Stack'Old else Peek = E);

private

   Top     : Length_Type              := 0 with Part_Of => The_Stack;
   Content : Element_Array (1 .. Max) := (others => 0) with
     Part_Of => The_Stack;

   type Stack_Model is record
      Top     : Length_Type := 0;
      Content : Element_Array (1 .. Max) := (others => 0);
   end record;

   function Peek      return Element     is (Content (Top));
   function Is_Full   return Boolean     is (Top >= Max);
   function Is_Empty  return Boolean     is (Top = 0);

   function Get_Stack return Stack_Model is (Stack_Model'(Top, Content));

end Stack;
```

```
package body Stack with
  Refined_State => (The_Stack => (Top, Content))
is

   procedure Push (E : Element) is
   begin
      if Top >= Max then
         return;
      end if;
      Top             := Top + 1;
      Content (Top) := E;
   end Push;

end Stack;
```

```
with Stack; use Stack;

procedure Use_Stack (E : Element) with
  Pre => not Is_Empty
is
   F : Element := Peek;
begin
   Push (E);
   pragma Assert (Peek = E or Peek = F);
end Use_Stack;
```

This example is correct. GNATprove can verify the assertion in `Use_Stack` because it has visibility to `Get_Stack`'s body.

4.10.9 Example #9

Package `Data` defines three variables, `Data_1`, `Data_2` and `Data_3`, that are initialized at elaboration (in `Data`'s package body) from an external interface that reads the file system.

```
package External_Interface with
  Abstract_State => File_System,
```

(continues on next page)

(continued from previous page)

```
  Initializes    => File_System
is
   type Data_Type_1 is new Integer;
   type Data_Type_2 is new Integer;
   type Data_Type_3 is new Integer;

   type Data_Record is record
      Field_1 : Data_Type_1;
      Field_2 : Data_Type_2;
      Field_3 : Data_Type_3;
   end record;

   procedure Read_Data (File_Name : String; Data : out Data_Record)
     with Global => File_System;
end External_Interface;
```

```
with External_Interface; use External_Interface;

package Data with
  Initializes => (Data_1, Data_2, Data_3)
is
   pragma Elaborate_Body;

   Data_1 : Data_Type_1;
   Data_2 : Data_Type_2;
   Data_3 : Data_Type_3;

end Data;
```

```
with External_Interface;
pragma Elaborate_All (External_Interface);

package body Data is
begin
   declare
      Data_Read : Data_Record;
   begin
      Read_Data ("data_file_name", Data_Read);
      Data_1 := Data_Read.Field_1;
      Data_2 := Data_Read.Field_2;
      Data_3 := Data_Read.Field_3;
   end;
end Data;
```

This example isn't correct. The dependency between `Data_1`'s initial value and `File_System` must be listed in `Data`'s `Initializes` aspect.

4.10.10 Example #10

Let's remove the `Initializes` contract on package `Data`.

```
package External_Interface with
  Abstract_State => File_System,
  Initializes    => File_System
is
   type Data_Type_1 is new Integer;
   type Data_Type_2 is new Integer;
   type Data_Type_3 is new Integer;
```

(continues on next page)

(continued from previous page)

```
   type Data_Record is record
      Field_1 : Data_Type_1;
      Field_2 : Data_Type_2;
      Field_3 : Data_Type_3;
   end record;

   procedure Read_Data (File_Name : String; Data : out Data_Record)
     with Global => File_System;
end External_Interface;
```

```
with External_Interface; use External_Interface;

package Data is
   pragma Elaborate_Body;

   Data_1 : Data_Type_1;
   Data_2 : Data_Type_2;
   Data_3 : Data_Type_3;

end Data;
```

```
with External_Interface;
pragma Elaborate_All (External_Interface);

package body Data is
begin
   declare
      Data_Read : Data_Record;
   begin
      Read_Data ("data_file_name", Data_Read);
      Data_1 := Data_Read.Field_1;
      Data_2 := Data_Read.Field_2;
      Data_3 := Data_Read.Field_3;
   end;
end Data;
```

This example is correct. Since `Data` has no `Initializes` aspect, GNATprove computes the set of variables initialized during its elaboration as well as their dependencies.

CHAPTER

FIVE

PROOF OF FUNCTIONAL CORRECTNESS

This section is dedicated to the functional correctness of programs. It presents advanced proof features that you may need to use for the specification and verification of your program's complex properties.

5.1 Beyond Program Integrity

When we speak about the *correctness* of a program or subprogram, we mean the extent to which it complies with its specification. Functional correctness is specifically concerned with properties that involve the relations between the subprogram's inputs and outputs, as opposed to other properties such as running time or memory consumption.

For functional correctness, we usually specify stronger properties than those required to just prove program integrity. When we're involved in a certification processes, we should derive these properties from the requirements of the system, but, especially in non-certification contexts, they can also come from more informal sources, such as the program's documentation, comments in its code, or test oracles.

For example, if one of our goals is to ensure that no runtime error is raised when using the result of the function `Find` below, it may be enough to know that the result is either 0 or in the range of A. We can express this as a postcondition of `Find`.

```
package Show_Find is

   type Nat_Array is array (Positive range <>) of Natural;

   function Find (A : Nat_Array; E : Natural) return Natural with
     Post => Find'Result in 0 | A'Range;

end Show_Find;
```

```
package body Show_Find is

   function Find (A : Nat_Array; E : Natural) return Natural is
   begin
      for I in A'Range loop
         if A (I) = E then
            return I;
         end if;
      end loop;
      return 0;
   end Find;

end Show_Find;
```

In this case, it's automatically proved by GNATprove.

However, to be sure that Find performs the task we expect, we may want to verify more complex properties of that function. For example, we want to ensure it returns an index of A where E is stored and returns 0 only if E is nowhere in A. Again, we can express this as a postcondition of Find.

```
package Show_Find is

   type Nat_Array is array (Positive range <>) of Natural;

   function Find (A : Nat_Array; E : Natural) return Natural with
     Post =>
       (if (for all I in A'Range => A (I) /= E)
          then Find'Result = 0
        else Find'Result in A'Range and then A (Find'Result) = E);

end Show_Find;
```

```
package body Show_Find is

   function Find (A : Nat_Array; E : Natural) return Natural is
   begin
      for I in A'Range loop
         if A (I) = E then
            return I;
         end if;
      end loop;
      return 0;
   end Find;

end Show_Find;
```

This time, GNATprove can't prove this postcondition automatically, but we'll see later that we can help GNATprove by providing a loop invariant, which is checked by GNATprove and allows it to automatically prove the postcondition for Find.

Writing at least part of your program's specification in the form of contracts has many advantages. You can execute those contracts during testing, which improves the maintainability of the code by detecting discrepancies between the program and its specification in earlier stages of development. If the contracts are precise enough, you can use them as oracles to decide whether a given test passed or failed. In that case, they can allow you to verify the outputs of specific subprograms while running a larger block of code. This may, in certain contexts, replace the need for you to perform unit testing, instead allowing you to run integration tests with assertions enabled. Finally, if the code is in SPARK, you can also use GNATprove to formally prove these contracts.

The advantage of a formal proof is that it verifies all possible execution paths, something which isn't always possible by running test cases. For example, during testing, the postcondition of the subprogram Find shown below is checked dynamically for the set of inputs for which Find is called in that test, but just for that set.

```
package Show_Find is

   type Nat_Array is array (Positive range <>) of Natural;

   function Find (A : Nat_Array; E : Natural) return Natural with
     Post =>
       (if (for all I in A'Range => A (I) /= E)
          then Find'Result = 0
        else Find'Result in A'Range and then A (Find'Result) = E);

end Show_Find;
```

```
package body Show_Find is

   function Find (A : Nat_Array; E : Natural) return Natural is
   begin
      for I in A'Range loop
         if A (I) = E then
            return I;
         end if;
      end loop;
      return 0;
   end Find;

end Show_Find;
```

```
with Ada.Text_IO; use Ada.Text_IO;
with Show_Find; use Show_Find;

procedure Use_Find with
  SPARK_Mode => Off
is
   Seq : constant Nat_Array (1 .. 3) := (1, 5, 3);
   Res : Natural;
begin
   Res := Find (Seq, 3);
   Put_Line ("Found 3 in index #" & Natural'Image (Res) & " of array");
end Use_Find;
```

However, if `Find` is formally verified, that verification checks its postcondition for all possible inputs. During development, you can attempt such verification earlier than testing since it's performed modularly on a per-subprogram basis. For example, in the code shown above, you can formally verify `Use_Find` even before you write the body for subprogram `Find`.

5.2 Advanced Contracts

Contracts for functional correctness are usually more complex than contracts for program integrity, so they more often require you to use the new forms of expressions introduced by the Ada 2012 standard. In particular, quantified expressions, which allow you to specify properties that must hold for all or for at least one element of a range, come in handy when specifying properties of arrays.

As contracts become more complex, you may find it useful to introduce new abstractions to improve the readability of your contracts. Expression functions are a good means to this end because you can retain their bodies in your package's specification.

Finally, some properties, especially those better described as invariants over data than as properties of subprograms, may be cumbersome to express as subprogram contracts. Type predicates, which must hold for every object of a given type, are usually a better match for this purpose. Here's an example.

```
package Show_Sort is

   type Nat_Array is array (Positive range <>) of Natural;

   function Is_Sorted (A : Nat_Array) return Boolean is
     (for all I in A'Range =>
        (if I < A'Last then A (I) <= A (I + 1)));
   --  Returns True if A is sorted in increasing order.
```

(continues on next page)

(continued from previous page)

```
   subtype Sorted_Nat_Array is Nat_Array with
     Dynamic_Predicate => Is_Sorted (Sorted_Nat_Array);
   --  Elements of type Sorted_Nat_Array are all sorted.

   Good_Array : Sorted_Nat_Array := (1, 2, 4, 8, 42);
end Show_Sort;
```

We can use the subtype `Sorted_Nat_Array` as the type of a variable that must remain sorted throughout the program's execution. Specifying that an array is sorted requires a rather complex expression involving quantifiers, so we abstract away this property as an expression function to improve readability. `Is_Sorted`'s body remains in the package's specification and allows users of the package to retain a precise knowledge of its meaning when necessary. (You must use `Nat_Array` as the type of the operand of `Is_Sorted`. If you use `Sorted_Nat_Array`, you'll get infinite recursion at runtime when assertion checks are enabled since that function is called to check all operands of type `Sorted_Nat_Array`.)

5.2.1 Ghost Code

As the properties you need to specify grow more complex, you may have entities that are only needed because they are used in specifications (contracts). You may find it important to ensure that these entities can't affect the behavior of the program or that they're completely removed from production code. This concept, having entities that are only used for specifications, is usually called having *ghost* code and is supported in SPARK by the `Ghost` aspect.

You can use `Ghost` aspects to annotate any entity including variables, types, subprograms, and packages. If you mark an entity as `Ghost`, GNATprove ensures it can't affect the program's behavior. When the program is compiled with assertions enabled, ghost code is executed like normal code so it can execute the contracts using it. You can also instruct the compiler to not generate code for ghost entities.

Consider the procedure `Do_Something` below, which calls a complex function on its input, `X`, and wants to check that the initial and modified values of `X` are related in that complex way.

```
package Show_Ghost is

   type T is record
      A, B, C, D, E : Boolean;
   end record;

   function Formula (X : T) return Boolean is
     ((X.A and X.B) or (X.C and (X.D or X.E)));

   function Is_Correct (X, Y : T) return Boolean is
     (Formula (X) = Formula (Y));

   procedure Do_Something (X : in out T);

end Show_Ghost;
```

```
package body Show_Ghost is

   procedure Do_Some_Complex_Stuff (X : in out T) is
   begin
      X := T'(X.B, X.A, X.C, X.E, X.D);
   end Do_Some_Complex_Stuff;

   procedure Do_Something (X : in out T) is
      X_Init : constant T := X with Ghost;
```

(continues on next page)

(continued from previous page)

```
   begin
      Do_Some_Complex_Stuff (X);
      pragma Assert (Is_Correct (X_Init, X));
      --  It is OK to use X_Init inside an assertion.
   end Do_Something;

end Show_Ghost;
```

`Do_Something` stores the initial value of X in a ghost constant, `X_Init`. We reference it in an assertion to check that the computation performed by the call to `Do_Some_Complex_Stuff` modified the value of X in the expected manner.

However, `X_Init` can't be used in normal code, for example to restore the initial value of X.

```
package Show_Ghost is

   type T is record
      A, B, C, D, E : Boolean;
   end record;

   function Formula (X : T) return Boolean is
     ((X.A and X.B) or (X.C and (X.D or X.E)));

   function Is_Correct (X, Y : T) return Boolean is
     (Formula (X) = Formula (Y));

   procedure Do_Something (X : in out T);

end Show_Ghost;
```

```
package body Show_Ghost is

   procedure Do_Some_Complex_Stuff (X : in out T) is
   begin
      X := T'(X.B, X.A, X.C, X.E, X.D);
   end Do_Some_Complex_Stuff;

   procedure Do_Something (X : in out T) is
      X_Init : constant T := X with Ghost;
   begin
      Do_Some_Complex_Stuff (X);
      pragma Assert (Is_Correct (X_Init, X));

      X := X_Init; -- ERROR

   end Do_Something;

end Show_Ghost;
```

```
with Show_Ghost; use Show_Ghost;

procedure Use_Ghost is
   X : T := (True, True, False, False, True);
begin
   Do_Something (X);
end Use_Ghost;
```

When compiling this example, the compiler flags the use of `X_Init` as illegal, but more complex cases of interference between ghost and normal code may sometimes only be detected when you run GNATprove.

5.2.2 Ghost Functions

Functions used only in specifications are a common occurrence when writing contracts for functional correctness. For example, expression functions used to simplify or factor out common patterns in contracts can usually be marked as ghost.

But ghost functions can do more than improve readability. In real-world programs, it's often the case that some information necessary for functional specification isn't accessible in the package's specification because of abstraction.

Making this information available to users of the packages is generally out of the question because that breaks the abstraction. Ghost functions come in handy in that case since they provide a way to give access to that information without making it available to normal client code.

Let's look at the following example.

```
package Stacks is

   pragma Unevaluated_Use_Of_Old (Allow);

   type Stack is private;

   type Element is new Natural;
   type Element_Array is array (Positive range <>) of Element;
   Max : constant Natural := 100;

   function Get_Model (S : Stack) return Element_Array with Ghost;
   --  Returns an array as a model of a stack.

   procedure Push (S : in out Stack; E : Element) with
     Pre  => Get_Model (S)'Length < Max,
     Post => Get_Model (S) = Get_Model (S)'Old & E;

private

   subtype Length_Type is Natural range 0 .. Max;

   type Stack is record
      Top     : Length_Type := 0;
      Content : Element_Array (1 .. Max) := (others => 0);
   end record;

end Stacks;
```

Here, the type `Stack` is private. To specify the expected behavior of the `Push` procedure, we need to go inside this abstraction and access the values of the elements stored in `S`. For this, we introduce a function `Get_Model` that returns an array as a representation of the stack. However, we don't want code that uses the `Stack` package to use `Get_Model` in normal code since this breaks our stack's abstraction.

Here's an example of trying to break that abstraction in the subprogram `Peek` below.

```
package Stacks is

   pragma Unevaluated_Use_Of_Old (Allow);

   type Stack is private;

   type Element is new Natural;
   type Element_Array is array (Positive range <>) of Element;
   Max : constant Natural := 100;

   function Get_Model (S : Stack) return Element_Array with Ghost;
```

(continues on next page)

(continued from previous page)

```
   --  Returns an array as a model of a stack.

   procedure Push (S : in out Stack; E : Element) with
     Pre  => Get_Model (S)'Length < Max,
     Post => Get_Model (S) = Get_Model (S)'Old & E;

   function Peek (S : Stack; I : Positive) return Element is
     (Get_Model (S) (I)); -- ERROR

private

   subtype Length_Type is Natural range 0 .. Max;

   type Stack is record
      Top     : Length_Type := 0;
      Content : Element_Array (1 .. Max) := (others => 0);
   end record;

end Stacks;
```

We see that marking the function as `Ghost` achieves this goal: it ensures that the subprogram `Get_Model` is never used in production code.

5.2.3 Global Ghost Variables

Though it happens less frequently, you may have specifications requiring you to store additional information in global variables that isn't needed in normal code. You should mark these global variables as ghost, allowing the compiler to remove them when assertions aren't enabled. You can use these variables for any purpose within the contracts that make up your specifications. A common scenario is writing specifications for subprograms that modify a complex or private global data structure: you can use these variables to provide a model for that structure that's updated by the ghost code as the program modifies the data structure itself.

You can also use ghost variables to store information about previous runs of subprograms to specify temporal properties. In the following example, we have two procedures, one that accesses a state A and the other that accesses a state B. We use the ghost variable `Last_Accessed_Is_A` to specify that B can't be accessed twice in a row without accessing A in between.

```
package Call_Sequence is

   type T is new Integer;

   Last_Accessed_Is_A : Boolean := False with Ghost;

   procedure Access_A with
     Post => Last_Accessed_Is_A;

   procedure Access_B with
     Pre  => Last_Accessed_Is_A,
     Post => not Last_Accessed_Is_A;
   --  B can only be accessed after A

end Call_Sequence;
```

```
package body Call_Sequence is

   procedure Access_A is
   begin
```

(continues on next page)

(continued from previous page)

```
      --  ...
      Last_Accessed_Is_A := True;
   end Access_A;

   procedure Access_B is
   begin
      --  ...
      Last_Accessed_Is_A := False;
   end Access_B;

end Call_Sequence;
```

```
with Call_Sequence; use Call_Sequence;

procedure Main is
begin
   Access_A;
   Access_B;
   Access_B; -- ERROR
end Main;
```

Let's look at another example. The specification of a subprogram's expected behavior is sometimes best expressed as a sequence of actions it must perform. You can use global ghost variables that store intermediate values of normal variables to write this sort of specification more easily.

For example, we specify the subprogram `Do_Two_Things` below in two steps, using the ghost variable `V_Interm` to store the intermediate value of `V` between those steps. We could also express this using an existential quantification on the variable `V_Interm`, but it would be impractical to iterate over all integers at runtime and this can't always be written in SPARK because quantification is restricted to `for ... loop` patterns.

Finally, supplying the value of the variable may help the prover verify the contracts.

```
package Action_Sequence is

   type T is new Integer;

   V_Interm : T with Ghost;

   function First_Thing_Done (X, Y : T) return Boolean with Ghost;
   function Second_Thing_Done (X, Y : T) return Boolean with Ghost;

   procedure Do_Two_Things (V : in out T) with
     Post => First_Thing_Done (V'Old, V_Interm)
       and then Second_Thing_Done (V_Interm, V);

end Action_Sequence;
```

5.3 Guide Proof

Since properties of interest for functional correctness are more complex than those involved in proofs of program integrity, we expect GNATprove to initially be unable to verify them even though they're valid. You'll find the techniques we discussed in *Debugging Failed Proof Attempts* (page 40) to come in handy here. We now go beyond those techniques and focus on more ways of improving results in the cases where the property is valid but GNATprove can't prove it in a reasonable amount of time.

In those cases, you may want to try and guide GNATprove to either complete the proof or strip it

down to a small number of easily-reviewable assumptions. For this purpose, you can add assertions to break complex proofs into smaller steps.

```
pragma Assert (Assertion_Checked_By_The_Tool);
--  info: assertion proved

pragma Assert (Assumption_Validated_By_Other_Means);
--  medium: assertion might fail

pragma Assume (Assumption_Validated_By_Other_Means);
--  The tool does not attempt to check this expression.
--  It is recorded as an assumption.
```

One such intermediate step you may find useful is to try to prove a theoretically-equivalent version of the desired property, but one where you've simplified things for the prover, such as by splitting up different cases or inlining the definitions of functions.

Some intermediate assertions may not be proved by GNATprove either because it's missing some information or because the amount of information available is confusing. You can verify these remaining assertions by other means such as testing (since they're executable) or by review. You can then choose to instruct GNATprove to ignore them, either by turning them into assumptions, as in our example, or by using a `pragma Annotate`. In both cases, the compiler generates code to check these assumptions at runtime when you enable assertions.

5.3.1 Local Ghost Variables

You can use ghost code to enhance what you can express inside intermediate assertions in the same way we did above to enhance our contracts in specifications. In particular, you'll commonly have local variables or constants whose only purpose is to be used in assertions. You'll mostly use these ghost variables to store previous values of variables or expressions you want to refer to in assertions. They're especially useful to refer to initial values of parameters and expressions since the `'Old` attribute is only allowed in postconditions.

In the example below, we want to help GNATprove verify the postcondition of `P`. We do this by introducing a local ghost constant, `X_Init`, to represent this value and writing an assertion in both branches of an `if` statement that repeats the postcondition, but using `X_Init`.

```
package Show_Local_Ghost is

   type T is new Natural;

   function F (X, Y : T) return Boolean is (X > Y) with Ghost;

   function Condition (X : T) return Boolean is (X mod 2 = 0);

   procedure P (X : in out T) with
     Pre  => X < 1_000_000,
     Post => F (X, X'Old);

end Show_Local_Ghost;
```

```
package body Show_Local_Ghost is

   procedure P (X : in out T) is
      X_Init : constant T := X with Ghost;
   begin
      if Condition (X) then
         X := X + 1;
         pragma Assert (F (X, X_Init));
      else
```

(continues on next page)

(continued from previous page)

```
         X := X * 2;
         pragma Assert (F (X, X_Init));
      end if;
   end P;

end Show_Local_Ghost;
```

You can also use local ghost variables for more complex purposes such as building a data structure that serves as witness for a complex property of a subprogram. In our example, we want to prove that the `Sort` procedure doesn't create new elements, that is, that all the elements present in `A` after the sort were in `A` before the sort. This property isn't enough to ensure that a call to `Sort` produces a value for `A` that's a permutation of its value before the call (or that the values are indeed sorted). However, it's already complex for a prover to verify because it involves a nesting of quantifiers. To help GNATprove, you may find it useful to store, for each index `I`, an index `J` that has the expected property.

```
procedure Sort (A : in out Nat_Array) with
  Post => (for all I in A'Range =>
             (for some J in A'Range => A (I) = A'Old (J)))
is
   Permutation : Index_Array := (1 => 1, 2 => 2, ...) with Ghost;
begin
   ...
end Sort;
```

5.3.2 Ghost Procedures

Ghost procedures can't affect the value of normal variables, so they're mostly used to perform operations on ghost variables or to group together a set of intermediate assertions.

Abstracting away the treatment of assertions and ghost variables inside a ghost procedure has several advantages. First, you're allowed to use these variables in any way you choose in code inside ghost procedures. This isn't the case outside ghost procedures, where the only ghost statements allowed are assignments to ghost variables and calls to ghost procedures.

As an example, the `for` loop contained in `Increase_A` couldn't appear by itself in normal code.

```
package Show_Ghost_Proc is

   type Nat_Array is array (Integer range <>) of Natural;

   A : Nat_Array (1 .. 100) with Ghost;

   procedure Increase_A with
     Ghost,
     Pre => (for all I in A'Range => A (I) < Natural'Last);

end Show_Ghost_Proc;
```

```
package body Show_Ghost_Proc is

   procedure Increase_A is
   begin
      for I in A'Range loop
         A (I) := A (I) + 1;
      end loop;
   end Increase_A;

end Show_Ghost_Proc;
```

Using the abstraction also improves readability by hiding complex code that isn't part of the functional behavior of the subprogram. Finally, it can help GNATprove by abstracting away assertions that would otherwise make its job more complex.

In the example below, calling `Prove_P` with `X` as an operand only adds `P (X)` to the proof context instead of the larger set of assertions required to verify it. In addition, the proof of `P` need only be done once and may be made easier not having any unnecessary information present in its context while verifying it. Also, if GNATprove can't fully verify `Prove_P`, you can review the remaining assumptions more easily since they're in a smaller context.

```
procedure Prove_P (X : T) with Ghost,
  Global => null,
  Post   => P (X);
```

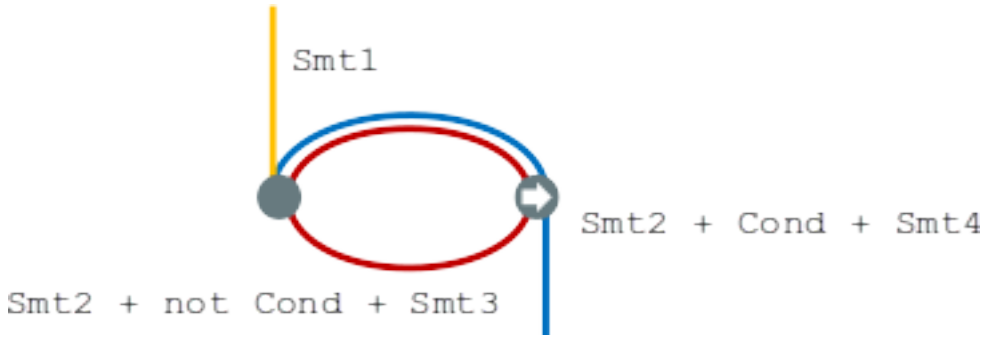
5.3.3 Handling of Loops

When the program involves a loop, you're almost always required to provide additional annotations to allow GNATprove to complete a proof because the verification techniques used by GNATprove doesn't handle cycles in a subprogram's control flow. Instead, loops are flattened by dividing them into several acyclic parts.

As an example, let's look at a simple loop with an exit condition.

```
Stmt1;
loop
  Stmt2;
  exit when Cond;
  Stmt3;
end loop;
Stmt4;
```

As shown below, the control flow is divided into three parts.



The first, shown in yellow, starts earlier in the subprogram and enters the loop statement. The loop itself is divided into two parts. Red represents a complete execution of the loop's body: an execution where the exit condition isn't satisfied. Blue represents the last execution of the loop, which includes some of the subprogram following it. For that path, the exit condition is assumed to hold. The red and blue parts are always executed after the yellow one.

GNATprove analyzes these parts independently since it doesn't have a way to track how variables may have been updated by an iteration of the loop. It forgets everything it knows about those variables from one part when entering another part. However, values of constants and variables that aren't modified in the loop are not an issue.

In other words, handling loops in that way makes GNATprove imprecise when verifying a subprogram involving a loop: it can't verify a property that relies on values of variables modified inside the loop. It won't forget any information it had on the value of constants or unmodified variables, but it nevertheless won't be able to deduce new information about them from the loop.

For example, consider the function `Find` which iterates over the array `A` and searches for an element where `E` is stored in `A`.

```
package Show_Find is

   type Nat_Array is array (Positive range <>) of Natural;

   function Find (A : Nat_Array; E : Natural) return Natural;

end Show_Find;
```

```
package body Show_Find is

   function Find (A : Nat_Array; E : Natural) return Natural is
   begin
      for I in A'Range loop
         pragma Assert (for all J in A'First .. I - 1 => A (J) /= E);
         --  assertion is not proved
         if A (I) = E then
            return I;
         end if;
         pragma Assert (A (I) /= E);
         --  assertion is proved
      end loop;
      return 0;
   end Find;

end Show_Find;
```

At the end of each loop iteration, GNATprove knows that the value stored at index `I` in `A` must not be `E`. (If it were, the loop wouldn't have reached the end of the interation.) This proves the second assertion. But it's unable to aggregate this information over multiple loop iterations to deduce that it's true for all the indexes smaller than `I`, so it can't prove the first assertion.

5.3.4 Loop Invariants

To overcome these limitations, you can provide additional information to GNATprove in the form of a *loop invariant*. In SPARK, a loop invariant is a Boolean expression which holds true at every iteration of the loop. Like other assertions, you can have it checked at runtime by compiling the program with assertions enabled.

The major difference between loop invariants and other assertions is the way it's treated for proofs. GNATprove performs the proof of a loop invariant in two steps: first, it checks that it holds for the first iteration of the loop and then it checks that it holds in an arbitrary iteration assuming it held in the previous iteration. This is called proof by induction[9].

As an example, let's add a loop invariant to the `Find` function stating that the first element of `A` is not `E`.

```
package Show_Find is

   type Nat_Array is array (Positive range <>) of Natural;

   function Find (A : Nat_Array; E : Natural) return Natural;

end Show_Find;
```

[9] https://en.wikipedia.org/wiki/Mathematical_induction

```
package body Show_Find is

   function Find (A : Nat_Array; E : Natural) return Natural is
   begin
      for I in A'Range loop
         pragma Loop_Invariant (A (A'First) /= E);
         --  loop invariant not proved in first iteration
         --  but preservation of loop invariant is proved
         if A (I) = E then
            return I;
         end if;
      end loop;
      return 0;
   end Find;

end Show_Find;
```

To verify this invariant, GNATprove generates two checks. The first checks that the assertion holds in the first iteration of the loop. This isn't verified by GNATprove. And indeed there's no reason to expect the first element of A to always be different from E in this iteration. However, the second check is proved: it's easy to deduce that if the first element of A was not E in a given iteration it's still not E in the next. However, if we move the invariant to the end of the loop, then it is successfully verified by GNATprove.

Not only do loop invariants allow you to verify complex properties of loops, but GNATprove also uses them to verify other properties, such as the absence of runtime errors over both the loop's body and the statements following the loop. More precisely, when verifying a runtime check or other assertion there, GNATprove assumes that the last occurrence of the loop invariant preceding the check or assertion is true.

Let's look at a version of `Find` where we use a loop invariant instead of an assertion to state that none of the array elements seen so far are equal to E.

```
package Show_Find is

   type Nat_Array is array (Positive range <>) of Natural;

   function Find (A : Nat_Array; E : Natural) return Natural;

end Show_Find;
```

```
package body Show_Find is

   function Find (A : Nat_Array; E : Natural) return Natural is
   begin
      for I in A'Range loop
         pragma Loop_Invariant
           (for all J in A'First .. I - 1 => A (J) /= E);
         if A (I) = E then
            return I;
         end if;
      end loop;
      pragma Assert (for all I in A'Range => A (I) /= E);
      return 0;
   end Find;

end Show_Find;
```

This version is fully verified by GNATprove! This time, it proves that the loop invariant holds in every iteration of the loop (separately proving this property for the first iteration and then for the following iterations). It also proves that none of the elements of A are equal to E after the loop exits by assuming that the loop invariant holds in the last iteration of the loop.

Finding a good loop invariant can turn out to be quite a challenge. To make this task easier, let's review the four good properties of a good loop invariant:

Property	Description
INIT	It should be provable in the first iteration of the loop.
INSIDE	It should allow proving the absence of run-time errors and local assertions inside the loop.
AFTER	It should allow proving absence of run-time errors, local assertions, and the subprogram postcondition after the loop.
PRESERVE	It should be provable after the first iteration of the loop.

Let's look at each of these in turn. First, the loop invariant should be provable in the first iteration of the loop (INIT). If your invariant fails to achieve this property, you can debug the loop invariant's initialization like any failing proof attempt using strategies for *Debugging Failed Proof Attempts* (page 40).

Second, the loop invariant should be precise enough to allow GNATprove to prove absence of run-time errors in both statements from the loop's body (INSIDE) and those following the loop (AFTER). To do this, you should remember that all information concerning a variable modified in the loop that's not included in the invariant is forgotten by GNATprove. In particular, you should take care to include in your invariant what's usually called the loop's *frame condition*, which lists properties of variables that are true throughout the execution of the loop even though those variables are modified by the loop.

Finally, the loop invariant should be precise enough to prove that it's preserved through successive iterations of the loop (PRESERVE). This is generally the trickiest part. To understand why GNATprove hasn't been able to verify the preservation of a loop invariant you provided, you may find it useful to repeat it as local assertions throughout the loop's body to determine at which point it can no longer be proved.

As an example, let's look at a loop that iterates through an array A and applies a function F to each of its elements.

```
package Show_Map is

   type Nat_Array is array (Positive range <>) of Natural;

   function F (V : Natural) return Natural is
     (if V /= Natural'Last then V + 1 else V);

   procedure Map (A : in out Nat_Array);

end Show_Map;
```

```
package body Show_Map is

   procedure Map (A : in out Nat_Array) is
      A_I : constant Nat_Array := A with Ghost;
   begin
      for K in A'Range loop
         A (K) := F (A (K));
         pragma Loop_Invariant
           (for all J in A'First .. K => A (J) = F (A'Loop_Entry (J)));
      end loop;
      pragma Assert (for all K in A'Range => A (K) = F (A_I (K)));
   end Map;

end Show_Map;
```

After the loop, each element of A should be the result of applying F to its previous value. We want to prove this. To specify this property, we copy the value of A before the loop into a ghost variable, A_I. Our loop invariant states that the element at each index less than K has been modified in the expected way. We use the Loop_Entry attribute to refer to the value of A on entry of the loop instead of using A_I.

Does our loop invariant have the four properties of a good loop-invariant? When launching GNATprove, we see that INIT is fulfilled: the invariant's initialization is proved. So are INSIDE and AFTER: no potential runtime errors are reported and the assertion following the loop is successfully verified.

The situation is slightly more complex for the PRESERVE property. GNATprove manages to prove that the invariant holds after the first iteration thanks to the automatic generation of frame conditions. It was able to do this because it completes the provided loop invariant with the following frame condition stating what part of the array hasn't been modified so far:

```
pragma Loop_Invariant
  (for all J in K .. A'Last => A (J) = (if J > K then A'Loop_Entry (J)));
```

GNATprove then uses both our and the internally-generated loop invariants to prove PRESERVE. However, in more complex cases, the heuristics used by GNATprove to generate the frame condition may not be sufficient and you'll have to provide one as a loop invariant. For example, consider a version of Map where the result of applying F to an element at index K is stored at index K-1:

```
package Show_Map is

   type Nat_Array is array (Positive range <>) of Natural;

   function F (V : Natural) return Natural is
     (if V /= Natural'Last then V + 1 else V);

   procedure Map (A : in out Nat_Array);

end Show_Map;
```

```
package body Show_Map is

   procedure Map (A : in out Nat_Array) is
      A_I : constant Nat_Array := A with Ghost;
   begin
      for K in A'Range loop
         if K /= A'First then
            A (K - 1) := F (A (K));
         end if;
         pragma Loop_Invariant
           (for all J in A'First .. K =>
             (if J /= A'First then A (J - 1) = F (A'Loop_Entry (J))));
         --  pragma Loop_Invariant
         --  (for all J in K .. A'Last => A (J) = A'Loop_Entry (J));
      end loop;
      pragma Assert (for all K in A'Range =>
                       (if K /= A'First then A (K - 1) = F (A_I (K))));
   end Map;

end Show_Map;
```

You need to uncomment the second loop invariant containing the frame condition in order to prove the assertion after the loop.

5.4 Code Examples / Pitfalls

This section contains some code examples and pitfalls.

5.4.1 Example #1

We implement a ring buffer inside an array `Content`, where the contents of a ring buffer of length `Length` are obtained by starting at index `First` and possibly wrapping around the end of the buffer. We use a ghost function `Get_Model` to return the contents of the ring buffer for use in contracts.

```
package Ring_Buffer is

   Max_Size : constant := 100;

   type Nat_Array is array (Positive range <>) of Natural;

   function Get_Model return Nat_Array with Ghost;

   procedure Push_Last (E : Natural) with
     Pre  => Get_Model'Length < Max_Size,
     Post => Get_Model'Length = Get_Model'Old'Length + 1;

end Ring_Buffer;
```

```
package body Ring_Buffer is

   subtype Length_Range is Natural range 0 .. Max_Size;
   subtype Index_Range  is Natural range 1 .. Max_Size;

   Content : Nat_Array (1 .. Max_Size) := (others => 0);
   First   : Index_Range               := 1;
   Length  : Length_Range              := 0;

   function Get_Model return Nat_Array with
     Refined_Post => Get_Model'Result'Length = Length
   is
      Size   : constant Length_Range := Length;
      Result : Nat_Array (1 .. Size) := (others => 0);
   begin
      if First + Length - 1 <= Max_Size then
         Result := Content (First .. First + Length - 1);
      else
         declare
            Len : constant Length_Range := Max_Size - First + 1;
         begin
            Result (1 .. Len) := Content (First .. Max_Size);
            Result (Len + 1 .. Length) := Content (1 .. Length - Len);
         end;
      end if;
      return Result;
   end Get_Model;

   procedure Push_Last (E : Natural) is
   begin
      if First + Length <= Max_Size then
         Content (First + Length) := E;
      else
         Content (Length - Max_Size + First) := E;
```

(continues on next page)

(continued from previous page)

```
      end if;
      Length := Length + 1;
   end Push_Last;

end Ring_Buffer;
```

This is correct: `Get_Model` is used only in contracts. Calls to `Get_Model` make copies of the buffer's contents, which isn't efficient, but is fine because `Get_Model` is only used for verification, not in production code. We enforce this by making it a ghost function. We'll produce the final production code with appropriate compiler switches (i.e., not using `-gnata`) that ensure assertions are ignored.

5.4.2 Example #2

Instead of using a ghost function, `Get_Model`, to retrieve the contents of the ring buffer, we're now using a global ghost variable, `Model`.

```
package Ring_Buffer is

   Max_Size : constant := 100;
   subtype Length_Range is Natural range 0 .. Max_Size;
   subtype Index_Range  is Natural range 1 .. Max_Size;

   type Nat_Array is array (Positive range <>) of Natural;

   type Model_Type (Length : Length_Range := 0) is record
      Content : Nat_Array (1 .. Length);
   end record
     with Ghost;

   Model : Model_Type with Ghost;

   function Valid_Model return Boolean;

   procedure Push_Last (E : Natural) with
     Pre  => Valid_Model
       and then Model.Length < Max_Size,
     Post => Model.Length = Model.Length'Old + 1;

end Ring_Buffer;
```

```
package body Ring_Buffer is

   Content : Nat_Array (1 .. Max_Size) := (others => 0);
   First   : Index_Range               := 1;
   Length  : Length_Range              := 0;

   function Valid_Model return Boolean is
      (Model.Content'Length = Length);

   procedure Push_Last (E : Natural) is
   begin
      if First + Length <= Max_Size then
         Content (First + Length) := E;
      else
         Content (Length - Max_Size + First) := E;
      end if;
      Length := Length + 1;
```

(continues on next page)

(continued from previous page)

```
   end Push_Last;

end Ring_Buffer;
```

This example isn't correct. `Model`, which is a ghost variable, must not influence the return value of the normal function `Valid_Model`. Since `Valid_Model` is only used in specifications, we should have marked it as `Ghost`. Another problem is that `Model` needs to be updated inside `Push_Last` to reflect the changes to the ring buffer.

5.4.3 Example #3

Let's mark `Valid_Model` as `Ghost` and update `Model` inside `Push_Last`.

```
package Ring_Buffer is

   Max_Size : constant := 100;
   subtype Length_Range is Natural range 0 .. Max_Size;
   subtype Index_Range  is Natural range 1 .. Max_Size;

   type Nat_Array is array (Positive range <>) of Natural;

   type Model_Type (Length : Length_Range := 0) is record
      Content : Nat_Array (1 .. Length);
   end record
     with Ghost;

   Model : Model_Type with Ghost;

   function Valid_Model return Boolean with Ghost;

   procedure Push_Last (E : Natural) with
     Pre  => Valid_Model
       and then Model.Length < Max_Size,
     Post => Model.Length = Model.Length'Old + 1;

end Ring_Buffer;
```

```
package body Ring_Buffer is

   Content : Nat_Array (1 .. Max_Size) := (others => 0);
   First   : Index_Range               := 1;
   Length  : Length_Range              := 0;

   function Valid_Model return Boolean is
      (Model.Content'Length = Length);

   procedure Push_Last (E : Natural) is
   begin
      if First + Length <= Max_Size then
         Content (First + Length) := E;
      else
         Content (Length - Max_Size + First) := E;
      end if;
      Length := Length + 1;
      Model := (Length  => Model.Length + 1,
                Content => Model.Content & E);
   end Push_Last;

end Ring_Buffer;
```

This example is correct. The ghost variable `Model` can be referenced both from the body of the ghost function `Valid_Model` and the non-ghost procedure `Push_Last` as long as it's only used in ghost statements.

5.4.4 Example #4

We're now modifying `Push_Last` to share the computation of the new length between the operational and ghost code.

```
package Ring_Buffer is

   Max_Size : constant := 100;
   subtype Length_Range is Natural range 0 .. Max_Size;
   subtype Index_Range  is Natural range 1 .. Max_Size;

   type Nat_Array is array (Positive range <>) of Natural;

   type Model_Type (Length : Length_Range := 0) is record
      Content : Nat_Array (1 .. Length);
   end record
     with Ghost;

   Model : Model_Type with Ghost;

   function Valid_Model return Boolean with Ghost;

   procedure Push_Last (E : Natural) with
     Pre  => Valid_Model
       and then Model.Length < Max_Size,
     Post => Model.Length = Model.Length'Old + 1;

end Ring_Buffer;
```

```
package body Ring_Buffer is

   Content : Nat_Array (1 .. Max_Size) := (others => 0);
   First   : Index_Range               := 1;
   Length  : Length_Range              := 0;

   function Valid_Model return Boolean is
      (Model.Content'Length = Length);

   procedure Push_Last (E : Natural) is
      New_Length : constant Length_Range := Model.Length + 1;
   begin
      if First + Length <= Max_Size then
         Content (First + Length) := E;
      else
         Content (Length - Max_Size + First) := E;
      end if;
      Length := New_Length;
      Model := (Length  => New_Length,
                Content => Model.Content & E);
   end Push_Last;

end Ring_Buffer;
```

This example isn't correct. We didn't mark local constant `New_Length` as `Ghost`, so it can't be computed from the value of ghost variable `Model`. If we made `New_Length` a ghost constant, the compiler would report the problem on the assignment from `New_Length` to `Length`. The correct solution here is to compute `New_Length` from the value of the non-ghost variable `Length`.

5.4.5 Example #5

Let's move the code updating `Model` inside a local ghost procedure, `Update_Model`, but still using a local variable, `New_Length`, to compute the length.

```
package Ring_Buffer is

   Max_Size : constant := 100;
   subtype Length_Range is Natural range 0 .. Max_Size;
   subtype Index_Range  is Natural range 1 .. Max_Size;

   type Nat_Array is array (Positive range <>) of Natural;

   type Model_Type (Length : Length_Range := 0) is record
      Content : Nat_Array (1 .. Length);
   end record
     with Ghost;

   Model : Model_Type with Ghost;

   function Valid_Model return Boolean with Ghost;

   procedure Push_Last (E : Natural) with
     Pre  => Valid_Model
       and then Model.Length < Max_Size,
     Post => Model.Length = Model.Length'Old + 1;

end Ring_Buffer;
```

```
package body Ring_Buffer is

   Content : Nat_Array (1 .. Max_Size) := (others => 0);
   First   : Index_Range               := 1;
   Length  : Length_Range              := 0;

   function Valid_Model return Boolean is
      (Model.Content'Length = Length);

   procedure Push_Last (E : Natural) is

      procedure Update_Model with Ghost is
         New_Length : constant Length_Range := Model.Length + 1;
      begin
         Model := (Length  => New_Length,
                   Content => Model.Content & E);
      end Update_Model;

   begin
      if First + Length <= Max_Size then
         Content (First + Length) := E;
      else
         Content (Length - Max_Size + First) := E;
      end if;
      Length := Length + 1;
      Update_Model;
   end Push_Last;

end Ring_Buffer;
```

Everything's fine here. `Model` is only accessed inside `Update_Model`, itself a ghost procedure, so it's fine to declare local variable `New_Length` without the `Ghost` aspect: everything inside a ghost procedure body is ghost. Moreover, we don't need to add any contract to `Update_Model`: it's

inlined by GNATprove because it's a local procedure without a contract.

5.4.6 Example #6

The function `Max_Array` takes two arrays of the same length (but not necessarily with the same bounds) as arguments and returns an array with each entry being the maximum values of both arguments at that index.

```
package Array_Util is

   type Nat_Array is array (Positive range <>) of Natural;

   function Max_Array (A, B : Nat_Array) return Nat_Array with
     Pre => A'Length = B'Length;

end Array_Util;
```

```
package body Array_Util is

   function Max_Array (A, B : Nat_Array) return Nat_Array is
      R : Nat_Array (A'Range);
      J : Integer := B'First;
   begin
      for I in A'Range loop
         if A (I) > B (J) then
            R (I) := A (I);
         else
            R (I) := B (J);
         end if;
         J := J + 1;
      end loop;
      return R;
   end Max_Array;

end Array_Util;
```

This program is correct, but GNATprove can't prove that J is always in the index range of B (the unproved index check) or even that it's always within the bounds of its type (the unproved overflow check). Indeed, when checking the body of the loop, GNATprove forgets everything about the current value of J because it's been modified by previous loop iterations. To get more precise results, we need to provide a loop invariant.

5.4.7 Example #7

Let's add a loop invariant that states that J stays in the index range of B and let's protect the increment to J by checking that it's not already the maximal integer value.

```
package Array_Util is

   type Nat_Array is array (Positive range <>) of Natural;

   function Max_Array (A, B : Nat_Array) return Nat_Array with
     Pre => A'Length = B'Length;

end Array_Util;
```

```
package body Array_Util is

   function Max_Array (A, B : Nat_Array) return Nat_Array is
      R : Nat_Array (A'Range);
      J : Integer := B'First;
   begin
      for I in A'Range loop
         pragma Loop_Invariant (J in B'Range);
         if A (I) > B (J) then
            R (I) := A (I);
         else
            R (I) := B (J);
         end if;
         if J < Integer'Last then
            J := J + 1;
         end if;
      end loop;
      return R;
   end Max_Array;

end Array_Util;
```

The loop invariant now allows verifying that no runtime error can occur in the loop's body (property INSIDE seen in section *Loop Invariants* (page 84)). Unfortunately, GNATprove fails to verify that the invariant stays valid after the first iteration of the loop (property PRESERVE). Indeed, knowing that `J` is in `B'Range` in a given iteration isn't enough to prove it'll remain so in the next iteration. We need a more precise invariant, linking `J` to the value of the loop index `I`, like `J = I - A'First + B'First`.

5.4.8 Example #8

We now consider a version of `Max_Array` which takes arguments that have the same bounds. We want to prove that `Max_Array` returns an array of the maximum values of both its arguments at each index.

```
package Array_Util is

   type Nat_Array is array (Positive range <>) of Natural;

   function Max_Array (A, B : Nat_Array) return Nat_Array with
     Pre  => A'First = B'First and A'Last = B'Last,
     Post => (for all K in A'Range =>
                Max_Array'Result (K) = Natural'Max (A (K), B (K)));

end Array_Util;
```

```
package body Array_Util is

   function Max_Array (A, B : Nat_Array) return Nat_Array is
      R : Nat_Array (A'Range) := (others => 0);
   begin
      for I in A'Range loop
         pragma Loop_Invariant (for all K in A'First .. I =>
                                  R (K) = Natural'Max (A (K), B (K)));
         if A (I) > B (I) then
            R (I) := A (I);
         else
            R (I) := B (I);
         end if;
```

(continues on next page)

(continued from previous page)

```
      end loop;
      return R;
   end Max_Array;

end Array_Util;
```

```
with Array_Util; use Array_Util;

procedure Main is
   A : Nat_Array := (1, 1, 2);
   B : Nat_Array := (2, 1, 0);
   R : Nat_Array (1 .. 3);
begin
   R := Max_Array (A, B);
end Main;
```

Here, GNATprove doesn't manage to prove the loop invariant even for the first loop iteration (property INIT seen in section *Loop Invariants* (page 84)). In fact, the loop invariant is incorrect, as you can see by executing the function `Max_Array` with assertions enabled: at each loop iteration, `R` contains the maximum of `A` and `B` only until `I - 1` because the `I`'th index wasn't yet handled.

5.4.9 Example #9

We now consider a procedural version of `Max_Array` which updates its first argument instead of returning a new array. We want to prove that `Max_Array` sets the maximum values of both its arguments into each index in its first argument.

```
package Array_Util is

   type Nat_Array is array (Positive range <>) of Natural;

   procedure Max_Array (A : in out Nat_Array; B : Nat_Array) with
     Pre  => A'First = B'First and A'Last = B'Last,
     Post => (for all K in A'Range =>
                A (K) = Natural'Max (A'Old (K), B (K)));

end Array_Util;
```

```
package body Array_Util is

   procedure Max_Array (A : in out Nat_Array; B : Nat_Array) is
   begin
      for I in A'Range loop
         pragma Loop_Invariant
           (for all K in A'First .. I - 1 =>
              A (K) = Natural'Max (A'Loop_Entry (K), B (K)));
         pragma Loop_Invariant
           (for all K in I .. A'Last => A (K) = A'Loop_Entry (K));
         if A (I) <= B (I) then
            A (I) := B (I);
         end if;
      end loop;
   end Max_Array;

end Array_Util;
```

Everything is proved. The first loop invariant states that the values of `A` before the loop index contains the maximum values of the arguments of `Max_Array` (referring to the input value of `A`

with `A'Loop_Entry`). The second loop invariant states that the values of A beyond and including the loop index are the same as they were on entry. This is the frame condition of the loop.

5.4.10 Example #10

Let's remove the frame condition from the previous example.

```
package Array_Util is

   type Nat_Array is array (Positive range <>) of Natural;

   procedure Max_Array (A : in out Nat_Array; B : Nat_Array) with
     Pre  => A'First = B'First and A'Last = B'Last,
     Post => (for all K in A'Range =>
                A (K) = Natural'Max (A'Old (K), B (K)));

end Array_Util;
```

```
package body Array_Util is

   procedure Max_Array (A : in out Nat_Array; B : Nat_Array) is
   begin
      for I in A'Range loop
         pragma Loop_Invariant
           (for all K in A'First .. I - 1 =>
              A (K) = Natural'Max (A'Loop_Entry (K), B (K)));
         if A (I) <= B (I) then
            A (I) := B (I);
         end if;
      end loop;
   end Max_Array;

end Array_Util;
```

Everything is still proved. GNATprove internally generates the frame condition for the loop, so it's sufficient here to state that A before the loop index contains the maximum values of the arguments of `Max_Array`.