SPARK User's Guide

Release 14.0.0

AdaCore and Capgemini Engineering

Mar 28, 2025

CONTENTS

CHAPTER

ONE

GETTING STARTED WITH SPARK

We begin with a very simple guide aimed at getting new users up and running with the SPARK tools. A small SPARK example program will be used for illustration.

Note: The online version of this User's Guide applies to the latest development version of the SPARK toolset. If you're using an official release, some of the described features may not apply. Refer to the version of the SPARK User's Guide shipping with your release, available through *Help* → *SPARK* in GNAT Studio and GNATbench IDEs, or under `share/doc/spark` in your SPARK installation.

As a prerequisite, it is assumed that the SPARK tools have already been installed. As a minimum you should install:

- SPARK Pro, SPARK Discovery or SPARK Community
- GNAT Studio or the GNATbench plug-in of Eclipse

SPARK Pro is the most complete toolset for SPARK. SPARK Discovery is a reduced toolset that still allows to perform all analyses presented in this User's Guide, but is less powerful than SPARK Pro. Compared to SPARK Pro, SPARK Discovery:

- only comes with one automatic prover instead of three
- does not generate counterexamples for failed proofs
- has limited proof support for programs using modular arithmetic or floating-point arithmetic
- comes without a lemma library for more difficult proofs

SPARK Community is a version packaged for free software developers, hobbyists, and students, which retains most of the capabilities of SPARK Pro.

Note that GNAT Studio is not strictly required for SPARK as all the commands can be invoked from the command line, or from Eclipse using the GNATbench plug-in, but the instructions in this section assume that GNAT Studio is being used. If you are a supported user, you can get more information on how to install the tools in "AdaCore Installation Procedures" under the "Download" tab in GNAT Tracker, or by contacting AdaCore for further advice.

The key tools that we will use in this example are GNATprove and GNAT Studio. To begin with, launch GNAT Studio with a new default project and check that the *SPARK* menu is present in the menu bar.

Note: For SPARK 2005 users, this menu will appear under the name *SPARK 2014*, to avoid any confusion with the existing *SPARK* menu for SPARK 2005 toolset.

Now open a new file in GNAT Studio and type the following short program into it. Save this file as `diff.adb`.

```
procedure Diff (X, Y : in Natural; Z : out Natural) with
  SPARK_Mode,
  Depends => (Z => (X, Y))
is
begin
   Z := X - X;
end Diff;
```

The program is intended to calculate the difference between `X` and `Y` and store the result in `Z`. This is reflected in the aspect `Depends` which states that the output value of Z depends on the input values of X and Y, but, as you may have noticed, there is a bug in the code. Note the use of aspect `SPARK_Mode` to identify this as SPARK code to be analysed with the SPARK tools. To analyze this program, select *SPARK → Examine File* from the menu in GNAT Studio. GNATprove executes in flow analysis mode and reports:

```
diff.adb:1:20: warning: unused variable "Y"
    1 |procedure Diff (X, Y : in Natural; Z : out Natural) with
      |                   ^ here

diff.adb:3:03: medium: missing dependency "null => Y"
    3 |  Depends => (Z => (X, Y))
      |  ^~~~~~~~~~~~~~~~~~~~~~

diff.adb:3:24: medium: incorrect dependency "Z => Y"
    3 |  Depends => (Z => (X, Y))
      |                        ^ here
```

These messages are informing us that there is a discrepancy between the program's contract (which says that the value of `Z` is obtained from the values of `X` and `Y`) and its implementation (in which the value of `Z` is derived only from the value of `X`, and `Y` is unused). In this case the contract is correct and the code is wrong, so fix the code by changing the assignment statement to `Z := X - Y;` and re-run the analysis. This time it should report no messages.

Having established that the program is free from flow errors, now let's run the tools in proof mode to check for run-time errors. Select *SPARK → Prove File* from the menu in GNAT Studio, and click on `Execute` in the resulting dialog box. GNATprove now attempts to show, using formal verification, that the program is free from run-time errors. But it finds a problem and highlights the assignment statement in red, reporting:

```
diff.adb:6:11: high: range check might fail, cannot prove lower bound for X - Y
    6 |   Z := X - Y;
      |        ~~^~~
  e.g. when X = 0
        and Y = 1
  reason for check: result of subtraction must fit in the target type of the assignment
  possible fix: add precondition (X >= Natural'First + Y) to subprogram at line 1
    1 |procedure Diff (X, Y : in Natural; Z : out Natural) with
      |          ^ here
```

This means that the tools are unable to show that the result of subtracting one `Natural` number from another will be within the range of the type `Natural`, which is hopefully not too surprising! There are various ways in which this could be addressed depending on what the requirements are for this subprogram, but for now let's change the type of parameter `Z` from `Natural` to `Integer`. If the analysis is re-run with this change in place then GNATprove will report no issues. All checks are proved so we can be confident that no exceptions will be raised by the execution of this code.

This short example was intended to give a flavor of the types of analysis that can be performed with the SPARK tools.

A more in-depth example is presented later in *SPARK Tutorial*.

CHAPTER

TWO

INTRODUCTION

SPARK is a programming language and a set of verification tools designed to meet the needs of high-assurance software development. SPARK is based on Ada, both subsetting the language to remove features that defy verification, but also extending the system of contracts and aspects to support modular, formal verification.

The new aspects support abstraction and refinement and facilitate deep static analysis to be performed including flow analysis and formal verification of an implementation against a specification.

The current version of SPARK, sometimes referred to as SPARK 2014, is a much larger and more flexible language than its predecessor SPARK 2005. The language can be configured to suit a number of application domains and standards, from server-class high-assurance systems (such as air-traffic management applications), to embedded, hard real-time, critical systems (such as avionic systems complying with DO-178C Level A).

A major feature of SPARK is the support for a mixture of proof and other verification methods such as testing, which facilitates in particular the use of unit proof in place of unit testing; an approach now formalized in DO-178C and the DO-333 formal methods supplement. Certain units may be formally proven and other units validated through testing.

SPARK is supported by various tools in the GNAT toolsuite:

- the GNAT compiler
- the GNAT Studio integrated development environment
- the GNATtest tool for unit testing harness generation
- the GNATprove tool for formal program verification

The remainder of this document is structured as follows:

- *Installation of GNATprove* goes through the installation steps on different platforms.
- *Identifying SPARK Code* describes the various means to identify the part of the program in SPARK that should be analyzed.
- *Overview of SPARK Language* provides an overview of the SPARK language.
- *SPARK Tutorial* gives an introduction to writing, testing and proving SPARK programs.
- *Formal Verification with GNATprove* describes the use of the GNATprove formal verification tool.
- *Applying SPARK in Practice* lists the main objectives and project scenarios for using SPARK.

CHAPTER

THREE

INSTALLATION OF GNATPROVE

In general, you will need to install a recent version of GNAT toolchain to compile SPARK programs. You will need to install one toolchain for each platform that you target, for example one toolchain for native compilation on your machine and one toolchain for cross compilation to an embedded platform.

For analyzing SPARK programs, we recommend to first install GNAT Studio and then install GNATprove under the same location. Alternatively, you can install the GNATbench plug-in for Eclipse instead of GNAT Studio, using the Eclipse installation mechanism. The same version of GNAT Studio or GNATbench can support both native and cross compilations, as well as SPARK analysis.

If you choose to install GNATprove in a different location, you should also modify the environment variables `GPR_PROJECT_PATH` (if you installed GNAT). On Windows, edit the value of `GPR_PROJECT_PATH` under the Environment Variables panel, and add to it the value of `<GNAT install dir>/lib/gnat` and `<GNAT install dir>/share/gpr` (so that SPARK can find library projects installed with GNAT) and `<SPARK install dir>/lib/gnat` (so that GNAT can find the SPARK lemma library project installed with SPARK, for details see *Manual Proof Using SPARK Lemma Library*). On Linux/Mac with Bourne shell, use:

```
export GPR_PROJECT_PATH=<GNAT install dir>/lib/gnat:<GNAT install dir>/share/gpr:<SPARK␣
↪install dir>/lib/gnat:$GPR_PROJECT_PATH
```

or on Linux/Mac with C shell:

```
setenv GPR_PROJECT_PATH <GNAT install dir>/lib/gnat:<GNAT install dir>/share/gpr:<SPARK␣
↪install dir>/lib/gnat:$GPR_PROJECT_PATH
```

See below for detailed installation instructions of GNAT Studio and GNATprove.

3.1 System Requirements

Formal verification is complex and time consuming, so GNATprove will benefit from all the speed (CPU) and memory (RAM) that can be made available. A minimum of 2 GB of RAM per core is recommended. More complex analyses will require more memory. A recommended configuration for running GNATprove on large systems is an x86-64 machine running Linux 64bits or Windows 64bits with at least 8 cores and 16 GB of RAM. Slower machines can be used to analyze small subsystems, but a minimum of 2.8Ghz CPU and 2 GB of RAM is required.

3.2 Installation under Windows

If not already done, first run the GNAT Studio installer by e.g. double clicking on *gnatstudio-<version>-i686-pc-mingw32.exe* and follow the instructions.

Note: If you're using GNAT Community instead of GNAT Pro, you should run instead the GNAT Community installer, which installs GNAT Studio and SPARK.

Then similarly run the GNATprove installer, by e.g. double clicking on *spark-<version>-x86-windows-bin.exe*. If you intend to install GNATprove in a location where a previous installation of GNATprove exists, we recommend that you uninstall the previous installation first.

You should have sufficient rights for installing the package (administrator or normal rights depending on whether it is installed for all users or a single user).

3.3 Installation under Linux/Mac

If not already done, you need to extract and install the GNAT Studio compressed tarball and then run the install, e.g.:

```
$ gzip -dc gnatstudio-<version>-<platform>-bin.tar.gz | tar xf -
$ cd gnatstudio-<version>-<platform>-bin
$ ./doinstall
```

Then follow the instructions displayed.

Note: If you're using GNAT Community instead of GNAT Pro, you should install instead the GNAT Community package, which installs GNAT Studio and SPARK.

Then do the same with the SPARK tarball, e.g.:

```
$ gzip -dc spark-<version>-<platform>-bin.tar.gz | tar xf -
$ cd spark-<version>-<platform>-bin
$ ./doinstall
```

Note that you need to have sufficient rights for installing the package at the chosen location (e.g. root rights for installing under /opt/spark).

3.4 Installation of the FSF version of SPARK

A so-called FSF version of SPARK is freely available.

3.4.1 Manual install

You can download a "gnatprove" package from this github project. Extracting the package and adding the `bin` directory to your PATH is enough. You can get the GNAT compiler from the same link, and there is a different project for GNATStudio, the IDE.

3.4.2 Install using alire

You can obtain SPARK via Alire. To do this, follow the installation instructions of Alire, then you can add the `gnatprove` dependency to an alire project using:

```
alr with gnatprove
```

Alire will download gnatprove if necessary.

3.4.3 The older GNAT Community version

There is an older community version of the tools, packaged with GNAT and GNATStudio. You can download it from AdaCore's website.

CHAPTER

FOUR

IDENTIFYING SPARK CODE

In general a program can have some parts that are in SPARK (and follow all the rules in the SPARK Reference Manual), and some parts that are full Ada. Pragma or aspect `SPARK_Mode` is used to identify which parts are in SPARK (by default programs are in full Ada).

This section contains a simple description of pragma and aspect `SPARK_Mode`. See *Aspect and Pragma SPARK_Mode* for the complete description.

Note that GNATprove only analyzes parts of the code that are identified as being in SPARK using pragma or aspect `SPARK_Mode`.

4.1 Mixing SPARK Code and Ada Code

An Ada program unit or other construct is said to be "in SPARK" if it complies with the restrictions required to permit formal verification given in the SPARK Reference Manual. Conversely, an Ada program unit or other construct is "not in SPARK" if it does not meet these requirements, and so is not amenable to formal verification.

Within a single Ada unit, constructs which are "in" and "not in" SPARK may be mixed at a fine level in accordance with the following two general principles:

- SPARK code shall only reference SPARK declarations, but a SPARK declaration which requires a completion may have a non-SPARK completion.
- SPARK code may enclose non-SPARK code.
- non-SPARK code may enclose SPARK code only at library level. A subprogram body which is not in SPARK cannot contain SPARK code.

More specifically, non-SPARK completions of SPARK declarations are allowed for subprogram declarations, package declarations, task type declarations, protected type declarations, private type declarations, private extension declarations, and deferred constant declarations. [Strictly speaking, the private part of a package, a task type or a protected type is considered to be part of its completion for purposes of the above rules; this is described in more detail below].

When a non-SPARK completion is provided for a SPARK declaration, the user has an obligation to ensure that the non-SPARK completion is consistent (with respect to the semantics of SPARK) with its SPARK declaration. For example, SPARK requires that a function call has no side effects. If the body of a given function is in SPARK, then this rule is enforced via various language rules; otherwise, it is the responsibility of the user to ensure that the function body does not violate this rule. As with other such constructs (notably pragma Assume), failure to meet this obligation can invalidate any or all analysis (proofs and/or flow analysis) associated with the SPARK portion of a program. A non-SPARK completion meets this obligation if it is semantically equivalent (with respect to dynamic semantics) to some notional completion that could have been written in SPARK.

When a non-SPARK package declaration or body is included in a SPARK subprogram or package, the user has an obligation to ensure that the non-SPARK declaration is consistent (with respect to the semantics of SPARK) with

a hypothetical equivalent SPARK declaration. For example, SPARK requires that package elaboration code cannot modify variables defined outside of the package.

The SPARK semantics (specifically including flow analysis and proof) of a "mixed" program which meets the aforementioned requirement are well defined - they are the semantics of the equivalent 100% SPARK program. For the semantics of other "mixed" programs refer to the Ada Reference Manual.

In the case of a package, a task type, or a protected type, the specification/completion division described above is a simplification of the true situation. For instance, a package is divided into 4 sections, not just 2: its visible part, its private part, the declarations of its body, and the statement list of its body. For a given package and any number N in the range 0 .. 4, the first N sections of the package might be in SPARK while the remainder is not.

For example, the following combinations may be typical:

- Package specification in SPARK. Package body not in SPARK.
- Visible part of package specification in SPARK. Private part and body not in SPARK.
- Package specification in SPARK. Package body almost entirely in SPARK, with a small number of subprogram bodies not in SPARK.
- Package specification in SPARK, with all subprogram bodies imported from another language.
- Package specification contains a mixture of declarations which are in SPARK and not in SPARK. The latter declarations are only visible and usable from client units which are not in SPARK.

Task types and protected types are similar to packages but only have 3 sections instead of 4. The statement list section of the body is missing.

Another typical use is to exempt part of a subprogram from analysis by isolating it in a local subprogram whose body is not in SPARK.

Such patterns are intended to allow for application of formal verification to a subset of a program, and the combination of formal verification with more traditional testing (see *Applying SPARK in Practice*).

4.2 Project File Setup

The project file is used to identify coarsely which parts of a program are in SPARK. To get more details on project file setup, see section *Setting Up a Project File*.

4.2.1 Setting the Default SPARK_Mode

There are two possible defaults:

1. No value of `SPARK_Mode` is specified as a configuration pragma. In that case, only the parts of the program explicitly marked with `SPARK_Mode => On` are in SPARK. This default is recommended if only a small number of units or subprograms are in SPARK.
2. A value of `SPARK_Mode => On` is specified as a configuration pragma. In that case, all the program should be in SPARK, except for those parts explicitly marked with `SPARK_Mode => Off` or a configuration pragma of `Auto` inside files. This mode is recommended if most of the program is in SPARK.

Here is how to specify a value of `SPARK_Mode => On` as a configuration pragma:

```
project My_Project is
   package Builder is
      for Global_Configuration_Pragmas use "spark.adc";
```

(continues on next page)

(continued from previous page)

```
   end Builder;
end My_Project;
```

where `spark.adc` is a configuration file containing at least the following line:

```
pragma SPARK_Mode (On);
```

4.2.2 Specifying Files To Analyze

By default, all files from a project are analyzed by GNATprove. It may be useful to restrict the set of files to analyze to speedup analysis if only a subset of the files contain SPARK code.

The set of files to analyze can be identified by specifying a different value of various project attributes in the mode used for formal verification:

- `Source_Dirs`: list of source directory names
- `Source_Files`: list of source file names
- `Source_List_File`: name of a file listing source file names

For example:

```
project My_Project is

  type Modes is ("Compile", "Analyze");
  Mode : Modes := External ("MODE", "Compile");

  case Mode is
     when "Compile" =>
        for Source_Dirs use (...);
     when "Analyze" =>
        for Source_Dirs use ("dir1", "dir2");
        for Source_Files use ("file1.ads", "file2.ads", "file1.adb", "file2.adb");
  end case;

end My_Project;
```

Then, GNATprove should be called by specifying the value of the `MODE` external variable as follows:

```
gnatprove -P my_project -XMODE=Analyze
```

4.2.3 Excluding Files From Analysis

When choosing a default value of `SPARK_Mode => On`, it may be needed to exclude some files from analysis (for example, because they contain non-SPARK code, or code that does not need to be formally analyzed).

The set of files to exclude can be identified by specifying a different value of various project attributes in the mode used for formal verification:

- `Excluded_Source_Dirs`: list of excluded source directory names
- `Excluded_Source_Files`: list of excluded source file names
- `Excluded_Source_List_File`: name of a file listing excluded source file names

For example:

```
project My_Project is
   package Builder is
      for Global_Configuration_Pragmas use "spark.adc";
   end Builder;

  type Modes is ("Compile", "Analyze");
  Mode : Modes := External ("MODE", "Compile");

  case Mode is
     when "Compile" =>
        null;
     when "Analyze" =>
        for Excluded_Source_Files use ("file1.ads", "file1.adb", "file2.adb");
  end case;

end My_Project;
```

Then, GNATprove should be called by specifying the value of the `MODE` external variable as follows:

```
gnatprove -P my_project -XMODE=Analyze
```

4.2.4 Using Multiple Projects

Sometimes, it is more convenient to analyze a subset of the source files with the default `SPARK_Mode => On` and the rest of the source files with no setting for `SPARK_Mode`. In that case, one can use two project files with different defaults, with each source file in one of the projects only. Files in one project can still refer to files in the other project by using a `limited with` clause between projects, as follows:

```
limited with "project_b"
project My_Project_A is
   package Compiler is
      for Local_Configuration_Pragmas use "spark.adc";
   end Compiler;
   for Source_Files use ("file1.ads", "file2.ads", "file1.adb", "file2.adb");
end My_Project_A;
```

```
limited with "project_a"
project My_Project_B is
   for Source_Files use ("file3.ads", "file4.ads", "file3.adb", "file4.adb");
end My_Project_B;
```

where `spark.adc` is a configuration file containing at least the following line:

```
pragma SPARK_Mode (On);
```

4.3 Using SPARK_Mode in Code

The pragma or aspect SPARK_Mode can be used in the code to identify precisely which parts of a program are in SPARK.

4.3.1 Basic Usage

The form of a pragma SPARK_Mode is as follows:

```
pragma SPARK_Mode [ (Auto | On | Off) ]
```

For example:

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

The value Auto is only allowed in configuration pragmas, either in a configuration pragma file, or inside a source file. Thus, value Auto is not allowed in aspect SPARK_Mode. Having a value Auto means that the file is analyzed as if no value of SPARK_Mode was specified, which is useful in cases where SPARK_Mode => On is specified in a configuration pragma file for the complete project, but a file contains both entities compatible with SPARK and entities not in SPARK.

The form of an aspect SPARK_Mode is as follows:

```
with SPARK_Mode => [ On | Off ]
```

For example:

```
package P with
  SPARK_Mode => On
is
```

A default argument of On is assumed for any SPARK_Mode pragma or aspect for which no argument is explicitly specified.

For example:

```
package P is
   pragma SPARK_Mode;  --  On is implicit here
```

or

```
package P with
  SPARK_Mode  --  On is implicit here
is
```

We say that a package or a subprogram is library-level if it is either top-level (i.e. it is a library unit; its declaration is the outermost program unit declared in a given compilation unit) or declared immediately within another library-level package (which excludes, for example, declarations inside subprogram bodies). For example, all the packages in the following code snippet are library-level packages:

```
package P is
   package Q is
      package R is
...
package body P is
   package S is
      package T is
```

The SPARK_Mode pragma can be used in the following places in the code:

- as a configuration pragma at unit level (even before with-clauses) in particular for unit-level generic instantiations
- immediately within a library-level package spec
- immediately within a library-level package body
- immediately following the `private` keyword of a library-level package spec
- immediately following the `begin` keyword of a library-level package body
- immediately following a library-level subprogram spec
- immediately within a library-level subprogram body
- immediately within a library-level task spec
- immediately within a library-level task body
- immediately following the `private` keyword of a library-level task spec
- immediately within a library-level protected spec
- immediately within a library-level protected body
- immediately following the `private` keyword of a library-level protected spec

The SPARK_Mode aspect can be used in the following places in the code:

- on a library-level package spec or body
- on a library-level subprogram spec or body
- on a library-level task spec or body
- on a library-level protected spec or body

If a SPARK_Mode pragma or aspect is not specified for a subprogram, package, task or protected spec/body, then its value is inherited from the current mode that is active at the point where the declaration occurs.

Note that a generic package instance is considered to be declared at its instantiation point. For example, a generic package cannot be both marked SPARK_Mode and instantiated in a subprogram body.

4.3.2 Consistency Rules

The basic rule is that you cannot turn SPARK_Mode back On, once you have explicitly turned if Off. So the following rules apply:

If a subprogram spec has SPARK_Mode Off, then the body cannot have SPARK_Mode On.

For a package, we have four parts:

1. the package public declarations
2. the package private part

3. the body of the package
4. the elaboration code after `begin`

For a package, the rule is that if you explicitly turn `SPARK_Mode` Off for any part, then all the following parts cannot have `SPARK_Mode` On. Note that this may require repeating a pragma `SPARK_Mode (Off)` in the body. For example, if we have a configuration pragma `SPARK_Mode (On)` that turns the mode On by default everywhere, and one particular package spec has pragma `SPARK_Mode (Off)`, then that pragma will need to be repeated in the package body.

Task types and protected types are handled similarly. If `SPARK_Mode` is set to Off on one part, it cannot be set to On on the following parts, among the three parts:

1. the spec
2. the private part
3. the body

There is an exception to this rule, when `SPARK_Mode` occurs in the code of a generic instantiated in code where `SPARK_Mode` is Off. In that case, occurrences of `SPARK_Mode` in the generic are ignored for this instance.

4.3.3 Examples of Use

Verifying Selected Subprograms

If only a few selected subprograms are in SPARK, then it makes sense to set no default for `SPARK_Mode`, and instead set `SPARK_Mode => On` directly on the subprograms of interest. For example:

```
package Selected_Subprograms is

   procedure Critical_Action with
     SPARK_Mode => On;

   procedure Sub_Action (X : out Boolean) with
     Post => X = True;

   procedure Non_Critical_Action;

end Selected_Subprograms;
```

Note that, although the bodies of procedures `Sub_Action` and `Non_Critical_Action` are not analyzed, it is valid to call `Sub_Action` in the body of procedure `Critical_Action`, even without specifying `SPARK_Mode => On` on the spec of `Sub_Action`. Indeed, GNATprove checks in that case that the spec of `Sub_Action` is in SPARK.

```
package body Selected_Subprograms is

   procedure Critical_Action with
     SPARK_Mode => On
   is
      --  this procedure body is analyzed
      X : Boolean;
   begin
      Sub_Action (X);
      pragma Assert (X = True);
   end Critical_Action;

```

(continues on next page)

(continued from previous page)

```
   procedure Sub_Action (X : out Boolean) is
   begin
      --  this procedure body is not analyzed
      X := True;
   end Sub_Action;

   procedure Non_Critical_Action is
   begin
      --  this procedure body is not analyzed
      null;
   end Non_Critical_Action;

end Selected_Subprograms;
```

Verifying Selected Units

If only a few selected units are in SPARK, then it makes sense to set no default for `SPARK_Mode`, and instead set `SPARK_Mode => On` directly on the units of interest. For example:

```
package Selected_Units with
  SPARK_Mode => On
is

   procedure Critical_Action;

   procedure Sub_Action (X : out Boolean) with
     Post => X = True;

   procedure Non_Critical_Action with
     SPARK_Mode => Off;

end Selected_Units;
```

Note that procedure `Sub_Action` can be called inside SPARK code, because its spec is in SPARK, even though its body is marked `SPARK_Mode => Off`. On the contrary, procedure `Non_Critical_Action` whose spec is marked `SPARK_Mode => Off` cannot be called inside SPARK code.

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

   procedure Critical_Action is
      --  this procedure body is analyzed
      X : Boolean;
   begin
      Sub_Action (X);
      pragma Assert (X = True);
   end Critical_Action;

   procedure Sub_Action (X : out Boolean) with
     SPARK_Mode => Off
```

(continues on next page)

(continued from previous page)

```
   is
   begin
      --  this procedure body is not analyzed
      X := True;
   end Sub_Action;

   procedure Non_Critical_Action with
     SPARK_Mode => Off
   is
   begin
      --  this procedure body is not analyzed
      null;
   end Non_Critical_Action;

end Selected_Units;
```

Excluding Selected Unit Bodies

If a unit spec is in SPARK, but its body is not in SPARK, the spec can be marked with `SPARK_Mode => On` and the body with `SPARK_Mode => Off`. This allows client code in SPARK to use this unit. If `SPARK_Mode` is On by default, then it need not be repeated on the unit spec.

```
package Exclude_Unit_Body with
  SPARK_Mode => On
is

   type T is private;

   function Get_Value return Integer;

   procedure Set_Value (V : Integer) with
     Post => Get_Value = V;

private
   pragma SPARK_Mode (Off);

   --  the private part of the package spec is not analyzed

   type T is access Integer;
end Exclude_Unit_Body;
```

Note that the private part of the spec (which is physically in the spec file, but is logically part of the implementation) can be excluded as well, by using a pragma `SPARK_Mode (Off)` at the start of the private part.

```
package body Exclude_Unit_Body with
  SPARK_Mode => Off
is
   --  this package body is not analyzed

   Value : T := new Integer;

```

(continues on next page)

(continued from previous page)

```
   function Get_Value return Integer is
   begin
      return Value.all;
   end Get_Value;

   procedure Set_Value (V : Integer) is
   begin
      Value.all := V;
   end Set_Value;

end Exclude_Unit_Body;
```

This scheme also works on generic units, which can then be instantiated both in code where SPARK_Mode is On, in which case only the body of the instantiated generic is excluded, or in code where SPARK_Mode is Off, in which case both the spec and the body of the instantiated generic are excluded.

```
generic
   type T is private;
package Exclude_Generic_Unit_Body with
  SPARK_Mode => On
is
   procedure Process (X : in out T);
end Exclude_Generic_Unit_Body;
```

```
package body Exclude_Generic_Unit_Body with
  SPARK_Mode => Off
is
   --  this package body is not analyzed
   procedure Process (X : in out T) is
   begin
      null;
   end Process;
end Exclude_Generic_Unit_Body;
```

```
with Exclude_Generic_Unit_Body;
pragma Elaborate_All (Exclude_Generic_Unit_Body);

package Use_Generic with
  SPARK_Mode => On
is
   --  the spec of this generic instance is analyzed
   package G1 is new Exclude_Generic_Unit_Body (Integer);

   procedure Do_Nothing;

end Use_Generic;
```

```
package body Use_Generic with
  SPARK_Mode => Off
is
   type T is access Integer;
```

(continues on next page)

(continued from previous page)

```
5
6     --  this generic instance is not analyzed
7     package G2 is new Exclude_Generic_Unit_Body (T);
8
9     procedure Do_Nothing is
10    begin
11       null;
12    end Do_Nothing;
13
14 end Use_Generic;
```

Excluding Selected Parts of a Unit

If most units are in SPARK except from some subprograms and packages, it makes sense to set the default to `SPARK_Mode (On)`, and set `SPARK_Mode => Off` on non-SPARK declarations. We assume here that a value of `SPARK_Mode => On` is specified as a configuration pragma.

```
1  package Exclude_Selected_Parts is
2
3     procedure Critical_Action;
4
5     procedure Non_Critical_Action;
6
7     package Non_Critical_Data with
8       SPARK_Mode => Off
9     is
10       type T is access Integer;
11       X : T;
12       function Get_X return Integer;
13    end Non_Critical_Data;
14
15 end Exclude_Selected_Parts;
```

Note that procedure `Non_Critical_Action` can be called inside SPARK code, because its spec is in SPARK, even though its body is marked `SPARK_Mode => Off`.

Note also that the local package `Non_Critical_Data` can contain any non-SPARK types, variables and subprograms, as it is marked `SPARK_Mode => Off`. It may be convenient to define such a local package to gather non-SPARK declarations, which allows to mark globally the unit `Exclude_Selected_Parts` with `SPARK_Mode => On`.

```
1  package body Exclude_Selected_Parts is
2
3     procedure Critical_Action is
4     begin
5        --  this procedure body is analyzed
6        Non_Critical_Action;
7     end Critical_Action;
8
9     procedure Non_Critical_Action with
10      SPARK_Mode => Off
11    is
12    begin
```

(continues on next page)

(continued from previous page)

```
      --  this procedure body is not analyzed
      null;
   end Non_Critical_Action;

   package body Non_Critical_Data with
     SPARK_Mode => Off
   is
      --  this package body is not analyzed
      function Get_X return Integer is
      begin
         return X.all;
      end Get_X;
   end Non_Critical_Data;

end Exclude_Selected_Parts;
```

CHAPTER

FIVE

OVERVIEW OF SPARK LANGUAGE

This chapter provides an overview of the SPARK language, detailing for each feature its consequences in terms of execution and formal verification. This is not a reference manual for the SPARK language, which can be found in:

- the Ada Reference Manual (for Ada features), and
- the SPARK Reference Manual (for SPARK-specific features)

More details on how GNAT compiles SPARK code can be found in the GNAT Reference Manual.

SPARK can be seen as a large subset of Ada with additional aspects/pragmas/attributes. It includes in particular:

- rich types (subtypes with bounds not known statically, discriminant records, subtype predicates, access types)
- flexible features to structure programs (function and operator overloading, early returns and exits, raise statements)
- code sharing features (generics, expression functions)
- object oriented features (tagged types, dispatching)
- concurrency features (tasks, protected objects)

In the rest of this chapter, the marker [Ada 2005] (resp. [Ada 2012] or [Ada 202X]) is used to denote that a feature defined in Ada 2005 (resp. Ada 2012 or Ada 202X) is supported in SPARK, and the marker [Ravenscar/Jorvik] is used to denote that a concurrency feature from Ada which belongs to the Ravenscar or Jorvik profiles is supported in SPARK. The marker [SPARK] is used to denote that a feature is specific to SPARK. Both the GNAT compiler and GNATprove analyzer support all features listed here.

Some code snippets presented in this section are available in the example called `gnatprove_by_example` distributed with the SPARK toolset. It can be found in the `share/examples/spark` directory below the directory where the toolset is installed, and can be accessed from the IDE (either GNAT Studio or GNATBench) via the *Help* → *SPARK* → *Examples* menu item.

5.1 Language Restrictions

5.1.1 Excluded Ada Features

To facilitate formal verification, SPARK enforces a number of global simplifications to Ada. The most notable simplifications are:

- Uses of access types and allocators must follow an ownership policy, so that only one access object has read-write permission to some allocated memory at any given time, or only read-only permission for that allocated memory is granted to possibly multiple access objects. See *Memory Ownership Policy*.

- All expressions (including function calls) are free of side effects, at the exception of calls to so-called functions with side effects (see *Aspect Side_Effects*) which can only appear as the right-hand side of assignments. Allowing functions with side effects everywhere could lead to non-deterministic evaluation due to conflicting side effects in sub-expressions of an enclosing expression. Allowing all functions to have side effects would conflict with the need to treat functions mathematically in specifications.
- Aliasing of names is not permitted. Aliasing may lead to unexpected interferences, in which the value denoted locally by a given name changes as the result of an update to another locally named variable. Formal verification of programs with aliasing is less precise and requires more manual work. See *Absence of Interferences*.
- The backward goto statement is not permitted. Backward gotos can be used to create loops, which require a specific treatment in formal verification, and thus should be precisely identified. See *Loop Invariants* and *Loop Variants*.
- The use of controlled types is not permitted. Controlled types lead to the insertion of implicit calls by the compiler. Formal verification of implicit calls makes it harder for users to interact with formal verification tools, as there is no source code on which information can be reported.
- Functions should always terminate when called on inputs satisfying the precondition, at the exception of so-called functions with side effects (see *Aspect Side_Effects*). While care is taken in GNATprove to detect possibilities of unsoundness resulting from nonterminating functions, it is possible that axioms generated for infeasible contracts may lead to unsoundness. See *Infeasible Subprogram Contracts*.
- Generic code is not analyzed directly. Doing so would require lengthy contracts on generic parameters, and would restrict the kind of code that can be analyzed, e.g. by forcing the variables read/written by a generic subprogram parameter. Instead, instantiations of generic code are analyzed in SPARK. See *Analysis of Generics*.

As formal verification technology advances the list will be revisited and it may be possible to relax some of these restrictions.

Uses of these features in SPARK code are detected by GNATprove and reported as errors. Formal verification is not possible on subprograms using these features. But these features can be used in subprograms in Ada not identified as SPARK code, see *Identifying SPARK Code*.

5.1.2 Sizes of Objects

GNATprove generally only knows the values of the `Size` and `Object_Size` attributes in simple cases such as scalar objects. For any more complex types such as arrays and records, the value of these attributes is unknown, and e.g. assertions referring to them remain unproved. The user can indicate the values of these attributes to SPARK via confirming representation clauses, using `for Type'Size use ...` or the aspect syntax `with Size => ...`. Only static values can be used in these representation aspects or clauses, which can only be used on type declarations and not on subtype declarations.

Note that for an object `X` of type `T`, the value of `X'Size` is *not* necessarily equal to `T'Size`, but equal to `T'Object_Size`. So it is generally more useful to specify `Object_Size` on types to be able to know the value the `Size` attribute of the type's objects. However, to compute the size of `T'Object_Size` for composite types, the value of `C'Size` is generally used, `C` being the type of a component. The value of `Object_Size` must be 8, 16, 32 or a multiple of 64, while the `Size` of a type can be any value.

Attributes `Size` and `Object_Size` are specific to a subtype. As such, it is not known if a subtype has the same value for these attributes as its base type, including when the subtype does not introduce any constraint as in `subtype S is T`.

The following code example shows some simple representation clauses using the aspect syntax:

```
   --  SPARK knows the 'Size and 'Object_Size of scalar types
   type Short_Short is range -128 .. 127;
```

(continues on next page)

(continued from previous page)

```
   type U8 is mod 2 ** 8;

   --  The following representation clauses are not needed, but serve to
   --  illustrate that in the record type declaration below, U7'Size (and not
   --  U7'Object_Size) is used to check the Object_Size of the record type.
   type U7 is mod 2 ** 7
   with Size => 7,
        Object_Size => 8;

   --  Without the packing instruction, the compiler would complain that
   --  objects of type R do not fit into 8 bits.
   type R is record
      A : U7;
      B : Boolean;
   end record
   with Pack, Object_Size => 8;
```

5.1.3 Data Validity

SPARK reinforces the strong typing of Ada with a stricter initialization policy (see *Data Initialization Policy*), and thus provides no means of specifying that some input data may be invalid. This has some impact on language features that process or potentially produce invalid values. SPARK issues checks specific to data validity on two language constructs:

- Calls to instances of `Unchecked_Conversion`
- Objects with a supported address clause (so-called overlays). An address clause is supported if it is of the form `with Address => Y'Address`, where `Y` is another object, and `Y` is part of a statically known object.

For occurences of these patterns, SPARK checks that no invalid values can be produced. Given that no invalid values can be constructed in SPARK, the evaluation of the attribute `Valid` is assumed to always return True.

These validity checks are illustrated in the following example:

```
1  package Validity with
2    SPARK_Mode
3  is
4
5     procedure Convert (X : Integer; Y : out Float);
6
7  end Validity;
```

```
1  with Ada.Unchecked_Conversion;
2
3  package body Validity with
4    SPARK_Mode
5  is
6
7     function Int_To_Float is new Ada.Unchecked_Conversion (Integer, Float);
8
9     procedure Convert (X : Integer; Y : out Float) is
10    begin
11       pragma Assert (X'Valid);
```

(continues on next page)

(continued from previous page)

```
      Y := Int_To_Float (X);
      pragma Assert (Y'Valid);
   end Convert;

end Validity;
```

GNATprove proves both assertions, but issues warnings about its assumptions that the evaluation of attribute `Valid` on both input parameter `X` and the result of the call to `Unchecked_Conversion` return True. It also issues a "high" unproved check that the unchecked conversion to `Float` may produce invalid values (for example, if an `Integer` is converted whose bit representation corresponds to a `NaN` float, which is not allowed in SPARK).

```
validity.adb:7:13: info: types in unchecked conversion have the same size
validity.adb:7:59: info: type is suitable as source for unchecked conversion

validity.adb:7:68: high: type is unsuitable as a target for unchecked conversion
    7 |   function Int_To_Float is new Ada.Unchecked_Conversion (Integer, Float);
      |                                                                    ^~~~~
  possible explanation: floating-point types have invalid bit patterns for SPARK

validity.adb:11:22: warning: attribute Valid is assumed to return True
   11 |      pragma Assert (X'Valid);
      |                     ^~~~~~~
validity.adb:11:22: info: assertion proved

validity.adb:13:22: warning: attribute Valid is assumed to return True
   13 |      pragma Assert (Y'Valid);
      |                     ^~~~~~~
validity.adb:13:22: info: assertion proved
validity.ads:5:36: info: initialization of "Y" proved
```

When checking an instance of `Unchecked_Conversion`, GNATprove also checks that both types have the same `Object_Size`. For non-scalar types, GNATprove doesn't know the `Object_Size` of the types, so representation clauses that specify `Object_Size` are required to prove such checks (see also *Sizes of Objects*). Similarly, for object declarations with an Address clause or aspect that refers to the `'Address` of another object, SPARK checks that both objects have the same known `Object_Size`.

SPARK allows conversions from (suitable) integer types or `System.Address_Type` to general access-to-object types. When calling such instances of `Unchecked_Conversion`, GNATprove makes some assumptions about the result of the call:

- The designated data has no aliases if it is an access-to-variable type and no mutable aliases otherwise.
- The returned object is a valid access and it designates a valid value of its type.

At each call to such `Unchecked_Conversion`, GNATprove raises warnings to notify the user that these assumptions need to be ascertained by other means.

Conversions from integer types or `System.Address_Type` to pool-specific access-to-object types are still forbidden, as these pointers should not be deallocated nor considered when checking for memory leaks. Conversions from access-to-object types to integer types or `System.Address_Type` are still forbidden, because SPARK does not handle addresses.

```
with Ada.Unchecked_Conversion;
with Interfaces; use Interfaces;
with System;     use System;

```

(continues on next page)

(continued from previous page)

```
5  package UC_To_Access with SPARK_Mode => On is
6     type Int_Access is access all Integer;
7
8     function Uns_To_Int_Access is new Ada.Unchecked_Conversion (Unsigned_64, Int_Access);
9     --  Accepted with warnings
10    function Uns_From_Int_Access is new Ada.Unchecked_Conversion (Int_Access, Unsigned_
   ↪64);
11    --  Rejected
12
13    C1 : constant Int_Access := Uns_To_Int_Access (30);
14
15    function Addr_To_Int_Access is new Ada.Unchecked_Conversion (Address, Int_Access);
16    --  Accepted with warnings
17    function Addr_From_Int_Access is new Ada.Unchecked_Conversion (Int_Access, Address);
18    --  Rejected
19
20    Addr : Address with Import;
21    C2 : constant Int_Access := Addr_To_Int_Access (Addr);
22
23    type PS_Access is access Integer;
24
25    function Uns_To_PS_Access is new Ada.Unchecked_Conversion (Unsigned_64, PS_Access);
26    --  Rejected
27 end UC_To_Access;
```

```
uc_to_access.ads:10:13: error: unchecked conversion instance from a type with access␣
 ↪subcomponents is not allowed in SPARK
uc_to_access.ads:10:13: error: violation of aspect SPARK_Mode at line 5
uc_to_access.ads:13:32: warning: call to "Uns_To_Int_Access" is assumed to return a␣
 ↪valid access designating a valid value
uc_to_access.ads:13:32: warning: the value returned by a call to "Uns_To_Int_Access" is␣
 ↪assumed to have no aliases
uc_to_access.ads:17:13: error: unchecked conversion instance from a type with access␣
 ↪subcomponents is not allowed in SPARK
uc_to_access.ads:17:13: error: violation of aspect SPARK_Mode at line 5
uc_to_access.ads:21:32: warning: call to "Addr_To_Int_Access" is assumed to return a␣
 ↪valid access designating a valid value
uc_to_access.ads:21:32: warning: the value returned by a call to "Addr_To_Int_Access" is␣
 ↪assumed to have no aliases
uc_to_access.ads:25:13: error: unchecked conversion instance to a pool-specific access␣
 ↪type is not allowed in SPARK
uc_to_access.ads:25:13: error: violation of aspect SPARK_Mode at line 5
gnatprove: error during flow analysis and proof
```

5.1.4 Data Initialization Policy

Modes on parameters and data dependency contracts (see *Data Dependencies*) in SPARK have a stricter meaning than in Ada:

- Parameter mode `in` (resp. global mode `Input`) indicates that the object denoted in the parameter (resp. data dependencies) should be completely initialized before calling the subprogram. It should not be written in the subprogram.
- Parameter mode `out` (resp. global mode `Output`) indicates that the object denoted in the parameter (resp. data dependencies) should be completely initialized before returning from the subprogram. It should not be read in the program prior to initialization.
- Parameter mode `in out` (resp. global mode `In_Out`) indicates that the object denoted in the parameter (resp. data dependencies) should be completely initialized before calling the subprogram. It can be written in the subprogram.
- Global mode `Proof_In` indicates that the object denoted in the data dependencies should be completely initialized before calling the subprogram. It should not be written in the subprogram, and only read in contracts and assertions.

Hence, all inputs should be completely initialized at subprogram entry, and all outputs should be completely initialized at subprogram output. Similarly, all objects should be completely initialized when read (e.g. inside subprograms), at the exception of record subcomponents (but not array subcomponents) provided the subcomponents that are read are initialized.

A consequence of the rules above is that a parameter (resp. global variable) that is partially written in a subprogram should be marked as `in out` (resp. `In_Out`), because the input value of the parameter (resp. global variable) is *read* when returning from the subprogram.

GNATprove will issue check messages if a subprogram does not respect the aforementioned data initialization policy. For example, consider a procedure `Proc` which has a parameter and a global item of each mode:

```
1  package Data_Initialization with
2    SPARK_Mode
3  is
4     type Data is record
5        Val : Float;
6        Num : Natural;
7     end record;
8
9     G1, G2, G3 : Data;
10
11    procedure Proc
12      (P1 : in     Data;
13       P2 :    out Data;
14       P3 : in out Data)
15    with
16      Global => (Input  => G1,
17                 Output => G2,
18                 In_Out => G3);
19
20    procedure Call_Proc with
21      Global => (Output => (G1, G2, G3));
22
23 end Data_Initialization;
```

Procedure `Proc` should completely initialize its outputs `P2` and `G2`, but it only initalizes them partially. Similarly, procedure `Call_Proc` which calls `Proc` should completely initalize all of `Proc`'s inputs prior to the call, but it only initalizes `G1` completely.

```
package body Data_Initialization with
  SPARK_Mode
is

   procedure Proc
     (P1 : in     Data;
      P2 :    out Data;
      P3 : in out Data) is
   begin
      P2.Val := 0.0;
      G2.Num := 0;
      --  fail to completely initialize P2 and G2 before exit
   end Proc;

   procedure Call_Proc is
      X1, X2, X3 : Data;
   begin
      X1.Val := 0.0;
      X3.Num := 0;
      G1.Val := 0.0;
      G1.Num := 0;
      --  fail to completely initialize X1, X3 and G3 before call
      Proc (X1, X2, X3);
   end Call_Proc;

end Data_Initialization;
```

On this program, GNATprove issues 6 high check messages, corresponding to the violations of the data initialization policy:

```
data_initialization.adb:23:07: high: "G3" is not an input in the Global contract of␣
↪subprogram "Call_Proc" at data_initialization.ads:20
   23 |      Proc (X1, X2, X3);
      |      ^~~~~~~~~~~~~~~~
  either make "G3" an input in the Global contract or initialize it before use

data_initialization.adb:23:13: high: "X1.Num" is not initialized
   23 |      Proc (X1, X2, X3);
      |            ^~

data_initialization.adb:23:17: warning: "X2" is set by "Proc" but not used after the call
   23 |      Proc (X1, X2, X3);
      |                ^~

data_initialization.adb:23:21: warning: "X3" is set by "Proc" but not used after the call
   23 |      Proc (X1, X2, X3);
      |                    ^~

data_initialization.adb:23:21: high: "X3.Val" is not initialized
```

(continues on next page)

(continued from previous page)

```
   23 |      Proc (X1, X2, X3);
      |                    ^~

data_initialization.ads:12:07: warning: unused variable "P1"
   12 |     (P1 : in     Data;
      |      ^~

data_initialization.ads:13:07: high: "P2.Num" is not initialized in "Proc"
   13 |      P2 :    out Data;
      |      ^~
  reason for check: OUT parameter should be fully initialized on return
  possible fix: initialize "P2.Num" on all paths, make "P2" an IN OUT parameter or␣
↪annotate it with aspect Relaxed_Initialization

data_initialization.ads:14:07: warning: "P3" is not modified, could be IN
   14 |      P3 : in out Data)
      |      ^~

data_initialization.ads:14:07: warning: unused variable "P3"
   14 |      P3 : in out Data)
      |      ^~

data_initialization.ads:16:27: low: unused global "G1"
   16 |     Global => (Input  => G1,
      |                           ^~

data_initialization.ads:17:27: high: "G2.Val" is not initialized
   17 |                Output => G2,
      |                           ^~

data_initialization.ads:18:27: warning: "G3" is not modified, could be INPUT
   18 |                In_Out => G3);
      |                           ^~

data_initialization.ads:18:27: low: unused global "G3"
   18 |                In_Out => G3);
      |                           ^~
```

While a user can justify individually such messages with pragma `Annotate` (see section *Justifying Check Messages*), it is under her responsibility to then ensure correct initialization of subcomponents that are read, as GNATprove relies during proof on the property that data is properly initialized before being read.

Note also the various low check messages and warnings that GNATprove issues on unused parameters, global items and assignments, also based on the stricter SPARK interpretation of parameter and global modes.

It is possible to opt out of the strong data initialization policy of SPARK on a case by case basis using the aspect `Relaxed_Initialization` (see section *Aspect Relaxed_Initialization and Ghost Attribute Initialized*). Parts of objects subject to this aspect only need to be initialized when actually read. Using `Relaxed_Initialization` requires specifying data initialization through contracts that are verified by proof (as opposed to flow analysis). Thus, `Relaxed_Initialization` should only be used when needed as it requires more effort to verify data initialization from both the user and the tool.

5.1.5 Memory Ownership Policy

In SPARK, access values (a.k.a. pointers) are only allowed to alias in known ways, so that formal verification can be applied *as if* allocated memory pointed to by access values was a component of the access value seen as a record object.

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.

For example, in the following example:

```
procedure Ownership_Transfer with
  SPARK_Mode
is
   type Int_Ptr is access Integer;
   X   : Int_Ptr;
   Y   : Int_Ptr;
   Tmp : Integer;
begin
   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
   Tmp   := X.all;      --  illegal
end Ownership_Transfer;
```

GNATprove correctly detects that `X.all` can neither be read nor written after the assignment of `X` to `Y` and issues corresponding messages:

```
ownership_transfer.adb:13:06: error: dereference from "X" is not writable
   13 |   X.all := X.all + 1;  --  illegal
      |   ~~^~~
  object was moved at line 11 [E0010]
   11 |   Y := X;
      |        ^ here
  launch "gnatprove --explain=E0010" for more information

ownership_transfer.adb:13:15: error: dereference from "X" is not readable
   13 |   X.all := X.all + 1;  --  illegal
      |            ~~^~~
  object was moved at line 11 [E0010]
   11 |   Y := X;
      |        ^ here
  launch "gnatprove --explain=E0010" for more information

ownership_transfer.adb:14:06: error: dereference from "X" is not writable
   14 |   X.all := 1;          --  illegal
      |   ~~^~~
  object was moved at line 11 [E0010]
   11 |   Y := X;
      |        ^ here
  launch "gnatprove --explain=E0010" for more information
```

(continues on next page)

(continued from previous page)

```
ownership_transfer.adb:15:15: error: dereference from "X" is not readable
   15 |   Tmp   := X.all;      --  illegal
      |            ~~^~~
  object was moved at line 11 [E0010]
   11 |   Y := X;
      |        ^ here
  launch "gnatprove --explain=E0010" for more information
gnatprove: error during flow analysis and proof
```

At call site, ownership is similarly transferred to the callee's parameters for the duration of the call, and returned to the actual parameters (a.k.a. arguments) when returning from the call.

For example, in the following example:

```
procedure Ownership_Transfer_At_Call with
  SPARK_Mode
is
   type Int_Ptr is access Integer;
   X : Int_Ptr;

   procedure Proc (Y : in out Int_Ptr)
     with Global => (In_Out => X)
   is
   begin
      Y.all := Y.all + 1;
      X.all := X.all + 1;
   end Proc;

begin
   X := new Integer'(1);
   X.all := X.all + 1;
   Proc (X);  --  illegal
end Ownership_Transfer_At_Call;
```

GNATprove correctly detects that the call to `Proc` cannot take `X` in argument as `X` is already accessed as a global variable by `Proc`.

```
  possible fix: subprogram at line 7 should mention X in a precondition
    7 |   procedure Proc (Y : in out Int_Ptr)
      |   ^ here
```

It is also possible to transfer the ownership of an object temporarily, for the duration of the lifetime of a local object. This can be achieved by declaring a local object of an anonymous access type and initializing it with a part of an existing object. In the following example, `B` temporarily borrows the ownership of `X`:

```
procedure Ownership_Borrowing with
  SPARK_Mode
is
   type Int_Ptr is access Integer;
   X   : Int_Ptr := new Integer'(1);
   Tmp : Integer;
begin
   declare
```

(continues on next page)

(continued from previous page)

```
      B : access Integer := X;
   begin
      B.all := B.all + 1;
      X.all := X.all + 1;  --  illegal
      X.all := 1;          --  illegal
      Tmp   := X.all;      --  illegal
   end;
   X.all := X.all + 1;
   X.all := 1;
   Tmp   := X.all;
end Ownership_Borrowing;
```

During the lifetime of `B`, it is incorrect to either read or modify `X`, but complete ownership is restored to `X` when `B` goes out of scope. GNATprove correctly detects that reading or assigning to `X` in the scope of `B` is incorrect.

```
ownership_borrowing.adb:12:09: error: object was borrowed at line 9
   12 |      X.all := X.all + 1;  --  illegal
      |      ~~^~~

ownership_borrowing.adb:12:18: error: object was borrowed at line 9
   12 |      X.all := X.all + 1;  --  illegal
      |               ~~^~~

ownership_borrowing.adb:13:09: error: object was borrowed at line 9
   13 |      X.all := 1;          --  illegal
      |      ~~^~~

ownership_borrowing.adb:14:18: error: object was borrowed at line 9
   14 |      Tmp   := X.all;      --  illegal
      |               ~~^~~
gnatprove: error during flow analysis and proof
```

It is also possible to only transfer read access to a local variable. This happens when the variable has an anonymous access-to-constant type, as in the following example:

```
procedure Ownership_Observing with
  SPARK_Mode
is
   type Int_Ptr is access Integer;
   X   : Int_Ptr := new Integer'(1);
   Tmp : Integer;
begin
   declare
      B : access constant Integer := X;
   begin
      Tmp   := B.all;
      Tmp   := X.all;
      X.all := X.all + 1;  --  illegal
      X.all := 1;          --  illegal
   end;
   X.all := X.all + 1;
```

(continues on next page)

(continued from previous page)

```
17    X.all := 1;
18    Tmp   := X.all;
19 end Ownership_Observing;
```

In this case, we say that `B` observes the value of `X`. During the lifetime of an observer, it is illegal to move or modify the observed object. GNATprove correctly flags the write inside `X` in the scope of `B` as illegal. Note that reading `X` is still possible in the scope of `B`:

```
ownership_observing.adb:13:09: error: object was observed at line 9
   13 |      X.all := X.all + 1;  --  illegal
      |      ~~^~~

ownership_observing.adb:14:09: error: object was observed at line 9
   14 |      X.all := 1;          --  illegal
      |      ~~^~~
gnatprove: error during flow analysis and proof
```

5.1.6 Absence of Interferences

In SPARK, an assignment to a variable cannot change the value of another variable. This is enforced by restricting the use of access types (pointers) in SPARK, and by restricting aliasing between parameters and global variables so that only benign aliasing is accepted (i.e. aliasing that does not cause interference).

The precise rules detailed in SPARK RM 6.4.2 can be summarized as follows:

- Two mutable parameters should never be aliased.
- An immutable and a mutable parameters should not be aliased, unless the immutable parameter is always passed by copy.
- A mutable parameter should never be aliased with a global variable referenced by the subprogram.
- An immutable parameter should not be aliased with a global variable referenced by the subprogram, unless the immutable parameter is always passed by copy.

An immutable parameter is either an input parameter that is not of an access type, or an anonymous access-to-constant parameter. Except for parameters of access types, the immutable/mutable distinction is the same as the input/output one.

These rules extend the existing rules in Ada RM 6.4.1 for restricting aliasing, which already make it illegal to call a procedure with problematic (non-benign) aliasing between parameters of scalar type that are *known to denote the same object* (a notion formally defined in Ada RM).

For example, in the following example:

```
1  package Aliasing with
2    SPARK_Mode
3  is
4     Glob : Integer;
5
6     procedure Whatever (In_1, In_2 : Integer; Out_1, Out_2 : out Integer) with
7       Global => Glob;
8
9  end Aliasing;
```

Procedure `Whatever` can only be called on arguments that satisfy the following constraints:

1. Arguments for `Out_1` and `Out_2` should not be aliased.
2. Variable `Glob` should not be passed in argument for `Out_1` and `Out_2`.

Note that there are no constraints on input parameters `In_1` and `In_2`, as these are always passed by copy (being of a scalar type). This would not be the case if these input parameters were of a record or array type.

For example, here are examples of correct and illegal (according to Ada and SPARK rules) calls to procedure `Whatever`:

```
with Aliasing; use Aliasing;

procedure Check_Param_Aliasing with
  SPARK_Mode
is
   X, Y, Z : Integer := 0;
begin
   Whatever (In_1 => X, In_2 => X, Out_1 => X, Out_2 => X);  --  illegal
   Whatever (In_1 => X, In_2 => X, Out_1 => X, Out_2 => Y);  --  correct
   Whatever (In_1 => X, In_2 => X, Out_1 => Y, Out_2 => X);  --  correct
   Whatever (In_1 => Y, In_2 => Z, Out_1 => X, Out_2 => X);  --  illegal
end Check_Param_Aliasing;
```

GNATprove (like GNAT compiler, since these are also Ada rules) correctly detects the two illegal calls and issues errors:

```
check_param_aliasing.adb:8:45: error: writable actual for "Out_1" overlaps with actual␣
↪for "Out_2"
    8 |   Whatever (In_1 => X, In_2 => X, Out_1 => X, Out_2 => X);  --  illegal
      |                                             ^ here

check_param_aliasing.adb:11:45: error: writable actual for "Out_1" overlaps with actual␣
↪for "Out_2"
   11 |   Whatever (In_1 => Y, In_2 => Z, Out_1 => X, Out_2 => X);  --  illegal
      |                                             ^ here
gnatprove: error during generation of Global contracts
```

Here are other examples of correct and incorrect calls (according to SPARK rules) to procedure `Whatever`:

```
with Aliasing; use Aliasing;

procedure Check_Aliasing with
  SPARK_Mode
is
   X, Y, Z : Integer := 0;
begin
   Whatever (In_1 => X, In_2 => X, Out_1 => X, Out_2 => Glob);     --  incorrect
   Whatever (In_1 => X, In_2 => Y, Out_1 => Z, Out_2 => Glob);     --  incorrect
   Whatever (In_1 => Glob, In_2 => Glob, Out_1 => X, Out_2 => Y);  --  correct
end Check_Aliasing;
```

GNATprove correctly detects the two incorrect calls and issues high check messages:

```
check_aliasing.adb:8:57: high: formal parameter "Out_2" and global "Glob" are aliased␣
↪(SPARK RM 6.4.2)
    8 |   Whatever (In_1 => X, In_2 => X, Out_1 => X, Out_2 => Glob);     --  incorrect
      |                                                   ^~~~
check_aliasing.adb:9:57: high: formal parameter "Out_2" and global "Glob" are aliased␣
↪(SPARK RM 6.4.2)
    9 |   Whatever (In_1 => X, In_2 => Y, Out_1 => Z, Out_2 => Glob);     --  incorrect
      |                                                   ^~~~
```

Note that SPARK currently does not detect aliasing between objects that arises due to the use of Address clauses or aspects.

5.1.7 Analysis of Generics

GNATprove does not directly analyze the code of generics. The following message is issued if you call GNATprove on a generic unit:

```
warning: no bodies have been analyzed by GNATprove
enable analysis of a non-generic body using SPARK_Mode
```

The advice given is to use `SPARK_Mode` on non-generic code, for example an instantiation of the generic unit. As `SPARK_Mode` aspect cannot be attached to a generic instantiation, it should be specified on the enclosing context, either through a pragma or aspect.

For example, consider the following generic increment procedure:

```
generic
   type T is range <>;
procedure Generic_Increment (X : in out T) with
  SPARK_Mode,
  Pre  => X < T'Last,
  Post => X = X'Old + 1;
```

```
procedure Generic_Increment (X : in out T) with
  SPARK_Mode
is
begin
   X := X + 1;
end Generic_Increment;
```

Procedure `Instance_Increment` is a specific instance of `Generic_Increment` for the type `Integer`:

```
pragma SPARK_Mode;
with Generic_Increment;

procedure Instance_Increment is new Generic_Increment (Integer);
```

GNATprove analyzes this instantiation and reports messages on the generic code, always stating to which instantiation the messages correspond to:

```
generic_increment.ads:6:11: info: postcondition proved, in instantiation at instance_
↪increment.ads:4
generic_increment.ads:6:21: info: overflow check proved, in instantiation at instance_
```

(continues on next page)

(continued from previous page)

```
↪increment.ads:4
generic_increment.adb:5:11: info: overflow check proved, in instantiation at instance_
↪increment.ads:4
```

Thus, it is possible that some checks are proved on an instance and not on another one. In that case, the chained locations in the messages issued by GNATprove allow you to locate the problematic instantiation. In order to prove a generic library for all possible uses, you should choose extreme values for the generic parameters such that, if these instantiations are proved, any other choice of parameters will be provable as well.

5.2 Subprogram Contracts

The most important feature to specify the intended behavior of a SPARK program is the ability to attach a contract to subprograms. In this document, a *subprogram* can be a procedure, a function or a protected entry. This contract is made up of various optional parts:

- The *precondition* introduced by aspect `Pre` specifies constraints on callers of the subprogram.
- The *postcondition* introduced by aspect `Post` specifies (partly or completely) the functional behavior of the subprogram.
- The *contract cases* introduced by aspect `Contract_Cases` is a way to partition the behavior of a subprogram. It can replace or complement a precondition and a postcondition.
- The *data dependencies* introduced by aspect `Global` specify the global data read and written by the subprogram.
- The *flow dependencies* introduced by aspect `Depends` specify how subprogram outputs depend on subprogram inputs.
- The *exceptional contract* introduced by aspect `Exceptional_Cases` specifies the exceptions that might be propagated by a procedure, along with exceptional postconditions.
- The *termination contract* introduced by aspect `Always_Terminates` requires procedures and entries to terminate, possibly under a particular condition.
- The *subprogram variant* introduced by aspect `Subprogram_Variant` is used to ensure termination of recursive subprograms.

Which contracts to write for a given verification objective, and how GNATprove generates default contracts, is detailed in *How to Write Subprogram Contracts*.

GNATprove formally verifies that each execution of each SPARK subprogram it analyzes will either:

- return normally in a state that respects the subprogram's postcondition,
- raise an exception in a state that respects the subprogram's exceptional contract,
- terminate abnormally as a result of a primary stack, secondary stack, or heap memory allocation failure, or
- not terminate at all when it is allowed by its termination contract.

GNATprove also checks that procedures that are marked with aspect or pragma `No_Return` do not return: they should either raise an exception, call a non-returning subprogram, or loop forever on any input.

5.2.1 Preconditions

[Ada 2012]

The precondition of a subprogram specifies constraints on callers of the subprogram. Typically, preconditions are written as conjunctions of constraints that fall in one of the following categories:

- exclusion of forbidden values of parameter, for example `X /= 0` or `Y not in Active_States`
- specification of allowed parameter values, for example `X in 1 .. 10` or `Y in Idle_States`
- relations that should hold between parameter values, for example `(if Y in Active_State then Z /= Null_State)`
- expected values of global variables denoting the state of the computation, for example `Current_State in Active_States`
- invariants about the global state that should hold when calling this subprogram, for example `Is_Complete (State_Mapping)`
- relations involving the global state and input parameters that should hold when calling this subprogram, for example `X in Next_States (Global_Map, Y)`

When the program is compiled with assertions (for example with switch `-gnata` in GNAT), the precondition of a subprogram is checked at run time every time the subprogram is called. An exception is raised if the precondition fails. Not all assertions need to be enabled though. For example, a common idiom is to enable only preconditions (and not other assertions) in the production binary, by setting pragma `Assertion_Policy` as follows:

```
pragma Assertion_Policy (Pre => Check);
```

When a subprogram is analyzed with GNATprove, its precondition is used to restrict the contexts in which it may be executed, which is required in general to prove that the subprogram's implementation:

- is free from run-time errors (see *Writing Contracts for Program Integrity*); and
- ensures that the postcondition of the subprogram always holds (see *Writing Contracts for Functional Correctness*).

In particular, the default precondition of `True` used by GNATprove when no explicit one is given may not be precise enough, unless it can be analyzed in the context of its callers by GNATprove (see *Contextual Analysis of Subprograms Without Contracts*). When a caller is analyzed with GNATprove, it checks that the precondition of the called subprogram holds at the point of call. And even when the implementation of the subprogram is not analyzed with GNATprove, it may be necessary to add a precondition to the subprogram for analyzing its callers (see *Writing Contracts on Imported Subprograms*).

For example, consider the procedure `Add_To_Total` which increments global counter `Total` by the value given in parameter `Incr`. To ensure that there are no integer overflows in the implementation, `Incr` should not be too large, which a user can express with the following precondition:

```
procedure Add_To_Total (Incr : in Integer) with
  Pre => Incr >= 0 and then Total <= Integer'Last - Incr;
```

To ensure that the value of `Total` remains non-negative, one should also add the condition `Total >= 0` to the precondition:

```
procedure Add_To_Total (Incr : in Integer) with
  Pre => Incr >= 0 and then Total in 0 .. Integer'Last - Incr;
```

Finally, GNATprove also analyzes preconditions to ensure that they are free from run-time errors in all contexts. This may require writing the precondition in a special way. For example, the precondition of `Add_To_Total` above uses the

shortcut boolean operator `and then` instead of `and`, so that calling the procedure in a context where `Incr` is negative does not result in an overflow when evaluating `Integer'Last - Incr`. Instead, the use of `and then` ensures that a precondition failure will occur before the expression `Integer'Last - Incr` is evaluated.

Note: It is good practice to use the shortcut boolean operator `and then` instead of `and` in preconditions. This is required in some cases by GNATprove to prove absence of run-time errors inside preconditions.

Raise expressions occuring in preconditions are handled in a special way. Indeed, it is a common pattern to use a raise expression to change the exception raised by a failed precondition. To support this use case, raising an expression in a precondition is considered in SPARK to be a failure of the precondition, as opposed to a runtime failure, which would not be allowed in SPARK. As an example, we may want to introduce specific exceptions for the the failure of each part of the precondition of `Add_To_Total`, so as to debug them more easily. This can be done by using two raise expressions as in the following snippet:

```
Negative_Increment  : exception;
Total_Out_Of_Bounds : exception;

procedure Add_To_Total (Incr : in Integer) with
  Pre => (Incr >= 0 or else raise Negative_Increment)
  and then (Total in 0 .. Integer'Last - Incr
            or else raise Total_Out_Of_Bounds);
```

The raise expressions are associated to each conjunct using an `or else` short circuit operator, so that they will be evaluated when the conjunct evaluates to `False` and the exception will be raised.

On this code, GNATprove will not attempt to verify that the exceptions can never be raised when evaluating the precondition in any context, like it does for other runtime exceptions. Instead, it will consider them being raised as a failure of the precondition. So, for GNATprove, the precondition with the raise expressions above is effectively equivalent to the precondition of the previous example.

5.2.2 Postconditions

[Ada 2012]

The postcondition of a subprogram specifies partly or completely the functional behavior of the subprogram. Typically, postconditions are written as conjunctions of properties that fall in one of the following categories:

- possible values returned by a function, using the special attribute `Result` (see *Attribute Result*), for example `Get'Result in Active_States`
- possible values of output parameters, for example `Y in Active_States`
- expected relations between output parameter values, for example `if Success then Y /= Null_State`
- expected relations between input and output parameter values, possibly using the special attribute `Old` (see *Attribute Old*), for example `if Success then Y /= Y'Old`
- expected values of global variables denoting updates to the state of the computation, for example `Current_State in Active_States`
- invariants about the global state that should hold when returning from this subprogram, for example `Is_Complete (State_Mapping)`
- relations involving the global state and output parameters that should hold when returning from this subprogram, for example `X in Next_States (Global_Map, Y)`

When the program is compiled with assertions (for example with switch `-gnata` in GNAT), the postcondition of a subprogram is checked at run time every time the subprogram returns. An exception is raised if the postcondition fails. Usually, postconditions are enabled during tests, as they provide dynamically checkable oracles of the intended behavior of the program, and disabled in the production binary for efficiency.

When a subprogram is analyzed with GNATprove, it checks that the postcondition of a subprogram cannot fail. This verification is modular: GNATprove considers all calling contexts in which the precondition of the subprogram holds for the analysis of a subprogram. GNATprove also analyzes postconditions to ensure that they are free from run-time errors, like any other assertion.

For example, consider the procedure `Add_To_Total` which increments global counter `Total` with the value given in parameter `Incr`. This intended behavior can be expressed in its postcondition:

```
procedure Add_To_Total (Incr : in Integer) with
  Post => Total = Total'Old + Incr;
```

The postcondition of a subprogram is used to analyze calls to the subprograms. In particular, the default postcondition of `True` used by GNATprove when no explicit one is given may not be precise enough to prove properties of its callers, unless it analyzes the subprogam's implementation in the context of its callers (see *Contextual Analysis of Subprograms Without Contracts*).

Recursive subprograms and mutually recursive subprograms are treated in this respect exactly like non-recursive ones. Provided the execution of these subprograms always terminates (a property that is not verified by GNATprove), then GNATprove correctly checks that their postcondition is respected by using this postcondition for recursive calls.

Special care should be exercized for functions that return a boolean, as a common mistake is to write the expected boolean result as the postcondition:

```
function Total_Above_Threshold (Threshold : in Integer) return Boolean with
  Post => Total > Threshold;
```

while the correct postcondition uses *Attribute Result*:

```
function Total_Above_Threshold (Threshold : in Integer) return Boolean with
  Post => Total_Above_Threshold'Result = Total > Threshold;
```

Both GNAT compiler and GNATprove issue a warning on the semantically correct but likely functionally wrong postcondition.

5.2.3 Contract Cases

[SPARK]

When a subprogram has a fixed set of different functional behaviors, it may be more convenient to specify these behaviors as contract cases rather than a postcondition. For example, consider a variant of procedure `Add_To_Total` which either increments global counter `Total` by the given parameter value when possible, or saturates at a given threshold. Each of these behaviors can be defined in a contract case as follows:

```
procedure Add_To_Total (Incr : in Integer) with
  Contract_Cases => (Total + Incr < Threshold  => Total = Total'Old + Incr,
                     Total + Incr >= Threshold => Total = Threshold);
```

Each contract case consists in a guard and a consequence separated by the symbol =>. When the guard evaluates to `True` on subprogram entry, the corresponding consequence should also evaluate to `True` on subprogram exit. We say that this contract case was enabled for the call. Exactly one contract case should be enabled for each call, or said equivalently, the contract cases should be disjoint and complete.

For example, the contract cases of `Add_To_Total` express that the subprogram should be called in two distinct cases only:

- on inputs that can be added to `Total` to obtain a value strictly less than a given threshold, in which case `Add_To_Total` adds the input to `Total`.
- on inputs whose addition to `Total` exceeds the given threshold, in which case `Add_To_Total` sets `Total` to the threshold value.

When the program is compiled with assertions (for example with switch `-gnata` in GNAT), all guards are evaluated on entry to the subprogram, and there is a run-time check that exactly one of them is `True`. For this enabled contract case, there is another run-time check when returning from the subprogram that the corresponding consequence evaluates to `True`.

When a subprogram is analyzed with GNATprove, it checks that there is always exactly one contract case enabled, and that the consequence of the contract case enabled cannot fail. If the subprogram also has a precondition, GNATprove performs these checks only for inputs that satisfy the precondition, otherwise for all inputs.

In the simple example presented above, there are various ways to express an equivalent postcondition, in particular using *Conditional Expressions*:

```
procedure Add_To_Total (Incr : in Integer) with
  Post => (if Total'Old + Incr < Threshold  then
             Total = Total'Old + Incr
           else
             Total = Threshold);

procedure Add_To_Total (Incr : in Integer) with
  Post => Total = (if Total'Old + Incr < Threshold then Total'Old + Incr else Threshold);

procedure Add_To_Total (Incr : in Integer) with
  Post => Total = Integer'Min (Total'Old + Incr, Threshold);
```

In general, an equivalent postcondition may be cumbersome to write and less readable. Contract cases also provide a way to automatically verify that the input space is partitioned in the specified cases, which may not be obvious with a single expression in a postcondition when there are many cases.

The guard of the last case may be `others`, to denote all cases not captured by previous contract cases. For example, the contract of `Add_To_Total` may be written:

```
procedure Add_To_Total (Incr : in Integer) with
  Contract_Cases => (Total + Incr < Threshold => Total = Total'Old + Incr,
                     others                   => Total = Threshold);
```

When `others` is used as a guard, there is no need for verification (both at run-time and using GNATprove) that the set of contract cases covers all possible inputs. Only disjointness of contract cases is checked in that case.

5.2.4 Data Dependencies

[SPARK]

The data dependencies of a subprogram specify the global data that a subprogram is allowed to read and write. Together with the parameters, they completely specify the inputs and outputs of a subprogram. Like parameters, the global variables mentioned in data dependencies have a mode: `Input` for inputs, `Output` for outputs and `In_Out` for global variables that are both inputs and outputs. A last mode of `Proof_In` is defined for inputs that are only read in contracts and assertions. For example, data dependencies can be specified for procedure `Add_To_Total` which increments global counter `Total` as follows:

```
procedure Add_To_Total (Incr : in Integer) with
  Global => (In_Out => Total);
```

For protected subprograms, the protected object is considered as an implicit parameter of the subprogram:

- it is an implicit parameter of mode `in` of a protected function; and
- it is an implicit parameter of mode `in out` of a protected procedure or a protected entry.

Data dependencies have no impact on compilation and the run-time behavior of a program. When a subprogram is analyzed with GNATprove, it checks that the implementation of the subprogram:

- only reads global inputs mentioned in its data dependencies,
- only writes global outputs mentioned in its data dependencies, and
- always completely initializes global outputs that are not also inputs.

See *Data Initialization Policy* for more details on this analysis of GNATprove. During its analysis, GNATprove uses the specified data dependencies of callees to analyze callers, if present, otherwise a default data dependency contract is generated (see *Generation of Dependency Contracts*) for callees.

There are various benefits when specifying data dependencies on a subprogram, which gives various reasons for users to add such contracts:

- GNATprove verifies automatically that the subprogram implementation respects the specified accesses to global data.
- GNATprove uses the specified contract during flow analysis, to analyze the data and flow dependencies of the subprogram's callers, which may result in a more precise analysis (less false alarms) than with the generated data dependencies.
- GNATprove uses the specified contract during proof, to check absence of run-time errors and the functional contract of the subprogram's callers, which may also result in a more precise analysis (less false alarms) than with the generated data dependencies.

When data dependencies are specified on a subprogram, they should mention all global data read and written in the subprogram. When a subprogram has neither global inputs nor global outputs, it can be specified using the `null` data dependencies:

```
function Get (X : T) return Integer with
  Global => null;
```

When a subprogram has only global inputs but no global outputs, it can be specified either using the `Input` mode:

```
function Get_Sum return Integer with
  Global => (Input => (X, Y, Z));
```

or equivalently without any mode:

```
function Get_Sum return Integer with
  Global => (X, Y, Z);
```

Note the use of parentheses around a list of global inputs or outputs for a given mode.

Global data that is both read and written should be mentioned with the `In_Out` mode, and not as both input and output. For example, the following data dependencies on `Add_To_Total` are illegal and rejected by GNATprove:

```
procedure Add_To_Total (Incr : in Integer) with
  Global => (Input  => Total,
             Output => Total);  --  INCORRECT
```

Global data that is partially written in the subprogram should also be mentioned with the `In_Out` mode, and not as an output. See *Data Initialization Policy*.

5.2.5 Flow Dependencies

[SPARK]

The flow dependencies of a subprogram specify how its outputs (both output parameters and global outputs) depend on its inputs (both input parameters and global inputs). For example, flow dependencies can be specified for procedure `Add_To_Total` which increments global counter `Total` as follows:

```
procedure Add_To_Total (Incr : in Integer) with
  Depends => (Total => (Total, Incr));
```

The above flow dependencies can be read as "the output value of global variable `Total` depends on the input values of global variable `Total` and parameter `Incr`".

Outputs (both parameters and global variables) may have an implicit input part depending on their type:

- an unconstrained array `A` has implicit input bounds `A'First` and `A'Last`
- a discriminated record `R` has implicit input discriminants, for example `R.Discr`

Thus, an output array `A` and an output discriminated record `R` may appear in input position inside a flow-dependency contract, to denote the input value of the bounds (for the array) or the discriminants (for the record).

For protected subprograms, the protected object is considered as an implicit parameter of the subprogram which may be mentioned in the flow dependencies, under the name of the protected unit (type or object) being declared:

- as an implicit parameter of mode `in` of a protected function, it can be mentioned on the right-hand side of flow dependencies; and
- as an implicit parameter of mode `in out` of a protected procedure or a protected entry, it can be mentioned on both sides of flow dependencies.

Flow dependencies have no impact on compilation and the run-time behavior of a program. When a subprogram is analyzed with GNATprove, it checks that, in the implementation of the subprogram, outputs depend on inputs as specified in the flow dependencies. During its analysis, GNATprove uses the specified flow dependencies of callees to analyze callers, if present, otherwise a default flow dependency contract is generated for callees (see *Generation of Dependency Contracts*).

When flow dependencies are specified on a subprogram, they should mention all flows from inputs to outputs. In particular, the output value of a parameter or global variable that is partially written by a subprogram depends on its input value (see *Data Initialization Policy*).

When the output value of a parameter or global variable depends on its input value, the corresponding flow dependency can use the shorthand symbol + to denote that a variable's output value depends on the variable's input value plus any other input listed. For example, the flow dependencies of `Add_To_Total` above can be specified equivalently:

```
procedure Add_To_Total (Incr : in Integer) with
  Depends => (Total =>+ Incr);
```

When an output value depends on no input value, meaning that it is completely (re)initialized with constants that do not depend on variables, the corresponding flow dependency should use the `null` input list:

```
procedure Init_Total with
  Depends => (Total => null);
```

5.2.6 State Abstraction and Contracts

[SPARK]

The subprogram contracts mentioned so far always used directly global variables. In many cases, this is not possible because the global variables are defined in another unit and not directly visible (because they are defined in the private part of a package specification, or in a package implementation). The notion of abstract state in SPARK can be used in that case (see *State Abstraction*) to name in contracts global data that is not visible.

State Abstraction and Dependencies

Suppose the global variable `Total` incremented by procedure `Add_To_Total` is defined in the package implementation, and a procedure `Cash_Tickets` in a client package calls `Add_To_Total`. Package `Account` which defines `Total` can define an abstract state `State` that represents `Total`, as seen in *State Abstraction*, which allows using it in `Cash_Tickets`'s data and flow dependencies:

```
procedure Cash_Tickets (Tickets : Ticket_Array) with
  Global  => (Output => Account.State),
  Depends => (Account.State => Tickets);
```

As global variable `Total` is not visible from clients of unit `Account`, it is not visible either in the visible part of `Account`'s specification. Hence, externally visible subprograms in `Account` must also use abstract state `State` in their data and flow dependencies, for example:

```
procedure Init_Total with
  Global  => (Output => State),
  Depends => (State => null);

procedure Add_To_Total (Incr : in Integer) with
  Global  => (In_Out => State),
  Depends => (State =>+ Incr);
```

Then, the implementations of `Init_Total` and `Add_To_Total` can define refined data and flow dependencies introduced respectively by `Refined_Global` and `Refined_Depends`, which give the precise dependencies for these subprograms in terms of concrete variables:

```
procedure Init_Total with
  Refined_Global  => (Output => Total),
  Refined_Depends => (Total => null)
is
```

(continues on next page)

(continued from previous page)

```
begin
   Total := 0;
end Init_Total;

procedure Add_To_Total (Incr : in Integer) with
  Refined_Global  => (In_Out => Total),
  Refined_Depends => (Total =>+ Incr)
is
begin
   Total := Total + Incr;
end Add_To_Total;
```

Here, the refined dependencies are the same as the abstract ones where `State` has been replaced by `Total`, but that's not always the case, in particular when the abstract state is refined into multiple concrete variables (see *State Abstraction*). GNATprove checks that:

- each abstract global input has at least one of its constituents mentioned by the concrete global inputs
- each abstract global in_out has at least one of its constituents mentioned with mode input and one with mode output (or at least one constituent with mode in_out)
- each abstract global output has to have all its constituents mentioned by the concrete global outputs
- the concrete flow dependencies are a subset of the abstract flow dependencies

GNATprove uses the abstract contract (data and flow dependencies) of `Init_Total` and `Add_To_Total` when analyzing calls outside package `Account` and the more precise refined contract (refined data and flow dependencies) of `Init_Total` and `Add_To_Total` when analyzing calls inside package `Account`.

Refined dependencies can be specified on both subprograms and tasks for which data and/or flow dependencies that are specified include abstract states which are refined in the current unit.

State Abstraction and Functional Contracts

If global variables are not visible for data dependencies, they are not visible either for functional contracts. For example, in the case of procedure `Add_To_Total`, if global variable `Total` is not visible, we cannot express anymore the precondition and postcondition of `Add_To_Total` as in *Preconditions* and *Postconditions*. Instead, we define accessor functions to retrieve properties of the state that we need to express, and we use these in contracts. For example here:

```
function Get_Total return Integer;

procedure Add_To_Total (Incr : in Integer) with
  Pre  => Incr >= 0 and then Get_Total in 0 .. Integer'Last - Incr,
  Post => Get_Total = Get_Total'Old + Incr;
```

Function `Get_Total` may be defined either in the private part of package `Account` or in its implementation. It may take the form of a regular function or an expression function (see *Expression Functions*), for example:

```
Total : Integer;

function Get_Total return Integer is (Total);
```

Although no refined preconditions and postconditions are required on the implementation of `Add_To_Total`, it is possible to provide a refined postcondition introduced by `Refined_Post` in that case, which specifies a more precise functional behavior of the subprogram. For example, procedure `Add_To_Total` may also increment the value of a counter `Call_Count` at each call, which can be expressed in the refined postcondition:

```
procedure Add_To_Total (Incr : in Integer) with
  Refined_Post => Total = Total'Old + Incr and Call_Count = Call_Count'Old + 1
is
   ...
end Add_To_Total;
```

A refined postcondition can be given on a subprogram implementation even when the unit does not use state abstraction, and even when the default postcondition of `True` is used implicitly on the subprogram declaration.

GNATprove uses the abstract contract (precondition and postcondition) of `Add_To_Total` when analyzing calls outside package `Account` and the more precise refined contract (precondition and refined postcondition) of `Add_To_Total` when analyzing calls inside package `Account`.

5.2.7 Exceptional Contracts

[SPARK]

In SPARK, every procedure which might propagate an exception should be annotated with an exceptional contract. This contract, introduced by the `Exceptional_Cases` aspect, lists all the exceptions which might be propagated by a procedure, and associates them to an exceptional postcondition. This postcondition describes the effect of the procedure when the exception is raised. As an example, consider the procedure `Incr_All` below. It goes over an array to increment its elements. If an overflow would occur, the exception `Overflow` is raised and the traversal is stopped. The global variable `Index` is used to store the current index at this point. The exceptional contract of `Incr_All` states both that it might propagate `Overflow`, and that it will only do so if it finds an offending index, using the global variable `Index`. The fact that `Overflow` is necessarily raised when such an index exists follows from the regular postcondition of `Incr_All`:

```
procedure Exceptions with SPARK_Mode is

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

   Overflow : exception;
   Index    : Positive;

   procedure Incr_All (A : in out Nat_Array) with
     Post =>
       (for all I in A'Range => A'Old (I) /= Natural'Last
          and then A (I) = A'Old (I) + 1),
     Exceptional_Cases =>
       (Overflow => Index in A'Old'Range and then A'Old (Index) = Natural'Last);

   procedure Incr_All (A : in out Nat_Array) is
   begin
      for I in A'Range loop
         if A (I) = Natural'Last then
            Index := I;
            raise Overflow;
         end if;

         A (I) := A (I) + 1;
         pragma Loop_Invariant
           (for all J in A'First .. I => A'Loop_Entry (J) < Natural'Last);
         pragma Loop_Invariant
```

(continues on next page)

(continued from previous page)

```
27             (for all J in A'First .. I => A (J) = A'Loop_Entry (J) + 1);
28       end loop;
29    end Incr_All;
30
31    procedure Incr_All_Cond (A : in out Nat_Array; Success : out Boolean) with
32      Post => Success = (for all I in A'Range => A'Old (I) /= Natural'Last)
33        and then
34          (if Success then (for all I in A'Range => A (I) = A'Old (I) + 1));
35
36    procedure Incr_All_Cond (A : in out Nat_Array; Success : out Boolean) is
37    begin
38       Incr_All (A);
39       Success := True;
40    exception
41       when Overflow =>
42          A := (others => 0);
43          Success := False;
44    end Incr_All_Cond;
45
46    procedure Incr_All_With_Pre (A : in out Nat_Array) with
47      Pre  => (for all I in A'Range => A (I) /= Natural'Last),
48      Post =>
49        (for all I in A'Range => A'Old (I) /= Natural'Last
50           and then A (I) = A'Old (I) + 1);
51
52    procedure Incr_All_With_Pre (A : in out Nat_Array) is
53    begin
54       Incr_All (A);
55    end Incr_All_With_Pre;
56
57 begin
58    null;
59 end Exceptions;
```

GNATprove can successfully verify both `Incr_All` above and its two callers: the exception is handled inside `Incr_All_Cond` and the call to `Incr_All` never raises `Overflow` in `Incr_All_With_Pre`.

```
exceptions.adb:10:08: info: postcondition proved
exceptions.adb:10:40: info: index check proved
exceptions.adb:11:23: info: index check proved
exceptions.adb:11:35: info: index check proved
exceptions.adb:11:38: info: overflow check proved
exceptions.adb:13:21: info: exceptional case proved
exceptions.adb:13:58: info: index check proved
exceptions.adb:19:22: info: range check proved
exceptions.adb:23:25: info: overflow check proved
exceptions.adb:25:13: info: loop invariant initialization proved
exceptions.adb:25:13: info: loop invariant preservation proved
exceptions.adb:25:56: info: index check proved
exceptions.adb:27:13: info: loop invariant preservation proved
exceptions.adb:27:13: info: loop invariant initialization proved
exceptions.adb:27:45: info: index check proved
```

(continues on next page)

(continued from previous page)

```
exceptions.adb:27:64: info: index check proved
exceptions.adb:27:67: info: overflow check proved
exceptions.adb:31:29: info: initialization of "A" proved
exceptions.adb:31:51: info: initialization of "Success" proved
exceptions.adb:32:14: info: postcondition proved
exceptions.adb:32:56: info: index check proved
exceptions.adb:34:55: info: index check proved
exceptions.adb:34:67: info: index check proved
exceptions.adb:34:70: info: overflow check proved
exceptions.adb:38:07: info: only expected exception raised
exceptions.adb:42:12: info: length check proved
exceptions.adb:47:42: info: index check proved
exceptions.adb:49:08: info: postcondition proved
exceptions.adb:49:40: info: index check proved
exceptions.adb:50:23: info: index check proved
exceptions.adb:50:35: info: index check proved
exceptions.adb:50:38: info: overflow check proved
exceptions.adb:54:07: info: only expected exception raised
```

The *Data Initialization Policy* of SPARK is mostly enforced on exceptional exit of subprograms. All global outputs shall be initialized when an exception is propagated, like `Index` in the example above. It is the case too for parameters which are necessarily passed by reference (tagged types, aliased parameters...). Other parameters, either necessarily passed by copy or for which the parameter passing mode is unspecified by the language, do not need to be initialized. As a result, after a call which has propagated an exception:

- output parameters necessarily passed by reference are considered to have been initialized or updated by the call as on a normal return,
- output parameters necessarily passed by copy are preserved, and
- output parameters for which the parameter passing mode is unspecified by the language are considered to not be initialized anymore; they should not be read after the call.

As an example, the parameter `A` of `Incr_All` is a composite type containing only subcomponents of a by-copy type (a scalar). As per the Ada reference manual, its parameter passing mode is unspecified. It means that its value will be unspecified if the call to `Incr_All` raises an exception, as can be seen on `Incr_All_Bad_Init`:

```
1  procedure Exceptions_Bad_Init with SPARK_Mode is
2
3     type Nat_Array is array (Positive range <>) of Natural;
4
5     Overflow : exception;
6     Index    : Positive;
7
8     procedure Incr_All (A : in out Nat_Array) with
9       Import,
10      Global => (In_Out => Index),
11      Always_Terminates,
12      Post =>
13        (for all I in A'Range => A'Old (I) /= Natural'Last
14           and then A (I) = A'Old (I) + 1),
15      Exceptional_Cases =>
16        (Overflow => Index in A'Old'Range and then A'Old (Index) = Natural'Last);
17
```

(continues on next page)

(continued from previous page)

```
   procedure Incr_All_Bad_Init
     (A       : in out Nat_Array;
      Success : out Boolean;
      N       : out Natural)
   with
     Post => Success = (for all I in A'Range => A'Old (I) /= Natural'Last)
       and then
         (if Success then (for all I in A'Range => A (I) = A'Old (I) + 1));

   procedure Incr_All_Bad_Init
     (A       : in out Nat_Array;
      Success : out Boolean;
      N       : out Natural)
   is
   begin
      Incr_All (A);
      Success := True;
      N := 0;
   exception
      when Overflow =>
         Success := False;
         N := A (Index);
   end Incr_All_Bad_Init;

begin
   null;
end Exceptions_Bad_Init;
```

```
exceptions_bad_init.adb:19:07: medium: "A" might not be initialized in "Incr_All_Bad_Init
↪"
   19 |     (A       : in out Nat_Array;
      |      ^ here
  reason for check: OUT parameter should be fully initialized on return
  possible fix: initialize "A" on all paths, make "A" an IN OUT parameter or annotate it␣
↪with aspect Relaxed_Initialization

exceptions_bad_init.adb:23:49: medium: "A" might not be initialized
   23 |     Post => Success = (for all I in A'Range => A'Old (I) /= Natural'Last)
      |                                                ^ here

exceptions_bad_init.adb:39:15: high: "A" is not initialized
   39 |         N := A (Index);
      |              ^ here
```

Note that, even though access types are passed by copy, `in` parameters of an access-to-variable part can be safely used after an exceptional exit as only the designated value can be modified.

To make it easier for the user, it is not allowed to mention parameters which are not necessarily passed by reference in an exceptional postcondition. An error is emitted if the exceptional postcondition of `Incr_All` is modified to mention `A`:

```
procedure Exceptions_Bad with SPARK_Mode is

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

   Overflow : exception;
   Index    : Positive;

   procedure Incr_All_Bad (A : in out Nat_Array) with
     Import,
     Post =>
       (for all I in A'Range => A'Old (I) /= Natural'Last
          and then A (I) = A'Old (I) + 1),
     Exceptional_Cases =>
       (Overflow => A'Old (Index) = Natural'Last
        and then (for all I in A'Range =>
                        A (I) = (if I < Index then A'Old (I) + 1 else A'Old (I))));

begin
   null;
end Exceptions_Bad;
```

```
exceptions_bad.adb:16:25: error: formal parameter of mode "in out" in consequence of␣
↪Exceptional_Cases is not allowed in SPARK
   16 |                        A (I) = (if I < Index then A'Old (I) + 1 else A'Old␣
↪(I))));
      |                        ^ here
  only parameters passed by reference are allowed
  violation of aspect SPARK_Mode at line 1
    1 |procedure Exceptions_Bad with SPARK_Mode is
      |                              ^ here
gnatprove: error during flow analysis and proof
```

The exceptional contract is allowed if the parameter `A` is marked as `aliased` however, as it is then necessarily passed by reference:

```
procedure Exceptions_Post with SPARK_Mode is

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

   Overflow : exception;
   Index    : Positive;

   procedure Incr_All_Post (A : aliased in out Nat_Array) with
     Post =>
       (for all I in A'Range => A'Old (I) /= Natural'Last
          and then A (I) = A'Old (I) + 1),
     Exceptional_Cases =>
       (Overflow => A'Old (Index) = Natural'Last
        and then (for all I in A'Range =>
                        A (I) = (if I < Index then A'Old (I) + 1 else A'Old (I))));

   procedure Incr_All_Post (A : aliased in out Nat_Array) is
```

(continues on next page)

(continued from previous page)

```
   begin
      for I in A'Range loop
         if A (I) = Natural'Last then
            Index := I;
            raise Overflow;
         end if;

         A (I) := A (I) + 1;
         pragma Loop_Invariant
           (for all J in A'First .. I => A'Loop_Entry (J) < Natural'Last);
         pragma Loop_Invariant
           (for all J in A'First .. I => A (J) = A'Loop_Entry (J) + 1);
      end loop;
   end Incr_All_Post;

begin
   null;
end Exceptions_Post;
```

```
exceptions_post.adb:10:08: info: postcondition proved
exceptions_post.adb:10:40: info: index check proved
exceptions_post.adb:11:23: info: index check proved
exceptions_post.adb:11:35: info: index check proved
exceptions_post.adb:11:38: info: overflow check proved
exceptions_post.adb:13:21: info: exceptional case proved
exceptions_post.adb:13:28: info: index check proved
exceptions_post.adb:15:28: info: index check proved
exceptions_post.adb:15:59: info: index check proved
exceptions_post.adb:15:62: info: overflow check proved
exceptions_post.adb:15:78: info: index check proved
exceptions_post.adb:21:22: info: range check proved
exceptions_post.adb:25:25: info: overflow check proved
exceptions_post.adb:27:13: info: loop invariant initialization proved
exceptions_post.adb:27:13: info: loop invariant preservation proved
exceptions_post.adb:27:56: info: index check proved
exceptions_post.adb:29:13: info: loop invariant preservation proved
exceptions_post.adb:29:13: info: loop invariant initialization proved
exceptions_post.adb:29:45: info: index check proved
exceptions_post.adb:29:64: info: index check proved
exceptions_post.adb:29:67: info: overflow check proved
```

Note that only exceptions which are raised explicitly in the code can be handled or propagated. For example, it would not be possible to remove the defensive code raising the exception in the loop and instead propagate `Constraint_Error` directly as in `Incr_All_CE`. Indeed, GNATprove always attempts to prove that runtime checks never fail. It complains on `Incr_All_CE` that the range check might fail, and flags the exceptional case as unreachable if proof warnings are enabled:

```
procedure Exceptions_RTE with SPARK_Mode is

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

   procedure Incr_All_CE (A : in out Nat_Array) with
```

(continues on next page)

(continued from previous page)

```
     Post =>
       (for all I in A'Range => A'Old (I) /= Natural'Last
          and then A (I) = A'Old (I) + 1),
     Exceptional_Cases => (Constraint_Error => True);

   procedure Incr_All_CE (A : in out Nat_Array) is
   begin
      for I in A'Range loop
         A (I) := A (I) + 1;
         pragma Loop_Invariant
           (for all J in A'First .. I => A'Loop_Entry (J) < Natural'Last);
         pragma Loop_Invariant
           (for all J in A'First .. I => A (J) = A'Loop_Entry (J) + 1);
      end loop;
   end Incr_All_CE;

begin
   null;
end Exceptions_RTE;
```

```
exceptions_rte.adb:9:48: warning: unreachable branch
    9 |     Exceptional_Cases => (Constraint_Error => True);
      |                                               ^~~~

exceptions_rte.adb:14:25: high: overflow check might fail, cannot prove upper bound for␣
↪A (I) + 1
   14 |         A (I) := A (I) + 1;
      |                  ~~~~~~^~~
  e.g. when A = (1 => Natural'Last)
        and A'First = 1
        and A'Last = 1
        and I = 1
  reason for check: result of addition must fit in a 32-bits machine integer
```

If the exception raised by a raise statement or procedure call is neither handled nor allowed by its exceptional contract (that is, it has no associated exceptional postcondition or this postcondition is statically False), then it is unexpected and GNATprove will make sure that it never occurs. More precisely, GNATprove treats raising an unexpected exception in the following way:

- in flow analysis, the program paths that lead to a statement raising an unexpected exception are not considered when checking the contract of the subprogram; and
- in proof, a check is generated for these statements, to prove that no such program point is reachable.

Occurences of `pragma Assert (X)` where `X` is an expression statically equivalent to `False` are treated in the same way.

As an example, consider the artificial subprogram `Check_OK` which raises an exception when parameter `OK` is `False`. The `Check_OK` procedure does not have an exceptional contract so the exception is unexpected:

```
package Abnormal_Terminations with
  SPARK_Mode
is
```

(continues on next page)

(continued from previous page)

```
4
5     G1, G2 : Integer := 0;
6
7     procedure Check_OK (OK : Boolean) with
8       Global => (Output => G1),
9       Pre    => OK;
10
11 end Abnormal_Terminations;
```

```
1  package body Abnormal_Terminations with
2    SPARK_Mode
3  is
4
5     procedure Check_OK (OK : Boolean) is
6     begin
7        if OK then
8           G1 := 1;
9        else
10          G2 := 1;
11          raise Program_Error;
12       end if;
13    end Check_OK;
14
15 end Abnormal_Terminations;
```

Note that, although `G2` is assigned in `Check_OK`, its assignment is directly followed by a `raise_statement`, so `G2` is never assigned on an execution of `Check_OK` that terminates normally. As a result, `G2` is not mentioned in the data dependencies of `Check_OK`. During flow analysis, GNATprove verifies that the body of `Check_OK` implements its declared data dependencies.

During proof, GNATprove generates a check that the `raise_statement` on line 11 is never reached. Here, it is proved thanks to the precondition of `Check_OK` which states that parameter `OK` should always be `True` on entry:

```
abnormal_terminations.adb:11:10: info: only expected exception raised
abnormal_terminations.ads:8:06: info: data dependencies proved
abnormal_terminations.ads:8:27: info: initialization of "G1" proved
```

Note: Raising an exception is a side-effect. As a consequence, the aspect `Exceptional_Cases` is not allowed on functions and exceptions raised by `raise_expressions` cannot be handled or propagated. GNATprove makes sure that they never occur.

5.2.8 Contracts for Termination

[SPARK]

By default, GNATprove verifies termination of all functions and automatically instantiated lemmas (procedures annotated with `Automatic_Instantiation`). For procedures or entries, GNATprove does not attempt to verify termination and is only concerned with their partial correctness. This means that GNATprove only verifies that the contract of each procedure or entry holds whenever it terminates (i.e., returns or raises an exception). It is still possible that the subprogram does not terminate in some or all cases (it can for example loop forever or exit the whole program using `GNAT.OS_Lib.OS_Exit`).

The `Always_Terminates` GNAT specific aspect allows users to request that GNATprove also verifies that a procedure or entry terminates. It is the case for example of the procedures `Ok_Terminating` and `Bad_Terminating` below. The aspect can also be used to provide a boolean condition like for the `Conditionally_Loop` procedure. If this condition is present, then the proof of termination is only attempted when the condition evaluates to True on subprogram entry. As an example, the procedure `Conditionally_Loop` might not terminate if its `Cond` parameter evaluates to True, and `Loop_Forever` never needs to terminate (but it might):

```
package Possibly_Nonterminating with
  SPARK_Mode
is

   procedure Loop_Forver with
     No_Return,
     Always_Terminates => False,
     Exceptional_Cases => (others => False);

   procedure Conditionally_Loop (Cond : Boolean) with
     Always_Terminates => not Cond;

   procedure OK_Terminating with
     Always_Terminates;

   procedure Bad_Terminating with
     Always_Terminates;

end Possibly_Nonterminating;
```

```
package body Possibly_Nonterminating with
  SPARK_Mode
is
   procedure Loop_Forver is
   begin
      loop
         null;
      end loop;
   end Loop_Forver;

   procedure Conditionally_Loop (Cond : Boolean) is
   begin
      if Cond then
         Loop_Forver;
      end if;
   end Conditionally_Loop;
```

(continues on next page)

(continued from previous page)

```
17
18    procedure Ok_Terminating is
19    begin
20       Conditionally_Loop (False);
21    end Ok_Terminating;
22
23    procedure Bad_Terminating is
24    begin
25       Conditionally_Loop (True);
26    end Bad_Terminating;
27
28 end Possibly_Nonterminating;
```

GNATprove verifies the termination of `OK_Terminating` and the conditional termination of `Conditionally_Loop` but a failed check is emitted for `Bad_Terminating` as it does not terminate.

```
possibly_nonterminating.adb:14:10: info: conditional call termination proved
possibly_nonterminating.adb:20:07: info: conditional call termination proved

possibly_nonterminating.adb:25:07: medium: call might not terminate
   25 |      Conditionally_Loop (True);
      |      ^~~~~~~~~~~~~~~~~~~~~~~~~
  reason for check: procedure "Bad_Terminating" has an Always_Terminates aspect
possibly_nonterminating.ads:13:14: info: aspect Always_Terminates on "OK_Terminating"␣
↪has been proved, subprogram will terminate
possibly_nonterminating.ads:16:14: info: aspect Always_Terminates on "Bad_Terminating"␣
↪has been proved, subprogram will terminate
```

A package can also be annotated with the `Always_Terminates` aspect. It does not apply to the elaboration of the package, which should always terminate in SPARK, but serves as a default for all the procedures located inside: unless specified otherwise, a procedure declared inside a package annotated with `Always_Terminates` should always terminate.

5.2.9 Subprogram Variant

[SPARK]

To ensure termination of recursive subprograms, it is possible to annotate them using the aspect `Subprogram_Variant`. This aspect provides a value which should *progress* in some sense between the beginning of the subprogram and each recursive call. The value is associated to a direction, which can be either `Increases` or `Decreases` for *numeric* variants, or `Structural` for *structural* variants.

Numeric variants can take a discrete value or, in the case of the direction `Decreases`, a big natural (see `Ada.Numerics.Big_Integers`). On every recursive call, a check is generated to ensure that the value progresses (decreases or increases) with respect to its value at the beginning of the subprogram. Since a discrete value is necessarily bounded by its Ada type, and a big natural is always greater than 0, it is enough to ensure that there will be no infinite chain of recursive calls.

In the following example, we can verify that the `Fibonacci` function terminates stating that its parameter `N` decreases at each recursive call:

```
1 with SPARK.Big_Integers; use SPARK.Big_Integers;
2
```

(continues on next page)

(continued from previous page)

```
package Recursive_Subprograms with SPARK_Mode is

   function Fibonacci (N : Big_Natural) return Big_Natural is
     (if N = 0 then 0
      elsif N = 1 then 1
      else Fibonacci (N - 1) + Fibonacci (N - 2))
   with Subprogram_Variant => (Decreases => N);
end Recursive_Subprograms;
```

GNATprove generates one verification condition per recursive call to make sure that the value given for `N` is smaller than the value of `N` on entry of `Fibonacci`:

```
recursive_subprograms.ads:5:13: info: implicit aspect Always_Terminates on "Fibonacci"␣
↪has been proved, subprogram will terminate
recursive_subprograms.ads:6:10: info: predicate check proved
recursive_subprograms.ads:6:14: info: predicate check proved
recursive_subprograms.ads:6:21: info: predicate check proved
recursive_subprograms.ads:7:13: info: predicate check proved
recursive_subprograms.ads:7:17: info: predicate check proved
recursive_subprograms.ads:7:24: info: predicate check proved
recursive_subprograms.ads:8:12: info: subprogram variant proved
recursive_subprograms.ads:8:12: info: predicate check proved
recursive_subprograms.ads:8:23: info: predicate check proved
recursive_subprograms.ads:8:25: info: predicate check proved
recursive_subprograms.ads:8:27: info: predicate check proved
recursive_subprograms.ads:8:30: info: predicate check proved
recursive_subprograms.ads:8:32: info: subprogram variant proved
recursive_subprograms.ads:8:32: info: predicate check proved
recursive_subprograms.ads:8:43: info: predicate check proved
recursive_subprograms.ads:8:45: info: predicate check proved
recursive_subprograms.ads:8:47: info: predicate check proved
recursive_subprograms.ads:9:45: info: range check proved
```

It is possible to give more than one numeric value in a subprogram variant. In this case, values are checked in the order in which they appear. If a value progresses (increases or decreases as specified) then it is enough to ensure the progression of the whole variant and the subsequent values are not considered. In the same way, if a value annotated with `Increases` actually decreases strictly (or the other way around) then the evaluation terminates and the verification of the variant fails. It is only if the values of all the preceding expressions have been found to be preserved that the subsequent value is considered. The function `Max` computes the index of the maximal value in a slice of an array. At each recursive call, it shifts the bound containing the smallest value:

```
package Recursive_Subprograms.Multiple with SPARK_Mode is
   type Nat_Array is array (Positive range <>) of Natural;

   function Max (A : Nat_Array; F, L : Positive) return Positive is
     (if F = L then F
      elsif A (F) > A (L) then Max (A, F, L - 1)
      else Max (A, F + 1, L))
   with Pre  => L in A'Range and F in A'Range and F <= L,
        Post => Max'Result in F .. L
         and then (for all I in F .. L => A (I) <= A (Max'Result)),
        Subprogram_Variant => (Increases => F, Decreases => L);
end Recursive_Subprograms.Multiple;
```

The variant specifies that, for each recursive call, either F increases, or F stays the same and L decreases. The order is not important here, as L and F are never modified at the same time. This variant can be verified by GNATprove.

```
recursive_subprograms-multiple.ads:4:13: info: implicit aspect Always_Terminates on "Max
↪" has been proved, subprogram will terminate
recursive_subprograms-multiple.ads:6:16: info: index check proved
recursive_subprograms-multiple.ads:6:24: info: index check proved
recursive_subprograms-multiple.ads:6:32: info: precondition proved
recursive_subprograms-multiple.ads:6:32: info: subprogram variant proved
recursive_subprograms-multiple.ads:6:45: info: range check proved
recursive_subprograms-multiple.ads:7:12: info: precondition proved
recursive_subprograms-multiple.ads:7:12: info: subprogram variant proved
recursive_subprograms-multiple.ads:7:22: info: overflow check proved
recursive_subprograms-multiple.ads:9:17: info: postcondition proved
recursive_subprograms-multiple.ads:10:46: info: index check proved
recursive_subprograms-multiple.ads:10:58: info: index check proved
```

Structural variants are generally used on recursive data-structures. The value associated to such a variant is necessarily a formal parameter of the subprogram. On every recursive call, a check is generated to ensure that the actual parameter denoted by the variant designates a strict subcomponent of the formal parameter denoted the variant at the beggining of the call. Since, due to the *Memory Ownership Policy* of SPARK, the data-structures cannot contain cycles, it is enough to ensure that there will be no infinite chain of recursive calls.

In the following example, we can verify that the `Length` function on singly-linked lists terminates stating that the structure designated by its parameter L structurally decreases between two recursive calls:

```
1  with Ada.Numerics.Big_Numbers.Big_Integers;
2  use  Ada.Numerics.Big_Numbers.Big_Integers;
3
4  package Recursive_Subprograms with SPARK_Mode is
5     type Cell;
6     type List is access Cell;
7     type Cell is record
8        Value : Integer;
9        Next  : List;
10    end record;
11
12    function Length (L : List) return Big_Natural is
13      (if L = null then Big_Natural'(0) else Length (L.Next) + 1)
14    with Subprogram_Variant => (Structural => L);
15 end Recursive_Subprograms;
```

The fact that the actual parameter for L on the recursive call designates a strict subcomponent of the structure designated by formal parameter L can be verified by GNATprove:

```
recursive_subprograms.ads:12:13: info: implicit aspect Always_Terminates on "Length" has␣
↪been proved, subprogram will terminate
recursive_subprograms.ads:13:37: info: predicate check proved
recursive_subprograms.ads:13:45: info: subprogram variant proved
recursive_subprograms.ads:13:45: info: predicate check proved
recursive_subprograms.ads:13:54: info: pointer dereference check proved
recursive_subprograms.ads:13:61: info: predicate check proved
recursive_subprograms.ads:13:63: info: predicate check proved
```

Structural variants are subjects to a number of restrictions. They cannot be combined with other variants, and are

checked according to a mostly syntactic criterion. When these restrictions cannot be followed, structural variants can be systematically replaced by a decreasing numeric variant providing the depth (or size) of the data structure, like function `Length` above. Strictly speaking, structural variants are only required to define the function returning that metric.

To verify the termination of mutually recursive subprograms, all subprograms should be annotated with *compatible* variants. We say that two variants are compatible if they have the same number of expressions, and matching values in the list have the same direction and the same base type. For example, the variants of `Is_Even` and `Is_Odd` are compatible, because both are of type `Integer` and both decrease.

```
1  package Recursive_Subprograms.Mutually with SPARK_Mode is
2     function Is_Odd (X : Natural) return Boolean with
3       Subprogram_Variant => (Decreases => X);
4     function Is_Even (X : Natural) return Boolean with
5       Subprogram_Variant => (Decreases => X);
6
7     function Is_Odd (X : Natural) return Boolean is
8       (if X = 0 then False else not Is_Even (X - 1));
9     function Is_Even (X : Natural) return Boolean is
10      (if X = 0 then True else not Is_Odd (X - 1));
11 end Recursive_Subprograms.Mutually;
```

GNATprove introduces a check to make sure that the variant progresses at each mutually recursive call.

```
recursive_subprograms-mutually.ads:2:13: info: implicit aspect Always_Terminates on "Is_
↪Odd" has been proved, subprogram will terminate
recursive_subprograms-mutually.ads:4:13: info: implicit aspect Always_Terminates on "Is_
↪Even" has been proved, subprogram will terminate
recursive_subprograms-mutually.ads:8:36: info: subprogram variant proved
recursive_subprograms-mutually.ads:8:47: info: range check proved
recursive_subprograms-mutually.ads:10:35: info: subprogram variant proved
recursive_subprograms-mutually.ads:10:45: info: range check proved
```

5.3 Package Contracts

Subprograms are not the only entities to bear contracts in SPARK. Package contracts are made up of various optional parts:

- The *state abstraction* specifies how global variables defined in the package are referred to abstractly where they are not visible. Aspect `Abstract_State` introduces abstract names and aspect `Refined_State` specifies the mapping between these names and global variables.
- The *package initialization* introduced by aspect `Initializes` specifies which global data (global variables and abstract state) defined in the package is initialized at package startup.
- The *package initial condition* introduced by aspect `Initial_Condition` specifies the properties holding after package startup.

Package startup (a.k.a. package *elaboration* in Ada RM) consists in the evaluation of all declarations in the package specification and implementation, in particular the evaluation of constant declarations and those variable declarations which contain an initialization expression, as well as the statements sometimes given at the end of a package body that are precisely executed at package startup.

5.3.1 State Abstraction

[SPARK]

The state abstraction of a package specifies a mapping between abstract names and concrete global variables defined in the package. State abstraction allows to define *Subprogram Contracts* at an abstract level that does not depend on a particular choice of implementation (see *State Abstraction and Contracts*), which is better both for maintenance (no need to change contracts) and scalability of analysis (contracts can be much smaller).

Basic State Abstraction

One abstract name may be mapped to more than one concrete variable, but no two abstract names can be mapped to the same concrete variable. When state abstraction is specified on a package, all non-visible global variables defined in the private part of the package specification and in its implementation should be mapped to abstract names. Thus, abstract names correspond to a partitioning of the non-visible global variables defined in the package.

The simplest use of state abstraction is to define a single abstract name (conventionally called `State`) to denote all non-visible global variables defined in the package. For example, consider package `Account` defining a global variable `Total` in its implementation, which is abstracted as `State`:

```
package Account with
  Abstract_State => State
is
   ...
end Account;

package body Account with
  Refined_State => (State => Total)
is
   Total : Integer;
   ...
end Account;
```

The aspect `Refined_State` maps each abstract name to a list of concrete global variables defined in the package. The list can be simply `null` to serve as placeholder for future definitions of global variables. Instead of concrete global variables, one can also use abstract names for the state of nested packages and private child packages, whose state is considered to be also defined in the parent package.

If global variable `Total` is defined in the private part of `Account`'s package specification, then the declaration of `Total` must use the special aspect `Part_Of` to declare its membership in abstract state `State`:

```
package Account with
  Abstract_State => State
is
   ...
private
  Total : Integer with Part_Of => State;
  ...
end Account;
```

This ensures that `Account`'s package specification can be checked by GNATprove even if its implementation is not in SPARK, or not available for analysis, or not yet developed.

A package with state abstraction must have a package body that states how abstract states are refined in aspect `Refined_State`, unless the package body is not in SPARK. If there is no other reason for the package to have a

body, then one should use `pragma Elaborate_Body` in the package spec to make it legal for the package to have a body on which to express state refinement.

In general, an abstract name corresponds to multiple global variables defined in the package. For example, we can imagine adding global variables to log values passed in argument to procedure `Add_To_Total`, that are also mapped to abstract name `State`:

```
package Account with
  Abstract_State => State
is
   ...
end Account;

package body Account with
  Refined_State => (State => (Total, Log, Log_Size))
is
   Total    : Integer;
   Log      : Integer_Array;
   Log_Size : Natural;
   ...
end Account;
```

We can also imagine defining different abstract names for the total and the log:

```
package Account with
  Abstract_State => (State, Internal_State)
is
   ...
end Account;

package body Account with
  Refined_State => (State => Total,
                    Internal_State => (Log, Log_Size))
is
   Total    : Integer;
   Log      : Integer_Array;
   Log_Size : Natural;
   ...
end Account;
```

The abstract names defined in a package are visible everywhere the package name itself is visible:

- in the scope where the package is declared, for a locally defined package
- in units that have a clause `with <package>;`
- in units that have a clause `limited with <package>;`

The last case allows subprograms in two packages to mutually reference the abstract state of the other package in their data and flow dependencies.

Special Cases of State Abstraction

Global constants with a statically known value are not part of a package's state. On the contrary, *constant with variable inputs* are constants whose value depends on the value of either a variable or a subprogram parameter. Since they participate in the flow of information between variables, constants with variable inputs are treated like variables: they are part of a package's state, and they must be listed in its state refinement whenever they are not visible. For example, constant `Total_Min` is not part of the state refinement of package `Account` below, while constant with variable inputs `Total_Max` is part of it:

```
package body Account with
  Refined_State => (State => (Total, Total_Max))
is
   Total     : Integer;
   Total_Min : constant Integer := 0;
   Total_Max : constant Integer := Compute_Total_Max(...);
   ...
end Account;
```

Global variables are not always the only constituents of a package's state. For example, if a package P contains a nested package N, then N's state is part of P's state. As a consequence, if N is hidden, then its state must be listed in P's refinement. For example, we can nest `Account` in the body of the `Account_Manager` package as follows:

```
package Account_Manager with
  Abstract_State => State
is
   ...
end Account_Manager;

package body Account_Manager with
  Refined_State => (State => Account.State)
is
   package Account with
     Abstract_State => State
   is
      ...
   end Account;
   ...
end Account_Manager;
```

State In The Private Part

Global variables and nested packages which themselves contain state may be declared in the private part of a package. For each such global variable and nested package state, it is mandatory to identify, using aspect `Part_Of`, the abstract state of the enclosing package of which it is a constituent:

```
package Account_Manager with
  Abstract_State => (Totals, Details)
is
   ...
private
   Total_Accounts : Integer with Part_Of => Totals;

   package Account with
```

(continues on next page)

(continued from previous page)

```
    Abstract_State => (State with Part_Of => Details)
  is
     Total : Integer with Part_Of => Totals;
     ...
  end Account;
  ...
end Account_Manager;
```

The purpose of using `Part_Of` is to enforce that each constituent of an abstract state is known at the declaration of the constituent (not having to look at the package body), which is useful for both code understanding and tool analysis (including compilation).

As the state of a private child package is logically part of its parent package, aspect `Part_Of` must also be specified in that case:

```
private package Account_Manager.Account with
  Abstract_State => (State with Part_Of => Details)
is
   Total : Integer with Part_Of => Totals;
   ...
end Account_Manager.Account;
```

Aspect `Part_Of` can also be specified on a generic package instantiation inside a private part, to specify that all the state (visible global variables and abstract states) of the package instantiation is a constituent of an abstract state of the enclosing package:

```
package Account_Manager with
  Abstract_State => (Totals, Details)
is
   ...
private
   package Account is new Generic_Account (Max_Total) with Part_Of => Details;
   ...
end Account_Manager;
```

5.3.2 Package Initialization

[SPARK]

The package initialization specifies which global data (global variables, constant with variable inputs, and abstract state) defined in the package is initialized at package startup. The corresponding global variables may either be initialized at declaration, or by the package body statements. Thus, package initialization can be seen as the output data dependencies of the package elaboration procedure generated by the compiler.

For example, we can specify that the state of package `Account` is initialized at package startup as follows:

```
package Account with
  Abstract_State => State,
  Initializes    => State
is
   ...
end Account;
```

Then, unless `Account`'s implementation is not in SPARK, it should initialize the corresponding global variable `Total` either at declaration:

```
package body Account with
  Refined_State => (State => Total)
is
   Total : Integer := 0;
   ...
end Account;
```

or in the package body statements:

```
package body Account with
  Refined_State => (State => Total)
is
   Total : Integer;
   ...
begin
   Total := 0;
end Account;
```

These initializations need not correspond to direct assignments, but may be performed in a call, for example here to procedure `Init_Total` as seen in *State Abstraction and Dependencies*. A mix of initializations at declaration and in package body statements is also possible.

Package initializations also serve as dependency contracts for global variables' initial values. That is, if the initial value of a global variable, state abstraction, or constant with variable inputs listed in a package initialization depends on the value of a variable defined outside the package, then this dependency must be listed in the package's initialization. For example, we can initialize `Total` by reading the value of an external variable:

```
package Account with
  Abstract_State => State,
  Initializes    => (State => External_Variable)
is
   ...
end Account;

package body Account with
  Refined_State => (State => Total)
is
   Total : Integer := External_Variable;
   ...
end Account;
```

5.3.3 Package Initial Condition

[SPARK]

The package initial condition specifies the properties holding after package startup. Thus, package initial condition can be seen as the postcondition of the package elaboration procedure generated by the compiler. For example, we can specify that the value of `Total` defined in package `Account`'s implementation is initially zero:

```
package Account with
  Abstract_State    => State,
```

(continues on next page)

(continued from previous page)

```
  Initial_Condition => Get_Total = 0
is
   function Get_Total return Integer;
   ...
end Account;
```

This is ensured either by initializing `Total` with value zero at declaration, or by assigning the value zero to `Total` in the package body statements, as seen in *Package Initialization*.

When the program is compiled with assertions (for example with switch `-gnata` in GNAT), the initial condition of a package is checked at run time after package startup. An exception is raised if the initial condition fails.

When a package is analyzed with GNATprove, it checks that the initial condition of a package cannot fail. GNATprove also analyzes the initial condition expression to ensure that it is free from run-time errors, like any other assertion.

5.3.4 Interfaces to the Physical World

[SPARK]

Volatile Variables

Most embedded programs interact with the physical world or other programs through so-called *volatile* variables, which are identified as volatile to protect them from the usual compiler optimizations. In SPARK, volatile variables are also analyzed specially, so that possible changes to their value from outside the program are taken into account, and so that changes to their value from inside the program are also interpreted correctly (in particular for checking flow dependencies).

For example, consider package `Volatile_Or_Not` which defines a volatile variable `V` and a non-volatile variable `N`, and procedure `Swap_Then_Zero` which starts by swapping the values of `V` and `N` before zeroing them out:

```
1  package Volatile_Or_Not with
2    SPARK_Mode,
3    Initializes => V
4  is
5     V : Integer with Volatile;
6     N : Integer;
7
8     procedure Swap_Then_Zero with
9       Global  => (In_Out => (N, V)),
10      Depends => (V => N, N => null, null => V);
11
12 end Volatile_Or_Not;
```

```
1  package body Volatile_Or_Not with
2    SPARK_Mode
3  is
4     procedure Swap_Then_Zero is
5        Tmp : constant Integer := V;
6     begin
7        --  Swap values of V and N
8        V := N;
9        N := Tmp;
```

(continues on next page)

(continued from previous page)

```
10       --  Zero out values of V and N
11       V := 0;
12       N := 0;
13    end Swap_Then_Zero;
14
15 end Volatile_Or_Not;
```

Compare the difference in contracts between volatile variable `V` and non-volatile variable `N`:

- The *Package Initialization* of package `Volatile_Or_Not` mentions `V` although this variable is not initialized at declaration or in the package body statements. This is because a volatile variable is assumed to be initialized.
- The *Flow Dependencies* of procedure `Swap_Then_Zero` are very different for `V` and `N`. If both variables were not volatile, the correct contract would state that both input values are not used with `null => (V, N)` and that both output values depend on no inputs with `(V, N) => null`. The difference lies with the special treatment of volatile variable `V`: as its value may be read at any time, the intermediate value `N` assigned to `V` on line 8 of `volatile_or_not.adb` needs to be mentioned in the flow dependencies for output `V`.

GNATprove checks that `Volatile_Or_Not` and `Swap_Then_Zero` implement their contract, and it issues a warning on the first assignment to `N`:

```
volatile_or_not.adb:9:09: warning: unused assignment
    9 |       N := Tmp;
      |       ~~^~~~~
```

This warning points to a real issue, as the intermediate value of `N` is not used before `N` is zeroed out on line 12. But note that no warning is issued on the similar first assignment to `V`, because the intermediate value of `V` may be read outside the program before `V` is zeroed out on line 11.

Note that in real code, the memory address of the volatile variable is set through aspect `Address` or the corresponding representation clause, so that it can be read or written outside the program.

Properties of Volatile Variables

Not all volatile variables are read and written outside the program, sometimes they are only read or only written outside the program. For example, the log introduced in *State Abstraction* could be implemented as an output port for the program logging the information, and as an input port for the program performing the logging. Two aspects are defined in SPARK to distinguish these different properties of volatile variables:

- Aspect `Async_Writers` indicates that the value of the variable may be changed at any time (asynchronously) by hardware or software outside the program.
- Aspect `Async_Readers` indicates that the value of the variable may be read at any time (asynchronously) by hardware or software outside the program.

Aspect `Async_Writers` has an effect on GNATprove's proof: two successive reads of such a variable may return different results. Aspect `Async_Readers` has an effect on GNATprove's flow analysis: an assignment to such a variable always has a potential effect, even if the value is never read in the program, since an external reader might actually read the value assigned.

These aspects are well suited to model respectively a sensor and a display, but not an input stream or an actuator, for which the act of reading or writing has an effect that should be reflected in the flow dependencies. Two more aspects are defined in SPARK to further refine the previous properties of volatile variables:

- Aspect `Effective_Reads` indicates that reading the value of the variable has an effect (for example, removing a value from an input stream). It can only be specified on a variable that also has `Async_Writers` set.

- Aspect `Effective_Writes` indicates that writing the value of the variable has an effect (for example, sending a command to an actuator). It can only be specified on a variable that also has `Async_Readers` set.

Both aspects `Effective_Reads` and `Effective_Writes` have an effect on GNATprove's flow analysis: reading the former or writing the latter is modelled as having an effect on the value of the variable, which needs to be reflected in flow dependencies. Because reading a variable with `Effective_Reads` set has an effect on its value, such a variable cannot be only a subprogram input, it must be also an output.

For example, the program writing in a log each value passed as argument to procedure `Add_To_Total` may model the output port `Log_Out` as a volatile variable with `Async_Readers` and `Effective_Writes` set:

```
package Logging_Out with
  SPARK_Mode
is
   Total   : Integer;
   Log_Out : Integer with Volatile, Async_Readers, Effective_Writes;

   procedure Add_To_Total (Incr : in Integer) with
     Global  => (In_Out => Total, Output => Log_Out),
     Depends => (Total =>+ Incr, Log_Out => Incr);

end Logging_Out;
```

```
package body Logging_Out with
  SPARK_Mode
is
   procedure Add_To_Total (Incr : in Integer) is
   begin
      Total := Total + Incr;
      Log_Out := Incr;
   end Add_To_Total;

end Logging_Out;
```

while the logging program may model the input port `Log_In` as a volatile variable with `Async_Writers` and `Effective_Reads` set:

```
package Logging_In with
  SPARK_Mode
is
   Log_In : Integer with Volatile, Async_Writers, Effective_Reads;

   type Integer_Array is array (Positive range 1 .. 100) of Integer;
   Log      : Integer_Array;
   Log_Size : Natural;

   procedure Get with
     Global  => (In_Out => (Log, Log_Size, Log_In)),
     Depends => ((Log_Size, Log_In) =>+ null, Log =>+ (Log_Size, Log_In));

end Logging_In;
```

```
package body Logging_In with
  SPARK_Mode
```

(continues on next page)

(continued from previous page)

```
is
   procedure Get is
   begin
      Log_Size := Log_Size + 1;
      Log (Log_Size) := Log_In;
   end Get;

end Logging_In;
```

GNATprove checks the specified data and flow dependencies on both programs.

A volatile variable on which none of the four aspects `Async_Writers`, `Async_Readers`, `Effective_Reads` or `Effective_Writes` is set is assumed to have all four aspects set to `True`. A volatile variable on which some of the four aspects are set to `True` is assumed to have the remaining ones set to `False`. See SPARK RM 7.1.3 for details.

Properties of Volatile Types

The four aspects `Async_Writers`, `Async_Readers`, `Effective_Reads` and `Effective_Writes` can be specified for a volatile type as well as a volatile variable. The rules stated for variables apply also to types for deciding on the value of the four aspects, when none, one or more values are specified explicitly. These aspects can only be specified on a type declaration, not on a subtype declaration.

Thus the declaration:

```
type T is new Integer;
Log_In : T with Volatile, Async_Writers, Effective_Reads;
```

can be written equivalently:

```
type T is new Integer with Volatile, Async_Writers, Effective_Reads;
Log_In : T;
```

Initialization of Volatile Variables and Variables with Address Clauses

For volatile variables and imported variables with address clauses, GNATprove assumes that the object is initialized. The following code does not raise errors about access to uninitialized data, and is proved. It does, however, raise several warnings on the object `Y` with an address clause, as it is not a precisely supported address clause.

```
X : Natural with Volatile;

subtype Even is Natural with Predicate => Even mod 2 = 0;

procedure Volatile_Init is
   Y : Even with Address => System'To_Address (16#DEADBEAF#), Import;
   Tmp : Integer := X;
begin
   pragma Assert (Y mod 2 = 0);
end P;
```

External State Abstraction

Volatile variables may be part of *State Abstraction*, in which case the volatility of the abstract name must be specified by using aspect `External` on the abstract name, as follows:

```
package Account with
  Abstract_State => (State with External)
is
   ...
end Account;
```

An external state may represent both volatile variables and non-volatile ones, for example:

```
package body Account with
  Refined_State => (State => (Total, Log, Log_Size))
is
   Total    : Integer;
   Log      : Integer_Array with Volatile;
   Log_Size : Natural with Volatile;
   ...
end Account;
```

The different *Properties of Volatile Variables* may also be specified in the state abstraction, which is then used by GNATprove to refine the analysis. For example, the program writing in a log seen in the previous section can be rewritten to abstract global variables as follows:

```
1  package Logging_Out_Abstract with
2    SPARK_Mode,
3    Abstract_State => (State with External => (Async_Readers, Effective_Writes)),
4    Initializes => State
5  is
6     procedure Add_To_Total (Incr : in Integer) with
7       Global  => (In_Out => State),
8       Depends => (State =>+ Incr);
9
10 end Logging_Out_Abstract;
```

```
1  package body Logging_Out_Abstract with
2    SPARK_Mode,
3    Refined_State => (State => (Log_Out, Total))
4  is
5     Total   : Integer := 0;
6     Log_Out : Integer := 0 with Volatile, Async_Readers, Effective_Writes;
7
8     procedure Add_To_Total (Incr : in Integer) with
9       Refined_Global  => (In_Out => Total, Output => Log_Out),
10      Refined_Depends => (Total =>+ Incr, Log_Out => Incr)
11    is
12    begin
13       Total := Total + Incr;
14       Log_Out := Incr;
15    end Add_To_Total;
16
17 end Logging_Out_Abstract;
```

while the logging program seen in the previous section may be rewritten to abstract global variables as follows:

```
package Logging_In_Abstract with
  SPARK_Mode,
  Abstract_State => (State with External => (Async_Writers, Effective_Reads))
is
   procedure Get with
     Global  => (In_Out => State),
     Depends => (State =>+ null);

end Logging_In_Abstract;
```

```
package body Logging_In_Abstract with
  SPARK_Mode,
  Refined_State => (State => (Log_In, Log, Log_Size))
is
   Log_In : Integer with Volatile, Async_Writers, Effective_Reads;

   type Integer_Array is array (Positive range 1 .. 100) of Integer;
   Log      : Integer_Array := (others => 0);
   Log_Size : Natural := 0;

   procedure Get with
     Refined_Global  => (In_Out => (Log, Log_Size, Log_In)),
     Refined_Depends => ((Log_Size, Log_In) =>+ null, Log =>+ (Log_Size, Log_In))
   is
   begin
      Log_Size := Log_Size + 1;
      Log (Log_Size) := Log_In;
   end Get;

end Logging_In_Abstract;
```

GNATprove checks the specified data and flow dependencies on both programs.

An external abstract state on which none of the four aspects `Async_Writers`, `Async_Readers`, `Effective_Reads` or `Effective_Writes` is set is assumed to have all four aspects set to `True`. An external abstract state on which some of the four aspects are set to `True` is assumed to have the remaining ones set to `False`. See SPARK RM 7.1.2 for details.

5.4 Type Contracts

SPARK contains various features to constrain the values of a given type:

- A *scalar range* may be specified on a scalar type or subtype to bound its values.
- A *record discriminant* may be specified on a record type to distinguish between variants of the same record.
- A *predicate* introduced by aspect `Static_Predicate`, `Dynamic_Predicate`, `Ghost_Predicate` or `Predicate` may be specified on a type or subtype to express a property verified by objects of the (sub)type.
- A *type invariant* introduced by aspect `Type_Invariant` or `Invariant` may be specified on the completion of a private type to express a property that is only guaranteed outside of the type scope.
- A *default initial condition* introduced by aspect `Default_Initial_Condition` on a private type specifies the initialization status and possibly properties of the default initialization for a type.

5.4.1 Scalar Ranges

[Ada 83]

Scalar types (signed integer types, modulo types, fixed-point types, floating-point types) can be given a low bound and a high bound to specify that values of the type must remain within these bounds. For example, the global counter `Total` can never be negative, which can be expressed in its type:

```
Total : Integer range 0 .. Integer'Last;
```

Any attempt to assign a negative value to variable `Total` results in raising an exception at run time. During analysis, GNATprove checks that all values assigned to `Total` are positive or null. The anonymous subtype above can also be given an explicit name:

```
subtype Nat is Integer range 0 .. Integer'Last;
Total : Nat;
```

or we can use the equivalent standard subtype `Natural`:

```
Total : Natural;
```

or `Nat` can be defined as a derived type instead of a subtype:

```
type Nat is new Integer range 0 .. Integer'Last;
Total : Nat;
```

or as a new signed integer type:

```
type Nat is range 0 .. Integer'Last;
Total : Nat;
```

All the variants above result in the same range checks both at run-time and in GNATprove. GNATprove also uses the range information for proving properties about the program (for example, the absence of overflows in computations).

5.4.2 Record Discriminants

[Ada 83]

Record types can use discriminants to:

- define multiple variants and associate each component with a specific variant
- bound the size of array components

For example, the log introduced in *State Abstraction* could be implemented as a discriminated record with a discriminant `Kind` selecting between two variants of the record for logging either only the minimum and maximum entries or the last entries, and a discriminant `Capacity` specifying the maximum number of entries logged:

```
1  package Logging_Discr with
2    SPARK_Mode
3  is
4     type Log_Kind is (Min_Max_Values, Last_Values);
5     type Integer_Array is array (Positive range <>) of Integer;
6
7     type Log_Type (Kind : Log_Kind; Capacity : Natural) is record
8        case Kind is
```

(continues on next page)

(continued from previous page)

```
         when Min_Max_Values =>
            Min_Entry : Integer;
            Max_Entry : Integer;
         when Last_Values =>
            Log_Data : Integer_Array (1 .. Capacity);
            Log_Size : Natural;
      end case;
   end record;

   subtype Min_Max_Log is Log_Type (Min_Max_Values, 0);
   subtype Ten_Values_Log is Log_Type (Last_Values, 10);

   function Log_Size (Log : Log_Type) return Natural;

   function Last_Entry (Log : Log_Type) return Integer with
     Pre => Log.Kind = Last_Values and then Log.Log_Size in 1 .. Log.Capacity;

end Logging_Discr;
```

Subtypes of `Log_Type` can specify fixed values for `Kind` and `Capacity`, like in `Min_Max_Log` and `Ten_Values_Log`. The discriminants `Kind` and `Capacity` are accessed like regular components, for example:

```
package body Logging_Discr with
  SPARK_Mode
is
   function Log_Size (Log : Log_Type) return Natural is
   begin
      case Log.Kind is
         when Min_Max_Values =>
            return 2;
         when Last_Values =>
            return Log.Log_Size;
      end case;
   end Log_Size;

   function Last_Entry (Log : Log_Type) return Integer is
   begin
      return Log.Log_Data (Log.Log_Size);
   end Last_Entry;

end Logging_Discr;
```

Any attempt to access a component not present in a variable (because it is of a different variant), or to access an array component outside its bounds, results in raising an exception at run time. During analysis, GNATprove checks that components accessed are present, and that array components are accessed within bounds:

```
logging_discr.adb:10:23: info: discriminant check proved
logging_discr.adb:16:17: info: discriminant check proved
logging_discr.adb:16:31: info: discriminant check proved
logging_discr.adb:16:31: info: index check proved
logging_discr.ads:13:13: info: range check proved
logging_discr.ads:18:37: info: range check proved
```

(continues on next page)

(continued from previous page)

```
logging_discr.ads:18:53: info: range check proved
logging_discr.ads:19:40: info: range check proved
logging_discr.ads:19:53: info: range check proved
logging_discr.ads:21:13: info: implicit aspect Always_Terminates on "Log_Size" has been␣
↪proved, subprogram will terminate
logging_discr.ads:23:13: info: implicit aspect Always_Terminates on "Last_Entry" has␣
↪been proved, subprogram will terminate
logging_discr.ads:24:48: info: discriminant check proved
```

5.4.3 Predicates

[Ada 2012]

Predicates can be used on any subtype to express a property verified by objects of the subtype at all times. Aspects `Static_Predicate` and `Dynamic_Predicate` are defined in Ada to associate a predicate with a subtype. Aspect `Dynamic_Predicate` allows to express more general predicates than aspect `Static_Predicate`, at the cost of restricting the use of variables of the subtype. The following table summarizes the main similarities and differences between both aspects:

Feature	`Static_Predicate`	`Dynamic_Predicate`
Applicable to scalar subtype	Yes	Yes
Applicable to array/record subtype	No	Yes
Allows simple comparisons with static values	Yes	Yes
Allows conjunctions/disjunctions	Yes	Yes
Allows function calls	No	Yes
Allows general Boolean properties	No	Yes
Can be used in membership test	Yes	Yes
Can be used as range in for-loop	Yes	No
Can be used as choice in case-statement	Yes	No
Can be used as prefix with attributes First, Last or Range	No	No
Can be used as index subtype in array	No	No

Aspect `Predicate` is specific to GNAT and can be used instead of `Static_Predicate` or `Dynamic_Predicate`. GNAT treats it as a `Static_Predicate` whenever possible and as a `Dynamic_Predicate` in the remaining cases, thus not restricting uses of variables of the subtype more than necessary.

Aspect `Ghost_Predicate` is also specific to GNAT and can be used instead of `Dynamic_Predicate` when the predicate expression needs to reference ghost entities or ghost attributes like `Initialized`. In that case, the subtype cannot be used as subtype_mark in a membership test.

Predicates are inherited by subtypes and derived types. If a subtype or a derived type inherits a predicate and defines its own predicate, both predicates are checked on values of the new (sub)type. Predicates are restricted in SPARK so that they cannot depend on variable input. In particular, a predicate cannot mention a global variable in SPARK, although it can mention a global constant.

GNATprove checks that all values assigned to a subtype with a predicate are allowed by its predicate (for all forms of predicate: `Predicate`, `Static_Predicate`, `Dynamic_Predicate` and `Ghost_Predicate`). GNATprove generates a predicate check even in cases where there is no corresponding run-time check, for example when assigning to a component of a record with a predicate. GNATprove also uses the predicate information for proving properties about the program.

Static Predicates

A static predicate allows specifying which values are allowed or forbidden in a scalar subtype, when this specification cannot be expressed with *Scalar Ranges* (because it has *holes*). For example, we can express that the global counter `Total` cannot be equal to `10` or `100` with the following static predicate:

```
subtype Count is Integer with
  Static_Predicate => Count /= 10 and Count /= 100;
Total : Count;
```

or equivalently:

```
subtype Count is Integer with
  Static_Predicate => Count in Integer'First .. 9 | 11 .. 99 | 101 .. Integer'Last;
Total : Count;
```

Uses of the name of the subtype `Count` in the predicate refer to variables of this subtype. Scalar ranges and static predicates can also be combined, and static predicates can be specified on subtypes, derived types and new signed integer types. For example, we may define `Count` as follows:

```
type Count is new Natural with
  Static_Predicate => Count /= 10 and Count /= 100;
```

Any attempt to assign a forbidden value to variable `Total` results in raising an exception at run time. During analysis, GNATprove checks that all values assigned to `Total` are allowed.

Similarly, we can express that values of subtype `Normal_Float` are the *normal* 32-bits floating-point values (thus excluding *subnormal* values), assuming here that `Float` is the 32-bits floating-point type on the target:

```
subtype Normal_Float is Float with
  Static_Predicate => Normal_Float <= -2.0**(-126) or Normal_Float = 0.0 or Normal_Float␣
↪>= 2.0**(-126);
```

Any attempt to assign a subnormal value to a variable of subtype `Normal_Float` results in raising an exception at run time. During analysis, GNATprove checks that only normal values are assigned to such variables.

Dynamic Predicates

A dynamic predicate allows specifying properties of scalar subtypes that cannot be expressed as static predicates. For example, we can express that values of subtype `Odd` and `Even` are distributed according to their name as follows:

```
subtype Odd is Natural with
  Dynamic_Predicate => Odd mod 2 = 1;

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

or that values of type `Prime` are prime numbers as follows:

```
type Prime is new Positive with
  Dynamic_Predicate => (for all Divisor in 2 .. Prime / 2 => Prime mod Divisor /= 0);
```

A dynamic predicate also allows specifying relations between components of a record. For example, we can express that the values paired together in a record are always distinct as follows:

```
type Distinct_Pair is record
  Val1, Val2 : Integer;
end record
  with Dynamic_Predicate => Distinct_Pair.Val1 /= Distinct_Pair.Val2;
```

or that a record stores pairs of values with their greatest common divisor as follows:

```
type Bundle_Values is record
  X, Y : Integer;
  GCD  : Natural;
end record
  with Dynamic_Predicate => Bundle_Values.GCD = Get_GCD (Bundle_Values.X, Bundle_Values.
↪Y);
```

or that the number of elements `Count` in a resizable table is always less than or equal to its maximal number of elements `Max` as follows:

```
type Resizable_Table (Max : Natural) is record
   Count : Natural;
   Data  : Data_Array(1 .. Max);
end record
  with Dynamic_Predicate => Resizable_Table.Count <= Resizable_Table.Max;
```

A dynamic predicate also allows specifying global properties over the content of an array. For example, we can express that elements of an array are stored in increasing order as follows:

```
type Ordered_Array is array (Index) of Integer
  with Dynamic_Predicate =>
    (for all I in Index => (if I < Index'Last then Ordered_Array(I) < Ordered_
↪Array(I+1)));
```

or that a special end marker is always present in the array as follows:

```
type Ended_Array is array (Index) of Integer
  with Dynamic_Predicate =>
    (for some I in Index => Ended_Array(I) = End_Marker);
```

Dynamic predicates are checked only at specific places at run time, as mandated by the Ada Reference Manual:

- when converting a value to the subtype with the predicate
- when returning from a call, for each in-out and out parameter passed by reference
- when declaring an object, except when there is no initialization expression and no subcomponent has a default expression

Thus, not all violations of the dynamic predicate are caught at run time. On the contrary, during analysis, GNATprove checks that initialized variables whose subtype has a predicate always contain a value allowed by the predicate.

5.4.4 Type Invariants

[Ada 2012]

In SPARK, type invariants can only be specified on completions of private types (and not directly on private type declarations). They express a property that is only guaranteed outside of the immediate scope of the type bearing the invariant. Aspect `Type_Invariant` is defined in Ada to associate an invariant with a type. Aspect `Invariant` is specific to GNAT and can be used instead of `Type_Invariant`.

GNATprove checks that, outside of the immediate scope of a type with an invariant, all values of this type are allowed by its invariant. In order to provide such a strong guarantee, GNATprove generates an invariant check even in cases where there is no corresponding run-time check, for example on global variables that are modified by a subprogram. GNATprove also uses the invariant information for proving properties about the program.

As an example, let us consider a stack, which can be queried for the maximum of the elements stored in it:

```
package P is

   type Stack is private;

   function Max (S : Stack) return Element;

private
```

In the implementation, an additional component is allocated for the maximum, which is kept up to date by the implementation of the stack. This information is a type invariant, which can be specified using a `Type_Invariant` aspect:

```
private

   type Stack is record
      Content : Element_Array := (others => 0);
      Size    : My_Length := 0;
      Max     : Element := 0;
   end record with
     Type_Invariant => Is_Valid (Stack);

   function Is_Valid (S : Stack) return Boolean is
     ((for all I in 1 .. S.Size => S.Content (I) <= S.Max)
      and (if S.Max > 0 then
               (for some I in 1 .. S.Size => S.Content (I) = S.Max)));

   function Max (S : Stack) return Element is (S.Max);

end P;
```

Like for subtype predicates, the name of the type can be used inside the invariant expression to refer to the current instance of the type. Here the subtype predicate of `Stack` expresses that the `Max` field of a valid stack is the maximum of the elements stored in the stack.

To make sure that the invariant holds for every value of type `Stack` outside of the package `P`, GNATprove introduces invariant checks in several places. First, at the type declaration, it will make sure that the invariant holds every time an object of type `Stack` is default initialized. Here, as the stack is empty by default and the default value of `Max` is 0, the check will succeed. It is also possible to forbid default initialization of objects of type `Stack` altogether by using a *Default Initial Condition* of `False`:

```
type Stack is private with Default_Initial_Condition => False;

type Stack is record
   Content : Element_Array;
   Size    : My_Length;
   Max     : Element;
end record with Type_Invariant => Is_Valid (Stack);
```

A check is also introduced to make sure the invariant holds for every global object declared in the scope of `Stack` after it has been initialized:

```
package body P is
   The_Stack : Stack := (Content => (others => 1),
                         Size    => 5,
                         Max     => 0);
begin
   The_Stack.Max := 1;
end P;
```

Here the global variable `The_Stack` is allowed to break its invariant during the elaboration of `P`. The invariant check will only be done at the end of the elaboration of `P`, and will succeed.

In the same way, variables and parameters of a subprogram are allowed to break their invariants in the subprogram body. Verification conditions are generated to ensure that no invariant breaking value can leak outside of `P`. More precisely, invariant checks on subprogram parameters are performed:

- when calling a subprogram visible outside of `P` from inside of `P`. Such a subprogram can be either declared in the visible part of `P` or in another unit,
- when returning from a subprogram declared in the visible part of `P`.

For example, let us consider the implementation of a procedure `Push` that pushes an element of top of a stack. It is declared in the visible part of the specification of `P`:

```
function Size (S : Stack) return My_Length;

procedure Push (S : in out Stack; E : Element) with
  Pre => Size (S) < My_Length'Last;

procedure Push_Zero (S : in out Stack) with
  Pre => Size (S) < My_Length'Last;
```

It is then implemented using an internal procedure `Push_Internal` declared in the body of `P`:

```
procedure Push_Internal (S : in out Stack; E : Element) with
  Pre  => S.Size < My_Length'Last,
  Post => S.Size = S.Size'Old + 1 and S.Content (S.Size) = E
  and S.Content (1 .. S.Size)'Old = S.Content (1 .. S.Size - 1)
  and S.Max = S.Max'Old
is
begin
   S.Size := S.Size + 1;
   S.Content (S.Size) := E;
end Push_Internal;
```

(continues on next page)

(continued from previous page)

```
procedure Push (S : in out Stack; E : Element) is
begin
   Push_Internal (S, E);
   if S.Max < E then
      S.Max := E;
   end if;
end Push;

procedure Push_Zero (S : in out Stack) is
begin
   Push (S, 0);
end Push_Zero;
```

On exit of `Push_Internal`, the invariant of `Stack` is broken. It is OK since `Push_Internal` is not visible from outside of `P`. Invariant checks are performed when exiting `Push` and when calling it from inside `Push_Zero`. They both succeed. Note that, because of invariant checks on parameters, it is not allowed in SPARK to call a function that is visible from outside `P` in the invariant of `Stack` otherwise this would lead to a recursive proof. In particular, it is not allowed to make `Is_Valid` visible in the public declarations of `P`. In the same way, the function `Size` cannot be used in the invariant of `Stack`. We also avoid using `Size` in the contract of `Push_Internal` as it would have enforced additional invariant checks on its parameter.

Checks are also performed for global variables accessed by subprograms inside `P`. Even if it is allowed to break the invariant of a global variable when inside the body of a subprogram declared in `P`, invariant checks are performed when calling and returning from every subprogram inside `P`. For example, if `Push` and `Push_Internal` are accessing directly the global stack `The_Stack` instead of taking it as a parameter, there will be a failed invariant check on exit of `Push_Internal`:

```
procedure Push_Internal (E : Element) with
  Pre  => The_Stack.Size < My_Length'Last
is
begin
   The_Stack.Size := The_Stack.Size + 1;
   The_Stack.Content (The_Stack.Size) := E;
end Push_Internal;

procedure Push (E : Element) is
begin
   Push_Internal (E);
   if The_Stack.Max < E then
      The_Stack.Max := E;
   end if;
end Push;
```

In this way, users will never have to use contracts to ensure that the invariant holds on global variable `The_Stack` through local subprogram calls.

5.4.5 Default Initial Condition

[SPARK]

Private types in a package define an encapsulation mechanism that prevents client units from accessing the implementation of the type. That boundary may also be used to specify properties that hold for default initialized values of that type in client units. For example, the log introduced in *State Abstraction* could be implemented as a private type with a default initial condition specifying that the size of the log is initially zero, where uses of the name of the private type `Log_Type` in the argument refer to variables of this type:

```
package Logging_Priv with
  SPARK_Mode
is
   Max_Count : constant := 100;

   type Log_Type is private with
     Default_Initial_Condition => Log_Size (Log_Type) = 0;

   function Log_Size (Log : Log_Type) return Natural;

   procedure Append_To_Log (Log : in out Log_Type; Incr : in Integer) with
     Pre => Log_Size (Log) < Max_Count;

private

   type Integer_Array is array (1 .. Max_Count) of Integer;

   type Log_Type is record
     Log_Data : Integer_Array;
     Log_Size : Natural := 0;
   end record;

   function Log_Size (Log : Log_Type) return Natural is (Log.Log_Size);

end Logging_Priv;
```

This may be useful to analyze with GNATprove client code that defines a variable of type `Log_Type` with default initialization, and then proceeds to append values to this log, as procedure `Append_To_Log`'s precondition requires that the log size is not maximal:

```
The_Log : Log_Type;
...
Append_To_Log (The_Log, X);
```

GNATprove's flow analysis also uses the presence of a default initial condition as an indication that default initialized variables of that type are considered as fully initialized. So the code snippet above would pass flow analysis without messages being issued on the read of `The_Log`. If the full definition of the private type is in SPARK, GNATprove also checks that the type is indeed fully default initialized, and if not issues a message like here:

```
logging_priv.ads:18:04: medium: type "Log_Type" is not fully initialized
   18>|   type Log_Type is record
  ... | ...
   21 |   end record;
```

If partial default initialization of the type is intended, in general for efficiency like here, then the corresponding message can be justified with pragma `Annotate`, see section *Justifying Check Messages*.

Aspect `Default_Initial_Condition` can also be specified without argument to only indicate that default initialized variables of that type are considered as fully initialized. This is equivalent to `Default_Initial_Condition => True`:

```
type Log_Type is private with
  Default_Initial_Condition;
```

The argument can also be `null` to specify that default initialized variables of that type are *not* considered as fully initialized:

```
type Log_Type is private with
  Default_Initial_Condition => null;
```

This is different from an argument of `False` which can be used to indicate that variables of that type should always be explicitly initialized (otherwise GNATprove will not be able to prove the condition `False` on the default initialization and will issue a message during proof).

In general, GNATprove generates checks for the default value of a type when a variable of this type is default initialized. This is not the case for private types, as the default value of a private type declared in a library unit is really the responsibility of the implementer of the library, not the user. If the private type has a known discriminant part, then default checks are done for any values of the discriminants.

If a private type has a `Default_Initial_Condition`, then this condition can act either as a precondition or as a postcondition of the default value computation. If the `Default_Initial_Condition` does not refer to the current type instance, or if it only refers to its discriminants, then it is considered to be a precondition: it is the user of the private type who is responsible for ensuring its validity. As such, the condition is assumed when checking the default value of the private type, and it is checked each time a variable of the type is default initialized. For example, in the following example, we must have `First < Last` to be allowed to safely default initialize our stack type:

```
type Stack (First, Last : Positive) is private with
  Default_Initial_Condition => First < Last;
```

GNATprove will take advantage of this information when checking the default value of `Stack` for run-time exceptions. For example, it will be able to ensure that the predicate will hold if `Stack` is defined as follows:

```
type Stack (First, Last : Positive) is record
   Content : Nat_Arr (First .. Last) := 0;
   Top     : Positive := First;
end record with
  Predicate => Top in Content'Range;
```

Otherwise, the `Default_Initial_Condition` is handled as a postcondition of the default value computation. It is checked once and for all when the definition of the type is analyzed.

5.5 Specification Features

SPARK contains many features for specifying the intended behavior of programs. Some of these features come from Ada 2012 (*Attribute Old* and *Expression Functions* for example). Other features are specific to SPARK (*Attribute Loop_Entry* and *Ghost Code* for example). In this section, we describe these features and their impact on execution and formal verification.

5.5.1 Aspect `Constant_After_Elaboration`

[SPARK]

Aspect `Constant_After_Elaboration` can be specified on a library level variable that has an initialization expression. When specified, the corresponding variable can only be changed during the elaboration of its enclosing package. SPARK ensures that users of the package do not change the variable. This feature can be particularly useful in tasking code since variables that are Constant_After_Elaboration are guaranteed to prevent unsynchronized modifications (see *Tasks and Data Races*).

```
package CAE is
   Var : Integer := 0 with
     Constant_After_Elaboration;

   --  The following is illegal because users of CAE could call Illegal
   --  and that would cause an update of Var after CAE has been
   --  elaborated.
   procedure Illegal with
      Global => (Output => Var);
end CAE;

package body CAE is
   procedure Illegal is
   begin
      Var := 10;
   end Illegal;

   --  The following subprogram is legal because it is declared inside
   --  the body of CAE and therefore it cannot be directly called
   --  from a user of CAE.
   procedure Legal is
   begin
      Var := Var + 2;
   end Legal;

begin
   --  The following statements are legal since they take place during
   --  the elaboration of CAE.
   Var := Var + 1;
   Legal;
end CAE;
```

5.5.2 Aspect No_Caching

[SPARK]

Aspect No_Caching can be specified for a volatile type or a volatile variable to indicate that this type or variable can be analyzed as non-volatile by GNATprove. This is typically used to hold the value of local variables guarding the access to some critical section of the code. To defend against fault injection attacks, a common practice is to duplicate the test guarding the critical section, and the variable is marked as volatile to prevent the compiler from optimizing out the duplicate tests. For example:

```
Cond : Boolean with Volatile, No_Caching := Some_Computation;

if not Cond then
    return;
end if;

if not Cond then
    return;
end if;

if Cond then
    -- here do some critical work
end if;
```

Without No_Caching, the volatile variable is assumed to be used for *Interfaces to the Physical World*, GNATprove analyses it specially and one cannot declare it inside a subprogram.

5.5.3 Aspect Relaxed_Initialization and Ghost Attribute Initialized

[SPARK]

Modes on parameters and data dependency contracts in SPARK have a stricter meaning than in Ada (see *Data Initialization Policy*). In general, this allows GNATprove to ensure correct initialization of data in a quick and scalable way through flow analysis, without the need for user-supplied annotations. However, in some cases, the initialization policy may be considered too constraining. In particular, it does not permit initializing composite objects by part through different subprograms, or leaving data uninitialized on return if an error occurred.

Aspect Relaxed_Initialization

To handle these cases, it is possible to relax the standard data initialization policy of SPARK using the Relaxed_Initialization aspect. This aspect can be used:

- on objects, to state that the object should not be subject to the initialization policy of SPARK,
- on types, so that it applies to every object or component of the type, or
- on subprograms, to annotate the parameters or result.

Here are some examples:

```
type My_Rec is record
   F, G : Positive;
end record;

G : My_Rec with Relaxed_Initialization;
```

(continues on next page)

(continued from previous page)

```
procedure Init_G_If_No_Errors (Error : out Boolean) with
   Global => (Output => G);
--  G is only initialized if the Error flag is False
```

In the snippet above, the aspect `Relaxed_Initialization` is used to annotate the object `G` so that SPARK will allow returning from `Init_G_If_No_Errors` with an uninitialized value in `G` in case of errors in the initialization routine.

On a subprogram, the `Relaxed_Initialization` aspect expects some parameters to specify to which objects it applies. For example, the parameter `X` of the procedures below is concerned by the aspect:

```
procedure Init_Only_F (X : out My_Rec) with
  Relaxed_Initialization => X;
--  Initialize the F component of X,
--  X.G should not be read after the call.

procedure Init_Only_G (X : in out My_Rec) with
  Relaxed_Initialization => X;
--  Initialize the G component of X,
--  X.F can be read after the call if it was already initialized.
```

The procedures `Init_Only_F` and `Init_Only_G` above differ only by the mode of parameter `X`. Just like for `Init_G_If_No_Errors`, the mode `out` in `Init_Only_F` does not mean that `X` should be entirely initialized by the call. Its purpose is mostly for data dependencies (see *Data Dependencies*). It states that the value on entry of the procedure call should not leak into the parts of the output value which are read after the call. To ensure that, GNATprove considers that `out` parameters may not be copied when entering a procedure call, and so, even for parameters which are in fact passed by reference.

To exempt the value returned by a function from the data initialization policy of SPARK, the result attribute can be specified as a parameter of the `Relaxed_Initialization` aspect, as in `Read_G` below. It is also possible to give several objects to the aspect using an aggregate notation:

```
procedure Copy (Source : My_Rec; Target : out My_Rec) with
  Relaxed_Initialization => (Source, Target);
--  Can copy a partially initialized record

function Read_G return My_Rec with
  Relaxed_Initialization => Read_G'Result;
--  The result of Read_G might not be initialized
```

Note: The `Relaxed_Initialization` aspect has no effect on subprogram parameters or function results of a scalar type with relaxed initialization. Indeed, the Ada semantics mandates a copy of scalars on entry and return of subprograms, which is considered to be an error if the object was not initialized.

Finally, if we want to exempt all objects of a type from the data initialization policy of SPARK, it is possible to specify the `Relaxed_Initialization` aspect on a type. This also allows to exempt a single component of a record, like in the following example:

```
type Content_Type is array (Positive range 1 .. 100) of Integer with
  Relaxed_Initialization;
type Stack is record
   Top     : Natural := 0;
   Content : Content_Type;
```

(continues on next page)

(continued from previous page)

```
end record
  with Predicate => Top in 0 .. 100;
--  Elements located after Top in Content do not need to be initialized
```

A stack is made of two components: an array `Content` storing the actual content of the stack, and the index `Top` of the topmost element currently allocated on the stack. If the stack is initialized, the `Top` component necessarily holds a meaningful value. However, because of the API of the stack, it is not possible to read a value stored above the `Top` index in `Content` without writing it first. For this reason, it is not necessary to initialize all elements of the stack at creation. To express that, we use in the type `Stack`, which itself is subject to the standard initialization policy, an array with the `Relaxed_Initialization` aspect for the `Content` field.

Note: The `Relaxed_Initialization` aspect is not allowed on subtypes, so a derived type is necessary to add the aspect to an existing type.

Ghost Attribute `Initialized`

As explained above, the standard data initialization policy does not apply to objects annotated with the `Relaxed_Initialization` aspect. As a result, it becomes necessary to annotate which parts of accessed objects are initialized on entry and exit of subprograms in contracts. This can be done using the `Initialized` ghost attribute. This attribute can be applied to (parts of) objects annotated with the `Relaxed_Initialization` aspect. If the object is completely initialized, except possibly for subcomponents of the object whose type is annotated with the `Relaxed_Initialization` aspect, this attribute evaluates to `True`.

Note: It is not true that the `Initialized` aspect necessarily evaluates to `False` on uninitialized data. This is to comply with execution, where some values may happen to be valid even if they have not been initialized. However, it is not possible to prove that the `Initialized` aspect evaluates to `True` if the object has not been entirely initialized.

As an example, let's add some contracts to the subprograms presented in the previous example to replace the comments. The case of `Init_G_If_No_Errors` is straightforward:

```
procedure Init_G_If_No_Errors (Error : out Boolean) with
  Post => (if not Error then G'Initialized);
```

It states that if no errors have occurred (`Error` is `False` on exit), `G` has been initialized by the call.

The postcondition of `Read_G` is a bit more complicated. We want to state that the function returns the value stored in `G`. However, we cannot use equality, as it would evaluate the components of both operands and fail if `G` is not entirely initialized. What we really want to say is that each component of the result of `Read_G` will be initialized if and only if the corresponding component in `G` is initialized, and then that the values of the components necessarily match in this case. To express that, we introduce safe accessors for the record components, which check whether the field is initialized before returning it. If the component is not initialized, they return `0` which is an invalid value since both components of `My_Rec` are of type `Positive`. This allows to encode both the initialization status and the value of the field in one go:

```
function Get_F (X : My_Rec) return Integer is
   (if X.F'Initialized then X.F else 0)
with Ghost,
  Relaxed_Initialization => X;

function Get_G (X : My_Rec) return Integer is
```

(continues on next page)

(continued from previous page)

```
   (if X.G'Initialized then X.G else 0)
with Ghost,
  Relaxed_Initialization => X;
```

Using these accessors, we can define an equality which can safely be called on uninitialized data, and use it in the postcondition of `Read_G`:

```
function Safe_Eq (X, Y : My_Rec) return Boolean is
  (Get_F (X) = Get_F (Y) and Get_G (X) = Get_G (Y))
with Ghost,
  Relaxed_Initialization => (X, Y);

function Read_G return My_Rec with
  Relaxed_Initialization => Read_G'Result,
  Post => Safe_Eq (Read_G'Result, G);
```

The same safe equality function can be used for the postcondition of `Copy`:

```
procedure Copy (Source : My_Rec; Target : out My_Rec) with
  Relaxed_Initialization => (Source, Target),
  Post => Safe_Eq (Source, Target);
```

Remain the procedures `Init_Only_F` and `Init_Only_G`. We reflect the asymmetry of their parameter modes in their postconditions:

```
procedure Init_Only_F (X : out My_Rec) with
  Relaxed_Initialization => X,
  Post => X.F'Initialized;

procedure Init_Only_G (X : in out My_Rec) with
  Relaxed_Initialization => X,
  Post => X.G'Initialized and Get_F (X) = Get_F (X)'Old;
```

The procedure `Init_Only_G` preserves the value of `X.F` whereas `Init_Only_F` does not preserve `X.G`. Note that a postcondition similar to the one of `Init_Only_G` would be proved on `Init_Only_F`, but it will be of no use as `out` parameters are considered to be havocked at the beginning of procedure calls, so `Get_G (X)'Old` wouldn't actually refer to the value of `G` before the call.

Finally, let's consider the type `Stack` defined above. We have annotated the array type used for its content with the `Relaxed_Initialization` aspect, so that we do not need to initialize all of its components at declaration. However, we still need to know that elements up to `Top` have been initialized to ensure that poping an element returns an initialized value. This can be stated by extending the subtype predicate of `Stack` in the following way:

```
type Stack is record
   Top     : Natural := 0;
   Content : Content_Type;
end record
  with Ghost_Predicate => Top in 0 .. 100
    and then (for all I in 1 .. Top => Content (I)'Initialized);
```

Since `Content_Type` is annotated with the `Relaxed_Initialization` aspect, references to the attribute `Initialized` on an object of type `Stack` will not consider the elements of `Content`, so `S'Initialized` can evaluate to True even if the stack `S` contains uninitialized elements.

Note: The predicate of type `Stack` is now introduced by aspect `Ghost_Predicate` to allow the use of ghost attribute `Initialized`.

Note: When the `Relaxed_Initialization` aspect is used, correct initialization is verified by proof (`--mode=all` or `--mode=silver`), and not flow analysis (`--mode=flow` or `--mode=bronze`).

It is possible to annotate an object with the `Relaxed_Initialization` aspect to use proof to verify its initialization. For example, it allows to workaround limitations in flow analysis with respect to initialization of arrays. However, if this initialization goes through a loop, using the `Initialized` attribute in a loop invariant might be required for proof to verify the program.

5.5.4 Aspect `Side_Effects`

[SPARK]

Unless stated otherwise, functions in SPARK cannot have side effects:

- A function must not have an `out` or `in out` parameter.
- A function must not write a global variable.
- A function must not raise exceptions.
- A function must always terminate.

The aspect `Side_Effects` can be used to indicate that a function may in fact have side effects, among the four possible side effects listed above. A *function with side effects* can be called only as the right-hand side of an assignment, as part of a list of statements where a procedure could be called:

```
function Increment_And_Return (X : in out Integer) return Integer
  with Side_Effects;

procedure Call is
  X : Integer := 5;
  Y : Integer;
begin
  Y := Increment_And_Return (X);
  --  The value of X is 6 here
end Call;
```

Note that a function with side effects could in general be converted into a procedure with an additional `out` parameter for the function's result. However, it can be more convenient to use a function with side effects when binding SPARK code with C code where functions have very often side effects.

5.5.5 Attribute Loop_Entry

[SPARK]

It is sometimes convenient to refer to the value of variables at loop entry. In many cases, the variable has not been modified between the subprogram entry and the start of the loop, so this value is the same as the value at subprogram entry. But *Attribute Old* cannot be used in that case. Instead, we can use attribute `Loop_Entry`. For example, we can express that after `J` iterations of the loop, the value of parameter array `X` at index `J` is equal to its value at loop entry plus one:

```
procedure Increment_Array (X : in out Integer_Array) is
begin
   for J in X'Range loop
      X(J) := X(J) + 1;
      pragma Assert (X(J) = X'Loop_Entry(J) + 1);
   end loop
end Increment_Array;
```

At run time, a copy of the variable `X` is made when entering the loop. This copy is then read when evaluating the expression `X'Loop_Entry`. No copy is made if the loop is never entered. Because it requires copying the value of `X`, the type of `X` cannot be limited.

Attribute `Loop_Entry` can only be used in top-level *Assertion Pragmas* inside a loop. It is mostly useful for expressing complex *Loop Invariants* which relate the value of a variable at a given iteration of the loop and its value at loop entry. For example, we can express that after `J` iterations of the loop, the value of parameter array `X` at all indexes already seen is equal to its value at loop entry plus one, and that its value at all indexes not yet seen is unchanged, using *Quantified Expressions*:

```
procedure Increment_Array (X : in out Integer_Array) is
begin
   for J in X'Range loop
      X(J) := X(J) + 1;
      pragma Loop_Invariant (for all K in X'First .. J => X(K) = X'Loop_Entry(K) + 1);
      pragma Loop_Invariant (for all K in J + 1 .. X'Last => X(K) = X'Loop_Entry(K));
   end loop;
end Increment_Array;
```

Attribute `Loop_Entry` may be indexed by the name of the loop to which it applies, which is useful to refer to the value of a variable on entry to an outter loop. When used without loop name, the attribute applies to the closest enclosing loop. For examples, `X'Loop_Entry` is the same as `X'Loop_Entry(Inner)` in the loop below, which is not the same as `X'Loop_Entry(Outter)` (although all three assertions are true):

```
procedure Increment_Matrix (X : in out Integer_Matrix) is
begin
   Outter: for J in X'Range(1) loop
      Inner: for K in X'Range(2) loop
         X(J,K) := X(J,K) + 1;
         pragma Assert (X(J,K) = X'Loop_Entry(J,K) + 1);
         pragma Assert (X(J,K) = X'Loop_Entry(Inner)(J,K) + 1);
         pragma Assert (X(J,K) = X'Loop_Entry(Outter)(J,K) + 1);
      end loop Inner;
   end loop Outter;
end Increment_Matrix;
```

By default, similar restrictions exist for the use of attribute `Loop_Entry` and the use of attribute `Old` *In a Potentially Unevaluated Expression*. The same solutions apply here, in particular the use of GNAT pragma

`Unevaluated_Use_Of_Old`.

5.5.6 Attribute `Old`

[Ada 2012]

In a Postcondition

Inside *Postconditions*, attribute `Old` refers to the values that expressions had at subprogram entry. For example, the postcondition of procedure `Increment` might specify that the value of parameter `X` upon returning from the procedure has been incremented:

```
procedure Increment (X : in out Integer) with
  Post => X = X'Old + 1;
```

At run time, a copy of the variable `X` is made when entering the subprogram. This copy is then read when evaluating the expression `X'Old` in the postcondition. Because it requires copying the value of `X`, the type of `X` cannot be limited.

Strictly speaking, attribute `Old` must apply to a *name* in Ada syntax, for example a variable, a component selection, a call, but not an addition like `X + Y`. For expressions that are not *names*, attribute `Old` can be applied to their qualified version, for example:

```
procedure Increment_One_Of (X, Y : in out Integer) with
  Post => X + Y = Integer'(X + Y)'Old + 1;
```

Because the compiler unconditionally creates a copy of the expression to which attribute `Old` is applied at subprogram entry, there is a risk that this feature might confuse users in more complex postconditions. Take the example of a procedure `Extract`, which copies the value of array `A` at index `J` into parameter `V`, and zeroes out this value in the array, but only if `J` is in the bounds of `A`:

```
procedure Extract (A : in out My_Array; J : Integer; V : out Value) with
  Post => (if J in A'Range then V = A(J)'Old);  --  INCORRECT
```

Clearly, the value of `A(J)` at subprogram entry is only meaningful if `J` is in the bounds of `A`. If the code above was allowed, then a copy of `A(J)` would be made on entry to subprogram `Extract`, even when `J` is out of bounds, which would raise a run-time error. To avoid this common pitfall, use of attribute `Old` in expressions that are potentially unevaluated (like the then-part in an if-expression, or the right argument of a shortcut boolean expression - See Ada RM 6.1.1) is restricted to plain variables: `A` is allowed, but not `A(J)`. The GNAT compiler issues the following error on the code above:

```
prefix of attribute "Old" that is potentially unevaluated must denote an entity
```

The correct way to specify the postcondition in the case above is to apply attribute `Old` to the entity prefix `A`:

```
procedure Extract (A : in out My_Array; J : Integer; V : out Value) with
  Post => (if J in A'Range then V = A'Old(J));
```

In Contract Cases

The rule for attribute `Old` inside *Contract Cases* is more permissive. Take for example the same contract as above for procedure `Extract`, expressed with contract cases:

```
procedure Extract (A : in out My_Array; J : Integer; V : out Value) with
  Contract_Cases => ((J in A'Range) => V = A(J)'Old,
                     others         => True);
```

Only the expressions used as prefixes of attribute `Old` in the *currently enabled case* are copied on entry to the subprogram. So if `Extract` is called with `J` out of the range of `A`, then the second case is enabled, so `A(J)` is not copied when entering procedure `Extract`. Hence, the above code is allowed.

It may still be the case that some contracts refer to the value of objects at subprogram entry inside potentially unevaluated expressions. For example, an incorrect variation of the above contract would be:

```
procedure Extract (A : in out My_Array; J : Integer; V : out Value) with
  Contract_Cases => (J >= A'First => (if J <= A'Last then V = A(J)'Old),  --  INCORRECT
                     others       => True);
```

For the same reason that such uses are forbidden by Ada RM inside postconditions, the SPARK RM forbids these uses inside contract cases (see SPARK RM 6.1.3(2)). The GNAT compiler issues the following error on the code above:

```
prefix of attribute "Old" that is potentially unevaluated must denote an entity
```

The correct way to specify the consequence expression in the case above is to apply attribute `Old` to the entity prefix `A`:

```
procedure Extract (A : in out My_Array; J : Integer; V : out Value) with
  Contract_Cases => (J >= A'First => (if J <= A'Last then V = A'Old(J)),
                     others       => True);
```

In a Potentially Unevaluated Expression

In some cases, the compiler issues the error discussed above (on attribute `Old` applied to a non-entity in a potentially unevaluated context) on an expression that can safely be evaluated on subprogram entry, for example:

```
procedure Extract (A : in out My_Array; J : Integer; V : out Value) with
  Post => (if J in A'Range then V = Get_If_In_Range(A,J)'Old);  --  ERROR
```

where function `Get_If_In_Range` returns the value `A(J)` when `J` is in the bounds of `A`, and a default value otherwise.

In that case, the solution is either to rewrite the postcondition using non-shortcut boolean operators, so that the expression is not *potentially evaluated* anymore, for example:

```
procedure Extract (A : in out My_Array; J : Integer; V : out Value) with
  Post => J not in A'Range or V = Get_If_In_Range(A,J)'Old;
```

or to rewrite the postcondition using an intermediate expression function, so that the expression is not *potentially evaluated* anymore, for example:

```
function Extract_Post (A : My_Array; J : Integer; V, Get_V : Value) return Boolean is
  (if J in A'Range then V = Get_V);

procedure Extract (A : in out My_Array; J : Integer; V : out Value) with
  Post => Extract_Post (A, J, V, Get_If_In_Range(A,J)'Old);
```

or to use the GNAT pragma `Unevaluated_Use_Of_Old` to allow such uses of attribute `Old` in potentially unevaluated expressions:

```
pragma Unevaluated_Use_Of_Old (Allow);

procedure Extract (A : in out My_Array; J : Integer; V : out Value) with
  Post => (if J in A'Range then V = Get_If_In_Range(A,J)'Old);
```

GNAT does not issue an error on the code above, and always evaluates the call to `Get_If_In_Range` on entry to procedure `Extract`, even if this value may not be used when executing the postcondition. Note that the formal verification tool GNATprove correctly generates all required checks to prove that this evaluation on subprogram entry does not fail a run-time check or a contract (like the precondition of `Get_If_In_Range` if any).

Pragma `Unevaluated_Use_Of_Old` applies to uses of attribute `Old` both inside postconditions and inside contract cases. See GNAT RM for a detailed description of this pragma.

5.5.7 Attribute `Result`

[Ada 2012]

Inside *Postconditions* of functions, attribute `Result` refers to the value returned by the function. For example, the postcondition of function `Increment` might specify that it returns the value of parameter `X` plus one:

```
function Increment (X : Integer) return Integer with
  Post => Increment'Result = X + 1;
```

Contrary to `Attribute Old`, attribute `Result` does not require copying the value, hence it can be applied to functions that return a limited type. Attribute `Result` can also be used inside consequence expressions in *Contract Cases*.

5.5.8 Aggregates

Aggregates are expressions, and as such can appear in assertions and contracts to specify the value of a composite type (record or array), without having to specify the value of each component of the object separately.

Record Aggregates

[Ada 83]

Since the first version, Ada has a compact syntax for expressing the value of a record type, optionally allowing to name the components. Given the following declaration of type `Point`:

```
type Point is record
   X, Y, Z : Float;
end record;
```

the value of the origin can be expressed with a named notation:

```
Origin : constant Point := (X => 0.0, Y => 0.0, Z => 0.0);
```

or with a positional notation, where the values for components are taken in the order in which they are declared in type `Point`, so the following is equivalent to the above named notation:

```
Origin : constant Point := (0.0, 0.0, 0.0);
```

With named notation, components can be given in any order:

```
Origin : constant Point := (Z => 0.0, Y => 0.0, X => 0.0);
```

Positional notation and named notation can be mixed, but, in that case, named associations should always follow positional associations, so positional notation will refer to the first components of the record, and named notation will refer to the last components of the record:

```
Origin : constant Point := (0.0, Y => 0.0, Z => 0.0);
Origin : constant Point := (0.0, 0.0, Z => 0.0);
```

Choices can be grouped with the bar symbol | to denote sets:

```
Origin : constant Point := (X | Y | Z => 0.0);
```

The choice `others` can be used with a value to refer to all other components, provided these components have the same type, and the `others` choice should come last:

```
Origin : constant Point := (X => 0.0, others => 0.0);
Origin : constant Point := (Z => 0.0, others => 0.0);
Origin : constant Point := (0.0, others => 0.0);  --  positional for X
Origin : constant Point := (others => 0.0);
```

The box notation <> can be used instead of an explicit value to denote the default value of the corresponding type:

```
Origin : constant Point := (X => <>, Y => 0.0, Z => <>);
```

In SPARK, this is only allowed if the types of the corresponding components have a default value, for example here:

```
type Zero_Init_Float is new Float with Default_Value => 0.0;

type Point is record
   X : Float := 0.0;
   Y : Float;
   Z : Zero_Init_Float;
end record;
```

Note that, when using box notation <> with an `others` choice, it is not required that these components have the same type.

Array Aggregates

[Ada 83]

Since the first version, Ada has the same compact syntax for expressing the value of an array type as for record types, optionally allowing to name the indexes. Given the following declaration of type `Point`:

```
type Dimension is (X, Y, Z);

type Point is array (Dimension) of Float;
```

the value of the origin can be expressed with a named notation:

```
Origin : constant Point := (X => 0.0, Y => 0.0, Z => 0.0);
```

or with a positional notation, where the values for components are taken in the order in which they are declared in type `Point`, so the following is equivalent to the above named notation:

```
Origin : constant Point := (0.0, 0.0, 0.0);
```

With the difference that named notation and positional notation cannot be mixed in an array aggregate, all other explanations presented for aggregates of record type `Point` in *Record Aggregates* are applicable to array aggregates here, so all the following declarations are valid:

```
Origin : constant Point := (Z => 0.0, Y => 0.0, X => 0.0);
Origin : constant Point := (X | Y | Z => 0.0);
Origin : constant Point := (X => 0.0, others => 0.0);
Origin : constant Point := (Z => 0.0, others => 0.0);
Origin : constant Point := (0.0, others => 0.0);  --  positional for X
Origin : constant Point := (others => 0.0);
```

while the use of box notation <> is only allowed in SPARK if array components have a default value, either through their type, or through aspect `Default_Component_Value` on the array type:

```
type Point is array (Dimension) of Float
  with Default_Component_Value => 0.0;
```

Note that in many cases, indexes take an integer value rather than an enumeration value:

```
type Dimension is range 1 .. 3;

type Point is array (Dimension) of Float;
```

In that case, choices will take an integer value too:

```
Origin : constant Point := (3 => 0.0, 2 => 0.0, 1 => 0.0);
Origin : constant Point := (1 | 2 | 3 => 0.0);
Origin : constant Point := (1 => 0.0, others => 0.0);
Origin : constant Point := (3 => 0.0, others => 0.0);
Origin : constant Point := (0.0, others => 0.0);  --  positional for 1
Origin : constant Point := (others => 0.0);
```

Note that one can also use X, Y and Z in place of literals 1, 2 and 3 with the prior definition of suitable named numbers:

```
X : constant := 1;
Y : constant := 2;
Z : constant := 3;
```

Note that allocators are allowed inside expressions, and that values in aggregates are evaluated for each corresponding choice, so it is possible to write the following without violating the *Memory Ownership Policy* of SPARK:

```
type Ptr is access Integer;
type Data is array (1 .. 10) of Ptr;

Database : Data := (others => new Integer'(0));
```

This would be also possible in a record aggregate, but it is more common in array aggregates.

Iterated Component Associations

[Ada 2022]

It is possible to have the value of an association depending on the choice, with the feature called *iterated component associations*. Here is how we can express that `Ident` is the identity mapping from values in `Index` to themselves:

```
type Index is range 1 .. 100;
type Mapping is array (Index) of Index;

Ident : constant Mapping := (for J in Index => J);
```

Such an iterated component association can appear next to other associations in an array aggregate using named notation. Here is how we can express that `Saturation` is the identity mapping between 10 and 90, and saturates outside of this range:

```
Saturation : constant Mapping :=
  (1 .. 10 => 10, for J in 11 .. 89 => J, 90 .. 100 => 90);
```

Initialization Using Array Aggregates

[Ada 83]

Both flow analysis and proof can be used in GNATprove to verify that data is correctly initialized before being read, following the *Data Initialization Policy* of SPARK. The decision to use one or the other is based on the presence or not of aspect `Relaxed_Initialization` (see *Aspect Relaxed_Initialization and Ghost Attribute Initialized*) on types and variables.

When using flow analysis to analyze the initialization of an array object (variable or component), false alarms may be emitted by GNATprove on code that initializes the array cell by cell, or groups of cells by groups of cells, even if the array ends up completely initialized. This is because flow analysis is not value dependent, so it cannot track the value of assigned array indexes. As a result, it cannot separate array cells in its analysis, hence it cannot deduce that such a sequence of partial initializations result in the array being completely initialized. For example, GNATprove issues false alarms on the code:

```
type Arr is array (1 .. 5) of Integer;
A : Arr;
...
A(1) := 1;
A(2) := 2;
A(3) := 3;
A(4) := 4;
A(5) := 5;
```

A better way to initialize an array is to use an aggregate (possibly with iterated component associations, if the value of the initialization element for a cell depends on the index of the cell). This makes it clear for both the human reviewer and for GNATprove that the array is completely initialized. For example, the code above can be rewritten as follows using an aggregate:

```
type Arr is array (1 .. 5) of Integer;
A : Arr;
...
A := (1, 2, 3, 4, 5);
```

or using an aggregate with an iterated component association:

```
type Arr is array (1 .. 5) of Integer;
A : Arr;
...
A := (for I in 1..5 => I);
```

In cases where initializing the array with an aggregate is not possible, the alternative is to mark the array object or its type as having relaxed initialization using aspect `Relaxed_Initialization` and to use proof to verify its correct initialization (see *Aspect Relaxed_Initialization and Ghost Attribute Initialized*). This should be reserved for cases where using an aggregate is not possible, as it requires more work for the user to express which parts of variables are initialized (in contracts and loop invariants typically), and it may be more difficult to prove.

Delta Aggregates

[Ada 2022]

It is quite common in *Postconditions* to relate the input and output values of parameters. While this can be as easy as `X = X'Old + 1` in the case of scalar parameters, it is more complex to express for array and record parameters. Delta aggregates are useful in that case, to denote the updated value of a composite variable. For example, we can express more clearly that procedure `Zero_Range` zeroes out the elements of its array parameter `X` between `From` and `To` by using a delta aggregate:

```
procedure Zero_Range (X : in out Integer_Array; From, To : Positive) with
  Post => X = (X'Old with delta From .. To => 0);
```

than with an equivalent postcondition using *Quantified Expressions* and *Conditional Expressions*:

```
procedure Zero_Range (X : in out Integer_Array; From, To : Positive) with
  Post => (for all J in X'Range =>
             (if J in From .. To then X(J) = 0 else X(J) = X'Old(J)));
```

Delta aggregates allow to specify a list of associations between indexes (for arrays) or components (for records) and values. Components can only be mentioned once, with the semantics that all values are evaluated before any update. Array indexes may be mentioned more than once, with the semantics that updates are applied in left-to-right order. For example, the postcondition of procedure `Swap` expresses that the values at indexes `J` and `K` in array `X` have been swapped:

```
procedure Swap (X : in out Integer_Array; J, K : Positive) with
  Post => X = (X'Old with delta J => X'Old(K), K => X'Old(J));
```

and the postcondition of procedure `Rotate_Clockwize_Z` expresses that the point `P` given in parameter has been rotated 90 degrees clockwise around the Z axis (thus component `Z` is preserved while components `X` and `Y` are modified):

```
procedure Rotate_Clockwize_Z (P : in out Point_3D) with
  Post => P = (P'Old with delta X => P.Y'Old, Y => - P.X'Old);
```

Similarly to their use in combination with attribute `Old` in postconditions, delta aggregates are useful in combination with *Attribute Loop_Entry* inside *Loop Invariants*. For example, we can express the property that, after iteration `J` in the main loop in procedure `Zero_Range`, the value of parameter array `X` at all indexes already seen is equal to zero:

```
procedure Zero_Range (X : in out Integer_Array; From, To : Positive) is
begin
   for J in From .. To loop
      X(J) := 0;
```

(continues on next page)

(continued from previous page)

```
      pragma Loop_Invariant (X = (X'Loop_Entry with delta From .. J => 0));
   end loop;
end Zero_Range;
```

Delta aggregates can also be used outside of assertions. They are particularly useful in expression functions. For example, the functionality in procedure `Rotate_Clockwize_Z` could be expressed equivalently as an expression function:

```
function Rotate_Clockwize_Z (P : Point_3D) return Point_3D is
  (P with delta X => P.Y, Y => - P.X);
```

Because it requires copying the value of `P`, the type of `P` cannot be limited.

Note: In SPARK versions up to SPARK 21, delta aggregates are not supported and an equivalent attribute named `Update` can be used instead.

Aspect Aggregate

[Ada 2022]

The `Aggregate` aspect has been introduced in Ada 2022. It allows providing subprograms that can be used to create aggregates of a particular container type. The required subprograms differ depending on the kind of aggregate being defined - positional, named, or indexed. Only positional and named container aggregates are allowed in SPARK. They require supplying an `Empty` function, to create the container, and an `Add` procedure to insert a new element (possibly associated to a key) in the container:

```
--  We can use positional aggregates for sets
type Set_Type is private
   with Aggregate => (Empty       => Empty_Set,
                      Add_Unnamed => Include);
function Empty_Set return Set_Type;
procedure Include (S : in out Set_Type; E : Element_Type);

--  and named aggregates for maps
type Map_Type is private
   with Aggregate =>  (Empty     => Empty_Map,
                       Add_Named => Add_To_Map);
function Empty_Map return Map_Type;
procedure Add_To_Map (M     : in out Map_Type;
                      Key   : Key_Type;
                      Value : Element_Type);
```

For execution, container aggregates are expanded into a call to the `Empty` function, followed by a sequence of calls to the `Add` procedure. However, for proof, this is not appropriate. Due to how VC generation works, instructions cannot be used to expand expressions occurring in annotations in particular. In addition, such an expansion would be inefficient in terms of provability, as it would introduce multiple intermediate values on which the provers need to reason.

To be able to use container aggregates in proof, additional annotations are necessary. They describe how the information supplied by the aggregate - the elements, the keys, their order, the number of elements... - affects the value of the container after the insertions. It works by supplying additional functions that should be used to describe the container. These functions and their intended usage are recognized using an *Annotation for Container Aggregates*.

Container aggregates follow the Ada 2022 syntax for homogeous aggregates. The values, or associations for named aggregates, are enclosed in square brackets. As an example, here are a few aggregates for functional and formal containers from the *SPARK Libraries*.

```
package Integer_Sets is new SPARK.Containers.Formal.Ordered_Sets (Integer);
S : Integer_Sets.Set := [1, 2, 3, 4, 12, 42];

package String_Lists is new
  SPARK.Containers.Formal.Unbounded_Doubly_Linked_Lists (String);
L : String_Lists.List := ["foo", "bar", "foobar"];

package Int_To_String_Maps is new
  SPARK.Containers.Functional.Maps (Integer, String);
M : Int_To_String_Maps.Map := [1 => "one", 2 => "two", 3 => "three"];
```

Note: So the handling is as precisely as possible, SPARK only supports aggregates with distinct values or keys for sets and maps.

5.5.9 Conditional Expressions

[Ada 2012]

A conditional expression is a way to express alternative possibilities in an expression. It is like the ternary conditional expression `cond ? expr1 : expr2` in C or Java, except more powerful. There are two kinds of conditional expressions in Ada:

- if-expressions are the counterpart of if-statements in expressions
- case-expressions are the counterpart of case-statements in expressions

For example, consider the variant of procedure `Add_To_Total` seen in *Contract Cases*, which saturates at a given threshold. Its postcondition can be expressed with an if-expression as follows:

```
procedure Add_To_Total (Incr : in Integer) with
  Post => (if Total'Old + Incr < Threshold  then
             Total = Total'Old + Incr
           else
             Total = Threshold);
```

Each branch of an if-expression (there may be one, two or more branches when `elsif` is used) can be seen as a logical implication, which explains why the above postcondition can also be written:

```
procedure Add_To_Total (Incr : in Integer) with
  Post => (if Total'Old + Incr < Threshold then Total = Total'Old + Incr) and
          (if Total'Old + Incr >= Threshold then Total = Threshold);
```

or equivalently (as the absence of `else` branch above is implicitly the same as `else True`):

```
procedure Add_To_Total (Incr : in Integer) with
  Post => (if Total'Old + Incr < Threshold then Total = Total'Old + Incr else True) and
          (if Total'Old + Incr >= Threshold then Total = Threshold else True);
```

If-expressions are not necessarily of boolean type, in which case they must have an `else` branch that gives the value of the expression for cases not covered in previous conditions (as there is no implicit `else True` in such a case). For example, here is a postcondition equivalent to the above, that uses an if-expression of `Integer` type:

```
procedure Add_To_Total (Incr : in Integer) with
  Post => Total = (if Total'Old + Incr < Threshold then Total'Old + Incr else Threshold);
```

Although case-expressions can be used to cover cases of any scalar type, they are mostly used with enumerations, and the compiler checks that all cases are disjoint and that together they cover all possible cases. For example, consider a variant of procedure `Add_To_Total` which takes an additional `Mode` global input of enumeration value `Single`, `Double`, `Negate` or `Ignore`, with the intuitive corresponding leverage effect on the addition. The postcondition of this variant can be expressed using a case-expression as follows:

```
procedure Add_To_Total (Incr : in Integer) with
  Post => (case Mode is
             when Single => Total = Total'Old + Incr,
             when Double => Total = Total'Old + 2 * Incr,
             when Ignore => Total = Total'Old,
             when Negate => Total = Total'Old - Incr);
```

Like if-expressions, case-expressions are not necessarily of boolean type. For example, here is a postcondition equivalent to the above, that uses a case-expression of `Integer` type:

```
procedure Add_To_Total (Incr : in Integer) with
  Post => Total = Total'Old + (case Mode is
                                 when Single => Incr,
                                 when Double => 2 * Incr,
                                 when Ignore => 0,
                                 when Negate => - Incr);
```

A last case of `others` can be used to denote all cases not covered by previous conditions. If-expressions and case-expressions should always be parenthesized.

5.5.10 Declare Expressions

[Ada 2022]

Declare expressions are used to factorize parts of an expression. They allow to declare constants and renamings which are local to the expression. A declare expression is made of two parts:

- A list of declarations of local constants and renamings
- An expression using the names introduced in these declarations.

To match the syntax of declare blocks, the first part is introduced by `declare` and the second by `begin`. The scope is delimited by enclosing parentheses, without `end` to close the scope.

As an example, we introduce a `Find_First_Zero` function which finds the index of the first occurrence of `0` in an array of integers and a procedure `Set_Range_To_Zero` which zeros out all elements located between the first and second occurrence of `0` in the array:

```
function Has_Zero (A : My_Array) return Boolean is
  (for some E of A => E = 0);

function Has_Two_Zeros (A : My_Array) return Boolean is
  (for some I in A'Range => A (I) = 0 and
```

(continues on next page)

(continued from previous page)

```
     (for some J in A'Range => A (J) = 0 and I /= J));

function Find_First_Zero (A : My_Array) return Natural with
  Pre  => Has_Zero (A),
  Post => Find_First_Zero'Result in A'Range
    and A (Find_First_Zero'Result) = 0
    and not Has_Zero (A (A'First .. Find_First_Zero'Result - 1));

procedure Set_Range_To_Zero (A : in out My_Array) with
  Pre  => Has_Two_Zeros (A),
  Post =>
     A = (A'Old with delta
            Find_First_Zero (A'Old) ..
              Find_First_Zero
                (A'Old (Find_First_Zero (A'Old) + 1 .. A'Last)) => 0);
```

In the contract of `Set_Range_To_Zero`, we use *Delta Aggregates* to state that elements of `A` located in the range between the first and the second occurrence of `0` in `A` have been set to `0` by the procedure. The second occurrence is found by calling `Find_First_Zero` on the slice of `A` starting just after the first occurrence of `0`.

To make the contract of `Set_Range_To_Zero` more readable, we can use a declare expression to introduce constants for the first and second occurrence of `0` in the array. The explicit names make it easier to understand what the bounds of the updated slice are supposed to be. It also avoids repeating the call to `Find_First_Zero` on `A` in the computation of the second bound:

```
procedure Set_Range_To_Zero (A : in out My_Array) with
  Pre  => Has_Two_Zeros (A),
  Post =>
    (declare
       Fst_Zero : constant Positive := Find_First_Zero (A'Old);
       Snd_Zero : constant Positive := Find_First_Zero
          (A'Old (Fst_Zero + 1 .. A'Last));
     begin
       A = (A'Old with delta Fst_Zero .. Snd_Zero => 0));
```

5.5.11 Expression Functions

[Ada 2012]

An expression function is a function whose implementation is given by a single expression. For example, the function `Increment` can be defined as an expression function as follows:

```
function Increment (X : Integer) return Integer is (X + 1);
```

For compilation and execution, this definition is equivalent to:

```
function Increment (X : Integer) return Integer is
begin
   return X + 1;
end Increment;
```

For GNATprove, this definition as expression function is equivalent to the same function body as above, plus a post-condition:

```
function Increment (X : Integer) return Integer with
  Post => Increment'Result = X + 1
is
begin
   return X + 1;
end Increment;
```

Thus, a user does not need in general to add a postcondition to an expression function, as the implicit postcondition generated by GNATprove is the most precise one. If a user adds a postcondition to an expression function, GNATprove uses this postcondition to analyze the function's callers as well as the most precise implicit postcondition.

On the contrary, it may be useful in general to add a precondition to an expression function, to constrain the contexts in which it can be called. For example, parameter `X` passed to function `Increment` should be less than the maximal integer value, otherwise an overflow would occur. We can specify this property in `Increment`'s precondition as follows:

```
function Increment (X : Integer) return Integer is (X + 1) with
  Pre => X < Integer'Last;
```

Note that the contract of an expression function follows its expression.

Expression functions can be defined in package declarations, hence they are well suited for factoring out common properties that are referred to in contracts. For example, consider the procedure `Increment_Array` that increments each element of its array parameter `X` by one. Its precondition can be expressed using expression functions as follows:

```
package Increment_Utils is

   function Not_Max (X : Integer) return Boolean is (X < Integer'Last);

   function None_Max (X : Integer_Array) return Boolean is
     (for all J in X'Range => Not_Max (X(J)));

   procedure Increment_Array (X : in out Integer_Array) with
     Pre => None_Max (X);

end Increment_Utils;
```

Expression functions can be defined over private types, and still be used in the contracts of publicly visible subprograms of the package, by declaring the function publicly and defining it in the private part. For example:

```
package Increment_Utils is

   type Integer_Array is private;

   function None_Max (X : Integer_Array) return Boolean;

   procedure Increment_Array (X : in out Integer_Array) with
     Pre => None_Max (X);

private

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

   function Not_Max (X : Integer) return Boolean is (X < Integer'Last);
```

(continues on next page)

(continued from previous page)

```
   function None_Max (X : Integer_Array) return Boolean is
     (for all J in X'Range => Not_Max (X(J)));

end Increment_Utils;
```

If an expression function is defined in a unit spec, GNATprove can use its implicit postcondition at every call. If an expression function is defined in a unit body, GNATprove can use its implicit postcondition at every call in the same unit, but not at calls inside other units. This is true even if the expression function is declared in the unit spec and defined in the unit body.

5.5.12 Ghost Code

[SPARK]

Sometimes, the variables and functions that are present in a program are not sufficient to specify intended properties and to verify these properties with GNATprove. In such a case, it is possible in SPARK to insert in the program additional code useful for specification and verification, specially identified with the aspect `Ghost` so that it can be discarded during compilation. So-called *ghost code* in SPARK are these parts of the code that are only meant for specification and verification, and have no effect on the functional behavior of the program.

Various kinds of ghost code are useful in different situations:

- *Ghost functions* are typically used to express properties used in contracts.
- *Global ghost variables* are typically used to keep track of the current state of a program, or to maintain a log of past events of some type. This information can then be referred to in contracts.
- *Local ghost variables* are typically used to hold intermediate values during computation, which can then be referred to in assertion pragmas like loop invariants.
- *Ghost types* are those types only useful for defining ghost variables.
- *Ghost procedures* can be used to factor out common treatments on ghost variables. Ghost procedures should not have non-ghost outputs, either output parameters or global outputs.
- *Ghost packages* provide a means to encapsulate all types and operations for a specific kind of ghost code.
- *Imported ghost subprograms* are used to provide placeholders for properties that are defined in a logical language, when using manual proof.
- *Ghost generic formal parameters* are used to pass on ghost entities (types, objects, subprograms, packages) as parameters in a generic instantiation.

When the program is compiled with assertions (for example with switch `-gnata` in GNAT), ghost code is executed like normal code. Ghost code can also be selectively enabled by setting pragma `Assertion_Policy` as follows:

```
pragma Assertion_Policy (Ghost => Check);
```

GNATprove checks that ghost code cannot have an effect on the behavior of the program. GNAT compiler also performs some of these checks, although not all of them. Apart from these checks, GNATprove treats ghost code like normal code during its analyses.

Ghost Functions

Ghost functions are useful to express properties only used in contracts, and to factor out common expressions used in contracts. For example, function `Get_Total` introduced in *State Abstraction and Functional Contracts* to retrieve the value of variable `Total` in the contract of `Add_To_Total` could be marked as a ghost function as follows:

```
function Get_Total return Integer with Ghost;
```

and still be used exactly as seen in *State Abstraction and Functional Contracts*:

```
procedure Add_To_Total (Incr : in Integer) with
  Pre  => Incr >= 0 and then Get_Total in 0 .. Integer'Last - Incr,
  Post => Get_Total = Get_Total'Old + Incr;
```

The definition of `Get_Total` would be also the same:

```
Total : Integer;

function Get_Total return Integer is (Total);
```

Although it is more common to define ghost functions as *Expression Functions*, a regular function might be used too:

```
function Get_Total return Integer is
begin
   return Total;
end Get_Total;
```

In that case, GNATprove uses only the contract of `Get_Total` (either user-specified or the default one) when analyzing its callers, like for a non-ghost regular function. (The same exception applies as for regular functions, when GNATprove can analyze a subprogram in the context of its callers, as described in *Contextual Analysis of Subprograms Without Contracts*.)

All functions which are only used in specification can be marked as ghost, but most don't need to. However, there are cases where marking a specification-only function as ghost really brings something. First, as ghost entities are not allowed to interfere with normal code, marking a function as ghost avoids having to break state abstraction for the purpose of specification. For example, marking `Get_Total` as ghost will prevent users of the package `Account` from accessing the value of `Total` from non-ghost code.

Then, in the usual context where ghost code is not kept in the final executable, the user is given more freedom to use in ghost code constructs that are less efficient than in normal code, which may be useful to express rich properties. For example, the ghost functions defined in the *Formal Containers Library* in the SPARK library typically copy the entire content of the argument container, which would not be acceptable for non-ghost functions.

Ghost Variables

Ghost variables are useful to keep track of local or global information during the computation, which can then be referred to in contracts or assertion pragmas.

Case 1: Keeping Intermediate Values

Local ghost variables are commonly used to keep intermediate values. For example, we can define a local ghost variable `Init_Total` to hold the initial value of variable `Total` in procedure `Add_To_Total`, which allows checking the relation between the initial and final values of `Total` in an assertion:

```
procedure Add_To_Total (Incr : in Integer) is
   Init_Total : Integer := Total with Ghost;
begin
   Total := Total + Incr;
   pragma Assert (Total = Init_Total + Incr);
end Add_To_Total;
```

Case 2: Keeping Memory of Previous State

Global ghost variables are commonly used to memorize the value of a previous state. For example, we can define a global ghost variable `Last_Incr` to hold the previous value passed in argument when calling procedure `Add_To_Total`, which allows checking in its precondition that the sequence of values passed in argument is non-decreasing:

```
Last_Incr : Integer := Integer'First with Ghost;

procedure Add_To_Total (Incr : in Integer) with
  Pre => Incr >= Last_Incr;

procedure Add_To_Total (Incr : in Integer) is
begin
   Total := Total + Incr;
   Last_Incr := Incr;
end Add_To_Total;
```

Case 3: Logging Previous Events

Going beyond the previous case, global ghost variables can be used to store a complete log of events. For example, we can define global ghost variables `Log` and `Log_Size` to hold the sequence of values passed in argument to procedure `Add_To_Total`, as in *State Abstraction*:

```
Log      : Integer_Array with Ghost;
Log_Size : Natural with Ghost;

procedure Add_To_Total (Incr : in Integer) with
  Post => Log_Size = Log_Size'Old + 1 and Log = (Log'Old with delta Log_Size => Incr);

procedure Add_To_Total (Incr : in Integer) is
begin
   Total := Total + Incr;
   Log_Size := Log_Size + 1;
   Log (Log_Size) := Incr;
end Add_To_Total;
```

The postcondition of `Add_To_Total` above expresses that `Log_Size` is incremented by one at each call, and that the current value of parameter `Incr` is appended to `Log` at each call (using *Attribute Old* and *Delta Aggregates*).

Case 4: Expressing Existentially Quantified Properties

In SPARK, universal quantification is only allowed in restricted cases (over integer ranges and over the content of a container). To express the existence of a particular object, it is sometimes easier to simply provide it. This can be done using a global ghost variable. This can be used in particular to split the specification of a complex procedure into smaller parts:

```
X_Interm : T with Ghost;

procedure Do_Two_Thing (X : in out T) with
  Post => First_Thing_Done (X'Old, X_Interm) and then
          Second_Thing_Done (X_Interm, X)
is
  X_Init : constant T := X with Ghost;
begin
  Do_Something (X);
  pragma Assert (First_Thing_Done (X_Init, X));
  X_Interm := X;

  Do_Something_Else (X);
  pragma Assert (Second_Thing_Done (X_Interm, X));
end Do_Two_Things;
```

More complicated uses can also be envisioned, up to constructing ghost data structures reflecting complex properties. For example, we can express that two arrays are a permutation of each other by constructing a permutation from one to the other:

```
Perm : Permutation with Ghost;

procedure Permutation_Sort (A : Nat_Array) with
  Post => A = Apply_Perm (Perm, A'Old)
is
begin
  --  Initalize Perm with the identity
  Perm := Identity_Perm;

  for Current in A'First .. A'Last - 1 loop
    Smallest := Index_Of_Minimum_Value (A, Current, A'Last);
    if Smallest /= Current then
      Swap (A, Current, Smallest);

      --  Update Perm each time we permute two elements in A
      Permute (Perm, Current, Smallest);
    end if;
   end loop;
 end Permutation_Sort;
```

Ghost Types

Ghost types can only be used to define ghost variables. For example, we can define ghost types `Log_Type` and `Log_Size_Type` that specialize the types `Integer_Array` and `Natural` for ghost variables:

```
subtype Log_Type is Integer_Array with Ghost;
subtype Log_Size_Type is Natural with Ghost;

Log      : Log_Type with Ghost;
Log_Size : Log_Size_Type with Ghost;
```

Ghost Procedures

Ghost procedures are useful to factor out common treatments on ghost variables. For example, we can define a ghost procedure `Append_To_Log` to append a value to the log as seen previously.

```
Log      : Integer_Array with Ghost;
Log_Size : Natural with Ghost;

procedure Append_To_Log (Incr : in Integer) with
  Ghost,
  Post => Log_Size = Log_Size'Old + 1 and Log = (Log'Old with delta Log_Size => Incr);

procedure Append_To_Log (Incr : in Integer) is
begin
   Log_Size := Log_Size + 1;
   Log (Log_Size) := Incr;
end Append_To_Log;
```

Then, this procedure can be called in `Add_To_Total` as follows:

```
procedure Add_To_Total (Incr : in Integer) is
begin
   Total := Total + Incr;
   Append_To_Log (Incr);
end Add_To_Total;
```

Ghost Packages

Ghost packages are useful to encapsulate all types and operations for a specific kind of ghost code. For example, we can define a ghost package `Logging` to deal with all logging operations on package `Account`:

```
package Logging with
  Ghost
is
   Log      : Integer_Array;
   Log_Size : Natural;

   procedure Append_To_Log (Incr : in Integer) with
     Post => Log_Size = Log_Size'Old + 1 and Log = (Log'Old with delta Log_Size => Incr);

   ...
```

(continues on next page)

(continued from previous page)

```
end Logging;
```

The implementation of package `Logging` is the same as if it was not a ghost package. In particular, a `Ghost` aspect is implicitly added to all declarations in `Logging`, so it is not necessary to specify it explicitly. `Logging` can be defined either as a local ghost package or as a separate unit. In the latter case, unit `Account` needs to reference unit `Logging` in a with-clause like for a non-ghost unit:

```
with Logging;

package Account is
   ...
end Account;
```

Imported Ghost Subprograms

When using manual proof (see *GNATprove and Manual Proof*), it may be more convenient to define some properties in the logical language of the prover rather than in SPARK. In that case, ghost functions might be marked as imported, so that no implementation is needed. For example, the ghost procedure `Append_To_Log` seen previously may be defined equivalently as a ghost imported function as follows:

```
function Append_To_Log (Log : Log_type; Incr : in Integer) return Log_Type with
  Ghost,
  Import;
```

where `Log_Type` is an Ada type used also as placeholder for a type in the logical language of the prover. To avoid any inconsistency between the interpretations of `Log_Type` in GNATprove and in the manual prover, it is preferable in such a case to mark the definition of `Log_Type` as not in SPARK, so that GNATprove does not make any assumptions on its content. This can be achieved by defining `Log_Type` as a private type and marking the private part of the enclosing package as not in SPARK:

```
package Logging with
  SPARK_Mode,
  Ghost
is
   type Log_Type is private;

   function Append_To_Log (Log : Log_type; Incr : in Integer) return Log_Type with
     Import;

   ...

private
   pragma SPARK_Mode (Off);

   type Log_Type is new Integer;  --  Any definition is fine here
end Logging;
```

A ghost imported subprogram cannot be executed, so calls to `Append_To_Log` above should not be enabled during compilation, otherwise a compilation error is issued. Note also that GNATprove will not attempt proving the contract of a ghost imported subprogram, as it does not have its body.

Ghost Generic Formal Parameters

Non-ghost generic units may depend on ghost entities for the specification and proof of their instantiations. In such a case, the ghost entities can be passed on as ghost generic formal parameters:

```
generic
   type T is private with Ghost;
   Var_Input  : T with Ghost;
   Var_Output : in out T with Ghost;
   with function F return T with Ghost;
   with procedure P (X : in out T) with Ghost;
   with package Pack is new Gen with Ghost;
package My_Generic with
  SPARK_Mode
is
   ...
```

At the point of instantiation of `My_Generic`, actual parameters for ghost generic formal parameters may be ghost, and in three cases, they must actually be ghost: the actual for a mutable ghost generic formal object, a ghost generic formal procedure, or a ghost generic formal package, must be ghost. Otherwise, writing to a ghost variable or calling a ghost procedure could have an effect on non-ghost variables.

```
package My_Instantiation is
  new My_Generic (T          => ... -- ghost or not
                  Var_Input  => ... -- ghost or not
                  Var_Output => ... -- must be ghost
                  F          => ... -- ghost or not
                  P          => ... -- must be ghost
                  Pack       => ... -- must be ghost
```

Ghost Models

When specifying a program, it is common to use a model, that is, an alternative, simpler view of a part of the program. As they are only used in annotations, models can be computed using ghost code.

Models of Control Flow

Global variables can be used to enforce properties over call cahains in the program. For example, we may want to express that `Total` cannot be incremented twice in a row without registering the transaction in between. This can be done by introducing a ghost global variable `Last_Transaction_Registered`, used to encode whether `Register_Transaction` was called since the last call to `Add_To_Total`:

```
Last_Transaction_Registered : Boolean := True with Ghost;

procedure Add_To_Total (Incr : Integer) with
  Pre  => Last_Transaction_Registered,
  Post => not Last_Transaction_Registered;

procedure Register_Transaction with
  Post => Last_Transaction_Registered;
```

The value of Last_Transaction_Registered should also be updated in the body of `Add_To_Total` and `Register_Transaction` to reflect their contracts:

```
procedure Add_To_Total (Incr : in Integer) is
begin
   Total := Total + Incr;
   Last_Transaction_Registered := False;
end Add_To_Total;
```

More generally, the expected control flow of a program can be modeled using an automaton. We can take as an example a mailbox containing only one message. The expected way `Receive` and `Send` should be interleaved can be expressed as a two state automaton. The mailbox can either be full, in which case `Receive` can be called but not `Send`, or it can be empty, in which case it is `Send` that can be called and not `Receive`. To express this property, we can define a ghost global variable of a ghost enumeration type to hold the state of the automaton:

```
type Mailbox_Status_Kind is (Empty, Full) with Ghost;
Mailbox_Status : Mailbox_Status_Kind := Empty with Ghost;

procedure Receive (X : out Message) with
  Pre  => Mailbox_Status = Full,
  Post => Mailbox_Status = Empty;

procedure Send (X : Message) with
  Pre  => Mailbox_Status = Empty,
  Post => Mailbox_Status = Full;
```

Like before, `Receive` and `Send` should update `Mailbox_Status` in their bodies. Note that all the transitions of the automaton need not be specified, only the part which are relevant to the properties we want to express.

If the program also has some regular state, an invariant can be used to link the value of this state to the value of the ghost state of the automaton. For example, in our mailbox, we may have a regular variable `Message_Content` holding the content of the current message, which is only known to be valid after a call to `Send`. We can introduce a ghost function linking the value of `Message_Content` to the value of `Mailbox_Status`, so that we can ensure that `Message_Content` is always valid when accessed from `Receive`:

```
function Invariant return Boolean is
  (if Mailbox_Status = Full then Valid (Message_Content))
with Ghost;

procedure Receive (X : out Message) with
  Pre  => Invariant and then Mailbox_Status = Full,
  Post => Invariant and then Mailbox_Status = Empty
      and then Valid (X)
is
  X := Message_Content;
end Receive;
```

Models of Data Structures

For specifying programs that use complex data structures (doubly-linked lists, maps...), it can be useful to supply a model for the data structure. A model is an alternative, simpler view of the data-structure which allows to write properties more easily. For example, a ring buffer, or a doubly-linked list, can be modeled using an array containing the elements from the buffer or the list in the right order. Typically, though simpler to reason with, the model is less efficient than the regular data structure. For example, inserting an element at the beginning of a doubly-linked list or at the beginning of a ring buffer can be done in constant time whereas inserting an element at the beginning of an array requires to slide all the elements to the right. As a result, models of data structures are usually supplied using ghost code. As an example, the package `Ring_Buffer` offers an implementation of a single instance ring buffer. A ghost variable `Buffer_Model` is used to write the specification of the `Enqueue` procedure:

```
package Ring_Buffer is
  function Get_Model return Nat_Array with Ghost;

  procedure Enqueue (E : Natural) with
    Post => Get_Model = E & Get_Model'Old (1 .. Max - 1);
private
  Buffer_Content : Nat_Array;
  Buffer_Top     : Natural;
  Buffer_Model   : Nat_Array with Ghost;

  function Get_Model return Nat_Array is (Buffer_Model);
end Ring_Buffer;
```

Then, just like for models of control flow, an invariant should be supplied to link the regular data structure to its model:

```
package Ring_Buffer is
  function Get_Model return Nat_Array with Ghost;
  function Invariant return Boolean with Ghost;

  procedure Enqueue (E : Natural) with
    Pre  => Invariant,
    Post => Invariant and then Get_Model = E & Get_Model'Old (1 .. Max - 1);
private
  Buffer_Content : Nat_Array;
  Buffer_Top     : Natural;
  Buffer_Model   : Nat_Array with Ghost;

  function Get_Model return Nat_Array is (Buffer_Model);
  function Invariant return Boolean is
    (Buffer_Model = Buffer_Content (Buffer_Top .. Max)
                  & Buffer_Content (1 .. Buffer_Top - 1));
end Ring_Buffer;
```

If a data structure type is defined, a ghost function can be provided to compute a model for objects of the data structure type, and the invariant can be stated as a postcondition of this function:

```
package Ring_Buffer is
  type Buffer_Type is private;
  subtype Model_Type is Nat_Array with Ghost;

  function Invariant (X : Buffer_Type; M : Model_Type) return Boolean with
    Ghost;
```

(continues on next page)

(continued from previous page)

```
  function Get_Model (X : Buffer_Type) return Model_Type with
    Ghost,
    Post => Invariant (X, Get_Model'Result);

  procedure Enqueue (X : in out Buffer_Type; E : Natural) with
    Post => Get_Model (X) = E & Get_Model (X)'Old (1 .. Max - 1);
private
  type Buffer_Type is record
    Content : Nat_Array;
    Top     : Natural;
  end record;
end Ring_Buffer;
```

More complex examples of models of data structure can be found in the *Formal Containers Library*.

Removal of Ghost Code

By default, GNAT completely discards ghost code during compilation, so that no ghost code is present in the object code or the executable. This ensures that, even if parts of the ghost could have side effects when executed (writing to variables, performing system calls, raising exceptions, etc.), by default the compiler ensures that it cannot have any effect on the behavior of the program.

This is also essential in domains submitted to certification where all instructions in the object code should be traceable to source code and requirements, and where testing should ensure coverage of the object code. As ghost code is not present in the object code, there is no additional cost for maintaining its traceability and ensuring its coverage by tests.

GNAT provides an easy means to check that no ignored ghost code is present in a given object code or executable, which relies on the property that, by definition, each ghost declaration or ghost statement mentions at least one ghost entity. GNAT prefixes all names of such ignored ghost entities in the object code with the string `___ghost_` (except for names of ghost compilation units). The initial triple underscore ensures that this substring cannot appear anywhere in the name of non-ghost entities or ghost entities that are not ignored. Thus, one only needs to check that the substring `___ghost_` does not appear in the list of names from the object code or executable.

On Unix-like platforms, this can done by checking that the following command does not output anything:

```
nm <object files or executable> | grep ___ghost_
```

The same can be done to check that a ghost compilation unit called `my_unit` (whatever the capitalization) is not included at all (entities in that unit would have been detected by the previous check) in the object code or executable. For example on Unix-like platforms:

```
nm <object files or executable> | grep my_unit
```

5.5.13 Quantified Expressions

[Ada 2012]

A quantified expression is a way to express a property over a collection, either an array or a container (see *Formal Containers Library*):

- a *universally quantified expression* using `for all` expresses a property that holds for all elements of a collection
- an *existentially quantified expression* using `for some` expresses a property that holds for at least one element of a collection

Quantified expressions should always be parenthesized.

Iteration Over Content vs. Over Positions

Iteration can be expressed either directly over the content of the collection, or over the range of positions of elements in the collection. The former is preferred when the property involved does not refer to the position of elements in the collection or to the previous value of the element at the same position in the collection (e.g. in a postcondition). Otherwise, the latter is needed. For example, consider the procedure `Nullify_Array` that sets each element of its array parameter `X` to zero. Its postcondition can be expressed using a universally quantified expression iterating over the content of the array as follows:

```
procedure Nullify_Array (X : out Integer_Array) with
  Post => (for all E in X => E = 0);
```

or using a universally quantified expression iterating over the range of the array as follows:

```
procedure Nullify_Array (X : out Integer_Array) with
  Post => (for all J in X'Range => X(J) = 0);
```

Quantification over formal containers can similarly iterate over their content, using the syntax `for .. of`, or their positions, using the syntax `for .. in`, see examples in *Loop Examples*.

Iteration over positions is needed when the property refers to the position of elements in the collection. For example, consider the procedure `Initialize_Array` that sets each element of its array parameter `X` to its position. Its postcondition can be expressed using a universally quantified expression as follows:

```
procedure Initialize_Array (X : out Integer_Array) with
  Post => (for all J in X'Range => X(J) = J);
```

Iteration over positions is also needed when the property refers to the previous value of the element at the same position in the collection. For example, consider the procedure `Increment_Array` that increments each element of its array parameter `X` by one. Its postcondition can be expressed using a universally quantified expression as follows:

```
procedure Increment_Array (X : in out Integer_Array) with
  Post => (for all J in X'Range => X(J) = X'Old(J) + 1);
```

The negation of a universal property being an existential property (the opposite is true too), the postcondition above can be expressed also using an existentially quantified expression as follows:

```
procedure Increment_Array (X : in out Integer_Array) with
  Post => not (for some J in X'Range => X(J) /= X'Old(J) + 1);
```

Execution vs. Proof

At run time, a quantified expression is executed like a loop, which exits as soon as the value of the expression is known: if the property does not hold (resp. holds) for a given element of a universally (resp. existentially) quantified expression, execution of the loop does not proceed with remaining elements and returns the value `False` (resp. `True`) for the expression.

When a quantified expression is analyzed with GNATprove, it uses the logical counterpart of the quantified expression. GNATprove also checks that the expression is free from run-time errors. For this checking, GNATprove checks that the enclosed expression is free from run-time errors over the *entire range* of the quantification, not only at points that would actually be reached at run time. As an example, consider the following expression:

```
(for all I in 1 .. 10 => 1 / (I - 3) > 0)
```

This quantified expression cannot raise a run-time error, because the enclosed expression `1 / (I - 3) > 0` is false for the first value of the range `I = 1`, so the execution of the loop exits immediately with the value `False` for the quantified expression. GNATprove is stricter and requires the enclosed expression `1 / (I - 3) > 0` to be free from run-time errors over the entire range `I in 1 .. 10` (including `I = 3`) so it issues a check message for a possible division by zero in this case.

Iterator Filters

The set of values or positions over which iteration is performed can be filtered with an *iterator filter* introduced by the keyword `when`. For example, we can express a property for all prime numbers in a given range as follows:

```
(for all N in 1 .. 1000 when Is_Prime (N) => ...)
```

5.6 Assertion Pragmas

SPARK contains features for directing formal verification with GNATprove. These features may also be used by other tools, in particular the GNAT compiler. Assertion pragmas are refinements of pragma `Assert` defined in Ada. For all assertion pragmas, an exception `Assertion_Error` is raised at run time when the property asserted does not hold, if the program was compiled with assertions. The real difference between assertion pragmas is how they are used by GNATprove during proof.

5.6.1 Pragma `Assert`

[Ada 2005]

Pragma `Assert` is the simplest assertion pragma. GNATprove checks that the property asserted holds, and uses the information that it holds for analyzing code that follows. For example, consider two assertions of the same property `X > 0` in procedure `Assert_Twice`:

```
1 procedure Assert_Twice (X : Integer) with
2   SPARK_Mode
3 is
4 begin
5    pragma Assert (X > 0);
6    pragma Assert (X > 0);
7 end Assert_Twice;
```

As expected, the first assertion on line 5 is not provable in absence of a suitable precondition for `Assert_Twice`, but GNATprove proves that it holds the second time the property is asserted on line 6:

```
assert_twice.adb:5:19: high: assertion might fail
    5 |   pragma Assert (X > 0);
      |                  ^~~~
  e.g. when X = 0
  possible fix: subprogram at line 1 should mention X in a precondition
    1 |procedure Assert_Twice (X : Integer) with
```

(continues on next page)

(continued from previous page)

```
      |^ here
assert_twice.adb:6:19: info: assertion proved
```

GNATprove considers that an execution of `Assert_Twice` with `X <= 0` stops at the first assertion that fails. Thus `X > 0` when execution reaches the second assertion. This is true if assertions are executed at run time, but not if assertions are discarded during compilation. In the latter case, unproved assertions should be inspected carefully to ensure that the property asserted will indeed hold at run time. This is true of all assertion pragmas, which GNATprove analyzes like pragma `Assert` in that respect.

5.6.2 Pragma `Assertion_Policy`

[Ada 2005/Ada 2012]

Assertions can be enabled either globally or locally. Here, *assertions* denote either *Assertion Pragmas* of all kinds (among which *Pragma Assert*) or functional contracts of all kinds (among which *Preconditions* and *Postconditions*).

By default, assertions are ignored in compilation, and can be enabled globally by using the compilation switch `-gnata`. They can be enabled locally by using pragma `Assertion_Policy` in the program, or globally if the pragma is put in a configuration file. They can be enabled for all kinds of assertions or specific ones only by using the version of pragma `Assertion_Policy` that takes named associations which was introduced in Ada 2012.

When used with the standard policies `Check` (for enabling assertions) or `Ignore` (for ignoring assertions), pragma `Assertion_Policy` has no effect on GNATprove. GNATprove takes all assertions into account, whatever the assertion policy in effect at the point of the assertion. For example, consider a code with some assertions enabled and some ignored:

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

procedure Assert_Enabled (X : in out Integer) with
  SPARK_Mode,
  Pre  => X > 0,  --  executed at run time
  Post => X > 2   --  ignored at run time
is
   pragma Assertion_Policy (Assert => Check);
   pragma Assert (X >= 0);  --  executed at run time

   pragma Assertion_Policy (Assert => Ignore);
   pragma Assert (X >= 0);  --  ignored at run time
begin
   X := X - 1;
end Assert_Enabled;
```

Although the postcondition and the second assertion are not executed at run time, GNATprove analyzes them and issues corresponding messages:

```
assert_enabled.adb:6:11: high: postcondition might fail
    6 |  Post => X > 2   --  ignored at run time
      |          ^~~~
  e.g. when X = 0
assert_enabled.adb:9:19: info: assertion proved
assert_enabled.adb:12:19: info: assertion proved
assert_enabled.adb:14:11: info: overflow check proved
```

On the contrary, when used with the GNAT-specific policy `Disable`, pragma `Assertion_Policy` causes the corresponding assertions to be skipped both during execution and analysis with GNATprove. For example, consider the same code as above where policy `Ignore` is replaced with policy `Disable`:

```
pragma Assertion_Policy (Pre => Check, Post => Disable);

procedure Assert_Disabled (X : in out Integer) with
  SPARK_Mode,
  Pre  => X > 0,  --  executed at run time
  Post => X > 2   --  ignored at compile time and in analysis
is
   pragma Assertion_Policy (Assert => Check);
   pragma Assert (X >= 0);  --  executed at run time

   pragma Assertion_Policy (Assert => Disable);
   pragma Assert (X >= 0);  --  ignored at compile time and in analysis
begin
   X := X - 1;
end Assert_Disabled;
```

On this program, GNATprove does not analyze the postcondition and the second assertion, and it does not issue corresponding messages:

```
assert_disabled.adb:9:19: info: assertion proved
assert_disabled.adb:14:11: info: overflow check proved
```

The policy of `Disable` should thus be reserved for assertions that are not compilable, typically because a given build environment does not define the necessary entities.

5.6.3 Loop Invariants

[SPARK]

Pragma `Loop_Invariant` is a special kind of assertion used in loops. GNATprove performs two checks that ensure that the property asserted holds at each iteration of the loop:

1. *loop invariant initialization*: GNATprove checks that the property asserted holds during the first iteration of the loop.
2. *loop invariant preservation*: GNATprove checks that the property asserted holds during an arbitrary iteration of the loop, assuming that it held in the previous iteration.

Each of these properties can be independently true or false. For example, in the following loop, the loop invariant is false during the first iteration and true in all remaining iterations:

```
   Prop := False;
   for J in 1 .. 10 loop
      pragma Loop_Invariant (Prop);
      Prop := True;
   end loop;
```

Thus, GNATprove checks that property 2 holds but not property 1:

```
simple_loops.adb:8:30: high: loop invariant might fail in first iteration
    8 |      pragma Loop_Invariant (Prop);
```

(continues on next page)

(continued from previous page)

```
      |                                ^~~~
  e.g. when Prop = False
simple_loops.adb:8:30: info: loop invariant preservation proved
```

Conversely, in the following loop, the loop invariant is true during the first iteration and false in all remaining iterations:

```
12    Prop := True;
13    for J in 1 .. 10 loop
14       pragma Loop_Invariant (Prop);
15       Prop := False;
16    end loop;
```

Thus, GNATprove checks that property 1 holds but not property 2:

```
simple_loops.adb:14:30: info: loop invariant initialization proved

simple_loops.adb:14:30: medium: loop invariant might not be preserved by an arbitrary␣
↪iteration
   14 |      pragma Loop_Invariant (Prop);
      |                             ^~~~
```

The following loop shows a case where the loop invariant holds both during the first iteration and all remaining iterations:

```
18    Prop := True;
19    for J in 1 .. 10 loop
20       pragma Loop_Invariant (Prop);
21       Prop := Prop;
22    end loop;
```

GNATprove checks here that both properties 1 and 2 hold:

```
simple_loops.adb:20:30: info: loop invariant preservation proved
simple_loops.adb:20:30: info: loop invariant initialization proved
```

In general, it is not sufficient that a loop invariant is true for GNATprove to prove it. The loop invariant should also be *inductive*: it should be precise enough that GNATprove can check loop invariant preservation by assuming *only* that the loop invariant held during the last iteration. For example, the following loop is the same as the previous one, except the loop invariant is true but not inductive:

```
24    Prop := True;
25    for J in 1 .. 10 loop
26       pragma Loop_Invariant (if J > 1 then Prop);
27       Prop := Prop;
28    end loop;
```

GNATprove cannot check property 2 on that loop:

```
simple_loops.adb:26:30: info: loop invariant initialization proved

simple_loops.adb:26:44: medium: loop invariant might not be preserved by an arbitrary␣
↪iteration, cannot prove Prop
   26 |      pragma Loop_Invariant (if J > 1 then Prop);
      |                                           ^~~~
```

Note also that not using an assertion (*Pragma Assert*) instead of a loop invariant also allows here to fully prove the corresponding property, by relying on *Automatic Unrolling of Simple For-Loops*:

```
simple_loops_unroll.adb:26:22: info: assertion proved
```

Returning to the case where automatic loop unrolling is not used, the reasoning of GNATprove for checking property 2 in that case can be summarized as follows:

- Let's take iteration K of the loop, where K > 1 (not the first iteration).
- Let's assume that the loop invariant held during iteration K-1, so we know that if K-1 > 1 then Prop holds.
- The previous assumption can be rewritten: if K > 2 then Prop.
- But all we know is that K > 1, so we cannot deduce Prop.

See *How to Write Loop Invariants* for further guidelines.

Pragma `Loop_Invariant` may appear anywhere at the top level of a loop: it is usually added at the start of the loop, but it may be more convenient in some cases to add it at the end of the loop, or in the middle of the loop, in cases where this simplifies the asserted property. In all cases, GNATprove checks loop invariant preservation by reasoning on the virtual loop that starts and ends at the loop invariant.

It is possible to use multiple loop invariants, which should be grouped together without intervening statements, declarations or pragmas, at the exception of pragma `Loop_Variant` and pragma `Annotate` (to justify check messages). The resulting complete loop invariant is the conjunction of individual ones. The benefits of writing multiple loop invariants instead of a conjunction can be improved readability and better provability (because GNATprove checks each pragma `Loop_Invariant` separately).

Finally, *Attribute Loop_Entry* and *Delta Aggregates* can be very useful to express complex loop invariants.

Note: Users that are already familiar with the notion of loop invariant in other proof systems should be aware that loop invariants in SPARK are slightly different from the usual ones. In SPARK, a loop invariant must hold when execution reaches the corresponding pragma inside the loop. Hence, it needs not hold when the loop is never entered, or when exiting the loop.

5.6.4 Loop Variants

[SPARK]

Pragma `Loop_Variant` is a special kind of assertion used in loops. GNATprove checks that the given value *progresses* in some sense at each iteration of the loop. The value is associated to a direction, which can be either `Increases` or `Decreases` for *numeric* variants, or `Structural` for *structural* variants.

Numeric variants can take a discrete value or, in the case of the direction `Decreases`, a big natural (see `SPARK.Big_Integers`). At each iteration, a check is generated to ensure that the value progresses (decreases or increases) with respect to its value at the beginning of the loop. Because a discrete value is always bounded by its type in Ada, and a big natural is never negative, it cannot decrease (or increase) at each iteration an infinite number of times, thus one of two outcomes is possible:

1. the loop exits, or
2. a run-time error occurs.

Therefore, it is possible to prove the termination of loops in SPARK programs by proving both a loop variant for each plain-loop or while-loop (for-loops always terminate in Ada) and the absence of run-time errors.

For example, the while-loops in procedure `Terminating_Loops` compute the value of `X - X mod 3` (or equivalently `X / 3 * 3`) in variable `Y`:

```
procedure Terminating_Loops (X : Natural) with
  SPARK_Mode
is
   Y : Natural;
begin
   Y := 0;
   while X - Y >= 3 loop
      Y := Y + 3;
      pragma Loop_Variant (Increases => Y);
   end loop;

   Y := 0;
   while X - Y >= 3 loop
      Y := Y + 3;
      pragma Loop_Variant (Decreases => X - Y);
   end loop;
end Terminating_Loops;
```

GNATprove is able to prove both loop variants, as well as absence of run-time errors in the subprogram, hence that loops terminate:

```
terminating_loops.adb:4:04: info: initialization of "Y" proved
terminating_loops.adb:7:12: info: overflow check proved
terminating_loops.adb:8:14: info: overflow check proved
terminating_loops.adb:9:28: info: loop variant proved
terminating_loops.adb:13:12: info: overflow check proved
terminating_loops.adb:14:14: info: overflow check proved
terminating_loops.adb:15:28: info: loop variant proved
terminating_loops.adb:15:43: info: overflow check proved
```

A numeric loop variant may be more complex than a single decreasing (or increasing) value, and be given instead by a list of either decreasing or increasing values (possibly a mix of both). In that case, the order of the list defines the lexicographic order of progress. See SPARK RM 5.5.3 for details.

The expression of a structural loop variant can be either a local borrower or a local observer (see *Observing* and *Borrowing*). A check is generated to ensure that, during each iteration of the loop, the object denoted by the variant is updated to designate a strict subcomponent of the structure it used to designate. Since, due to the *Memory Ownership Policy* of SPARK, the structure cannot contain cycles, it is enough to ensure that the loop cannot be executed an infinite number of times.

In the following example, we can verify that the `while` loop in the `Set_All_To_Zero` procedure terminates by stating that the local borrower `X` used to traverse the linked list structurally decreases at each iteration:

```
package Terminating_Loops with
  SPARK_Mode
is
   type Cell;
   type List is access Cell;
   type Cell is record
      Value : Integer;
      Next  : List;
   end record;

   procedure Set_All_To_Zero (L : List);
```

(continues on next page)

(continued from previous page)

```
12
13 end Terminating_Loops;
```

```
1  package body Terminating_Loops with
2    SPARK_Mode
3  is
4
5     procedure Set_All_To_Zero (L : List) is
6        X : access Cell := L;
7     begin
8        while X /= null loop
9           pragma Loop_Variant (Structural => X);
10          X.Value := 0;
11          X := X.Next;
12       end loop;
13    end Set_All_To_Zero;
14
15 end Terminating_Loops;
```

Structural variants are subjects to a number of restrictions. They cannot be combined with other variants, and are checked according to a mostly syntactic criterion. When these restrictions cannot be followed, structural variants can be systematically replaced by a decreasing numeric variant providing the depth (or size) of the data structure, like function `Length` in *Subprogram Variant*. Strictly speaking, structural variants are only required to define the function returning that metric.

The fact that, at each iteration, the variable `X` is updated to designate a strict subcomponent of the structure it used to designate can be verified by GNATprove:

```
terminating_loops.adb:9:10: info: loop variant proved
terminating_loops.adb:10:11: info: pointer dereference check proved
terminating_loops.adb:11:16: info: pointer dereference check proved
```

Pragma `Loop_Variant` may appear anywhere a loop invariant appears. It is also possible to use multiple loop variants, which should be grouped together with loop invariants.

5.6.5 Pragma `Assume`

[SPARK]

Pragma `Assume` is a variant of *Pragma Assert* that does not require GNATprove to check that the property holds. This is used to convey trustable information to GNATprove, in particular properties about external objects that GNATprove has no control upon. GNATprove uses the information that the assumed property holds for analyzing code that follows. For example, consider an assumption of the property `X > 0` in procedure `Assume_Then_Assert`, followed by an assertion of the same property:

```
1  procedure Assume_Then_Assert (X : Integer) with
2    SPARK_Mode
3  is
4  begin
5     pragma Assume (X > 0);
6     pragma Assert (X > 0);
7  end Assume_Then_Assert;
```

As expected, GNATprove does not check the property on line 5, but used it to prove that the assertion holds on line 6:

```
assume_then_assert.adb:6:19: info: assertion proved
```

GNATprove considers that an execution of `Assume_Then_Assert` with `X <= 0` stops at the assumption on line 5, and it does not issue a message in that case because the user explicitly indicated that this case is not possible. Thus `X > 0` when execution reaches the assertion on line 6. This is true if assertions (of which assumptions are a special kind) are executed at run time, but not if assertions are discarded during compilation. In the latter case, assumptions should be inspected carefully to ensure that the property assumed will indeed hold at run time. This inspection may be facilitated by passing a justification string as the second argument to pragma `Assume`.

5.6.6 Pragma `Assert_And_Cut`

[SPARK]

Pragma `Assert_And_Cut` is a variant of *Pragma Assert* that allows hiding some information to GNATprove. GNATprove checks that the property asserted holds, and uses *only* the information that it holds for analyzing code that follows. For example, consider two assertions of the same property `X = 1` in procedure `Forgetful_Assert`, separated by a pragma `Assert_And_Cut`:

```
1  procedure Forgetful_Assert (X : out Integer) with
2    SPARK_Mode
3  is
4  begin
5     X := 1;
6
7     pragma Assert (X = 1);
8
9     pragma Assert_And_Cut (X > 0);
10
11    pragma Assert (X > 0);
12    pragma Assert (X = 1);
13 end Forgetful_Assert;
```

GNATprove proves that the assertion on line 7 holds, but it cannot prove that the same assertion on line 12 holds:

```
forgetful_assert.adb:1:29: info: initialization of "X" proved
forgetful_assert.adb:7:19: info: assertion proved
forgetful_assert.adb:9:27: info: assertion proved
forgetful_assert.adb:11:19: info: assertion proved

forgetful_assert.adb:12:19: medium: assertion might fail
   12 |   pragma Assert (X = 1);
      |                  ^~~~~
```

GNATprove *forgets* the exact value of `X` after line 9. All it knows is the information given in pragma `Assert_And_Cut`, here that `X > 0`. And indeed GNATprove proves that such an assertion holds on line 11. But it cannot prove the assertion on line 12, showing indeed that GNATprove forgot its value of 1.

Pragma `Assert_And_Cut` may be useful in two cases:

1. When the automatic provers are overwhelmed with information from the context, pragma `Assert_And_Cut` may be used to simplify this context, thus leading to more automatic proofs.
2. When GNATprove is proving checks for each path through the subprogram (see switch `--proof` in *Running GNATprove from the Command Line*), and the number of paths is very large, pragma `Assert_And_Cut` may be

used to reduce the number of paths, thus leading to faster automatic proofs.

For example, consider procedure `P` below, where all that is needed to prove that the code using `X` is free from run-time errors is that `X` is positive. Let's assume that we are running GNATprove with switch `--proof=per_path` so that a formula is generated for each execution path. Without the pragma, GNATprove considers all execution paths through `P`, which may be many. With the pragma, GNATprove only considers the paths from the start of the procedure to the pragma, and the paths from the pragma to the end of the procedure, hence many fewer paths.

```
1  procedure P is
2     X : Integer;
3  begin
4     --  complex computation that sets X
5     pragma Assert_And_Cut (X > 0);
6     --  complex computation that uses X
7  end P;
```

GNATprove only forgets information from inside the enclosing sequence of statements, meaning information about

1. variables modified since the start of the enclosing sequence of statements
2. Boolean tests (including checks) that must have evaluated to true for control to reach the pragma from the start of the enclosing sequence of statements.

Procedure `Partial_Knowledge` below shows examples of informations that are remembered and forgotten.

```
1  procedure Partial_Knowledge (X : Integer) is
2     Y : Integer;
3     Z : Integer;
4  begin
5     Y := 0;
6     if (X <= 0) then
7        return;
8     end if;
9     begin
10       Z := 1;
11       begin
12          if (X >= 2) then
13             return;
14          end if;
15       end;
16       pragma Assert_And_Cut (Y < Z);
17       pragma Assert (Y = 0); -- Remembered
18       pragma Assert (X > 0); -- Remembered
19       pragma Assert (Y < Z); -- From cut
20       pragma Assert (Z = 1); -- Forgotten
21       pragma Assert (X < 2); -- Forgotten
22    end;
23 end;
```

Since variable `Y` is not modified in the inner block, the information that `Y` was zero is not forgotten, and the assertion at line 17 is proved. Similarly, GNATprove does not forget that `X` must have been positive to reach the inner block in the first place, and proves the assertion at line 18. However, it does not prove the following assertions at lines 20/21, and displays counter-examples with values of 2 for `X,Z`, showing indeed that GNATprove forgot the value of `Z`, as well as the fact that the program should have exited already when `X` is 2.

```
partial_knowledge.adb:2:04: info: initialization of "Y" proved
partial_knowledge.adb:3:04: info: initialization of "Z" proved
partial_knowledge.adb:16:30: info: assertion proved
partial_knowledge.adb:17:22: info: assertion proved
partial_knowledge.adb:18:22: info: assertion proved
partial_knowledge.adb:19:22: info: assertion proved

partial_knowledge.adb:20:22: medium: assertion might fail
   20 |      pragma Assert (Z = 1); -- Forgotten
      |                     ^~~~~
  e.g. when Z = 2
  possible fix: add or complete related loop invariants or postconditions

partial_knowledge.adb:21:22: medium: assertion might fail
   21 |      pragma Assert (X < 2); -- Forgotten
      |                     ^~~~~
  e.g. when X = 2
  possible fix: subprogram at line 1 should mention X in a precondition
    1 |procedure Partial_Knowledge (X : Integer) is
      |^ here
```

Note: Due to pragmas `Assert_And_Cut` and `Loop_Invariant` both acting as cut points for verification, but in slightly different ways, GNATprove does not support the full breadth of their potential interactions. Pragma `Assert_And_Cut` is only supported within loops when the immediately surrounding statement sequence does not contain the loop invariants, including any occurring within nested blocks. GNATprove also supports as a convenience the special case when the loop invariants occurs at top level in the sequence prefix preceding pragma `Assert_And_Cut`, by implicitly assuming that a new block starts immediately after the last pragma `Loop_Invariant`.

5.7 Overflow Modes

Annotations such as preconditions, postconditions, assertions, loop invariants, are analyzed by GNATprove with the exact same meaning that they have during execution. In particular, evaluating the expressions in an annotation may raise a run-time error, in which case GNATprove will attempt to prove that this error cannot occur, and report a warning otherwise.

Integer overflows are a kind of run-time error that occurs when the result of an arithmetic computation does not fit in the bounds of the machine type used to hold the result. In some cases, it is convenient to express properties in annotations as they would be expressed in mathematics, where quantities are unbounded. This is best achieved using the *Big Numbers Library*, which defines types for unbounded integers and rational numbers, operations on these and conversions from/to machine integers and reals.

Alternatively, GNATprove supports different overflow modes, so that the usual signed arithmetic operations are interpreted differently from their standard interpretation. For example:

```
1 function Add (X, Y : Integer) return Integer with
2   Pre  => X + Y in Integer,
3   Post => Add'Result = X + Y;
```

The precondition of `Add` states that the result of adding its two parameters should fit in type `Integer`. In the default mode, evaluating this expression will fail an overflow check, because the result of `X + Y` is stored in a temporary of type `Integer`. If the compilation switch `-gnato13` is used, then annotations are compiled specially, so that arithmetic

operations use unbounded intermediate results. In this mode, GNATprove does not generate a check for the addition of `X` and `Y` in the precondition of `Add`, as there is no possible overflow here.

There are three overflow modes:

- Use base type for intermediate operations (STRICT): in this mode, all intermediate results for predefined arithmetic operators are computed using the base type, and the result must be in range of the base type.
- Most intermediate overflows avoided (MINIMIZED): in this mode, the compiler attempts to avoid intermediate overflows by using a larger integer type, typically Long_Long_Integer, as the type in which arithmetic is performed for predefined arithmetic operators.
- All intermediate overflows avoided (ELIMINATED): in this mode, the compiler avoids all intermediate overflows by using arbitrary precision arithmetic as required.

The desired mode for handling intermediate overflow can be specified using either the Overflow_Mode pragma or an equivalent compiler switch. The pragma has the form:

```
pragma Overflow_Mode ([General =>] MODE [, [Assertions =>] MODE]);
```

where MODE is one of

- STRICT: intermediate overflows checked (using base type)
- MINIMIZED: minimize intermediate overflows
- ELIMINATED: eliminate intermediate overflows

For example:

```
pragma Overflow_Mode (General => Strict, Assertions => Eliminated);
```

specifies that general expressions outside assertions be evaluated in the usual strict mode, and expressions within assertions be evaluated in "eliminate intermediate overflows" mode. Currently, GNATprove only supports pragma `Overflow_Mode` being specified as a configuration pragma, either in a configuration pragma file or directly in a unit.

Additionally, a compiler switch `-gnato??` can be used to control the checking mode default. Here *?* is one of the digits *1* through *3*:

1. use base type for intermediate operations (STRICT)
2. minimize intermediate overflows (MINIMIZED)
3. eliminate intermediate overflows (ELIMINATED)

The switch `-gnato13`, like the `Overflow_Mode` pragma above, specifies that general expressions outside assertions be evaluated in the usual strict mode, and expressions within assertions be evaluated in "eliminate intermediate overflows" mode.

Note that these modes apply only to the evaluation of predefined arithmetic, membership, and comparison operators for signed integer arithmetic.

For further details of the meaning of these modes, and for further information about the treatment of overflows for fixed-point and floating-point arithmetic please refer to the "Overflow Check Handling in GNAT" appendix in the GNAT User's Guide.

5.8 Object Oriented Programming and Liskov Substitution Principle

SPARK supports safe Object Oriented Programming by checking behavioral subtyping between parent types and derived types, a.k.a. Liskov Substitution Principle: every overriding operation of the derived type should behave so that it can be substituted for the corresponding overridden operation of the parent type anywhere.

5.8.1 Class-Wide Subprogram Contracts

[Ada 2012]

Specific *Subprogram Contracts* are required on operations of tagged types, so that GNATprove can check Liskov Substitution Principle on every overriding operation:

- The *class-wide precondition* introduced by aspect `Pre'Class` is similar to the normal precondition.
- The *class-wide postcondition* introduced by aspect `Post'Class` is similar to the normal postcondition.

Although these contracts are defined in Ada, they have a stricter meaning in SPARK for checking Liskov Substitution Principle:

- The class-wide precondition of an overriding operation should be weaker (more permissive) than the class-wide precondition of the corresponding overridden operation.
- The class-wide postcondition of an overriding operation should be stronger (more restrictive) than the class-wide postcondition of the corresponding overridden operation.

For example, suppose that the `Logging` unit introduced in *Ghost Packages* defines a tagged type `Log_Type` for logs, with corresponding operations:

```
1  package Logging with
2    SPARK_Mode
3  is
4     Max_Count : constant := 10_000;
5
6     type Log_Count is range 0 .. Max_Count;
7
8     type Log_Type is tagged private;
9
10    function Log_Size (Log : Log_Type) return Log_Count;
11
12    procedure Init_Log (Log : out Log_Type) with
13      Post'Class => Log.Log_Size = 0;
14
15    procedure Append_To_Log (Log : in out Log_Type; Incr : in Integer) with
16      Pre'Class  => Log.Log_Size < Max_Count,
17      Post'Class => Log.Log_Size = Log.Log_Size'Old + 1;
18
19 private
20
21    subtype Log_Index is Log_Count range 1 .. Max_Count;
22    type Integer_Array is array (Log_Index) of Integer;
23
24    type Log_Type is tagged record
25       Log_Data : Integer_Array;
26       Log_Size : Log_Count;
```

(continues on next page)

(continued from previous page)

```
   end record;

   function Log_Size (Log : Log_Type) return Log_Count is (Log.Log_Size);

end Logging;
```

and that this type is derived in `Range_Logging.Log_Type` which additionally keeps track of the minimum and maximum values in the log, so that they can be accessed in constant time:

```
with Logging; use type Logging.Log_Count;

package Range_Logging with
  SPARK_Mode
is
   type Log_Type is new Logging.Log_Type with private;

   not overriding
   function Log_Min (Log : Log_Type) return Integer;

   not overriding
   function Log_Max (Log : Log_Type) return Integer;

   overriding
   procedure Init_Log (Log : out Log_Type) with
     Post'Class => Log.Log_Size = 0 and
                   Log.Log_Min = Integer'Last and
                   Log.Log_Max = Integer'First;

   overriding
   procedure Append_To_Log (Log : in out Log_Type; Incr : in Integer) with
     Pre'Class  => Log.Log_Size < Logging.Max_Count,
     Post'Class => Log.Log_Size = Log.Log_Size'Old + 1 and
                   Log.Log_Min = Integer'Min (Log.Log_Min'Old, Incr) and
                   Log.Log_Max = Integer'Max (Log.Log_Max'Old, Incr);

private

   type Log_Type is new Logging.Log_Type with record
     Min_Entry : Integer;
     Max_Entry : Integer;
   end record;

   function Log_Min (Log : Log_Type) return Integer is (Log.Min_Entry);
   function Log_Max (Log : Log_Type) return Integer is (Log.Max_Entry);

end Range_Logging;
```

GNATprove proves that the contracts on `Logging.Append_To_Log` and its overriding `Range_Logging.Append_To_Log` respect the Liskov Substitution Principle:

```
range_logging.ads:9:13: info: implicit aspect Always_Terminates on "Log_Min" has been␣
↪proved, subprogram will terminate
```

(continues on next page)

(continued from previous page)

```
range_logging.ads:12:13: info: implicit aspect Always_Terminates on "Log_Max" has been␣
↪proved, subprogram will terminate
range_logging.ads:16:20: info: class-wide postcondition is stronger than overridden one
range_logging.ads:22:20: info: class-wide precondition is weaker than overridden one
range_logging.ads:23:20: info: class-wide postcondition is stronger than overridden one
logging.ads:10:13: info: implicit aspect Always_Terminates on "Log_Size" has been proved,
↪ subprogram will terminate
```

Units `Logging` and `Range_Logging` need not be implemented, or available, or in SPARK. It is sufficient that the specification of `Logging` and `Range_Logging` are in SPARK for this checking. Here, the postcondition of `Range_Logging.Append_To_Log` is strictly stronger than the postcondition of `Logging.Append_To_Log`, as it also specifies the new expected value of the minimum and maximum values. The preconditions of both procedures are exactly the same, which is the most common case, but in other cases it might be useful to be more permissive in the overriding operation's precondition. For example, `Range_Logging.Append_To_Log` could allocate dynamically additional memory for storing an unbounded number of events, instead of being limited to `Max_Count` events like `Logging.Append_To_Log`, in which case its precondition would be simply `True` (the default precondition).

A derived type may inherit both from a parent type and from one or more interfaces, which only provide abstract operations and no components. GNATprove checks Liskov Substitution Principle on every overriding operation, both when the overridden operation is inherited from the parent type and when it is inherited from an interface.

GNATprove separately checks that a subprogram implements its class-wide contract, like for a specific contract.

5.8.2 Mixing Class-Wide and Specific Subprogram Contracts

[Ada 2012]

It is possible to specify both a specific contract and a class-wide contract on a subprogram, in order to use a more precise contract (the specific one) for non-dispatching calls and a contract compatible with the Liskov Substitution Principle (the class-wide contract) for dispatching calls. In that case, GNATprove checks that:

- The specific precondition is weaker (more permissive) than the class-wide precondition.
- The specific postcondition is stronger (more restrictive) than the class-wide postcondition.

For example, `Logging.Append_To_Log` could set a boolean flag `Special_Value_Logged` when some `Special_Value` is appended to the log, and express this property in its specific postcondition so that it is available for analyzing non-dispatching calls to the procedure:

```
procedure Append_To_Log (Log : in out Log_Type; Incr : in Integer) with
  Pre'Class  => Log.Log_Size < Max_Count,
  Post'Class => Log.Log_Size = Log.Log_Size'Old + 1,
  Post       => Log.Log_Size = Log.Log_Size'Old + 1 and
                (if Incr = Special_Value then Special_Value_Logged = True);
```

This additional postcondition would play no role in dispatching calls, thus it is not involved in checking the Liskov Substitution Principle. Note that the absence of specific precondition on procedure `Append_To_Log` does not mean that the default precondition of `True` is used: as a class-wide precondition is specified on procedure `Append_To_Log`, it is also used as specific precondition. Similarly, if a procedure has a class-wide contract and a specific precondition, but no specific postcondition, then the class-wide postcondition is also used as specific postcondition.

When both a specific contract and a class-wide contract are specified on a subprogram, GNATprove only checks that the subprogram implements its specific (more precise) contract.

5.8.3 Dispatching Calls and Controlling Operands

[Ada 2012]

In a dispatching call, the *controlling operand* is the parameter of class-wide type whose dynamic type determinates the actual subprogram called. The dynamic type of this controlling operand may be any type derived from the specific type corresponding to the class-wide type of the parameter (the specific type is `T` when the class-wide type is `T'Class`). Thus, in general it is not possible to know in advance which subprograms may be called in a dispatching call, when separately analyzing a unit.

In SPARK, there is no need to know all possible subprograms called in order to analyze a dispatching call, which makes it possible for GNATprove to perform this analysis without knowledge of the whole program. As SPARK enforces Liskov Substitution Principle, the class-wide contract of an overriding operation is always less restrictive than the class-wide contract of the corresponding overridden operation. Thus, GNATprove uses the class-wide contract of the operation for the specific type of controlling operand to analyze a dispatching call.

For example, suppose a global variable `The_Log` of class-wide type defines the log that should be used in the program:

```
The_Log : Logging.Log_Type'Class := ...
```

The call to `Append_To_Log` in procedure `Add_To_Total` may dynamically call either `Logging.Append_To_Log` or `Range_Logging.Append_To_Log`:

```
procedure Add_To_Total (Incr : in Integer) is
begin
   Total := Total + Incr;
   The_Log.Append_To_Log (Incr);
end Add_To_Total;
```

Because GNATprove separately checks Liskov Substitution Principle for procedure `Append_To_Log`, it can use the class-wide contract of `Logging.Append_To_Log` for analyzing procedure `Add_To_Total`.

5.8.4 Dynamic Types and Invisible Components

[SPARK]

The *Data Initialization Policy* in SPARK applies specially to objects of tagged type. In general, the dynamic type of an object of tagged type may be different from its static type, hence the object may have invisible components, that are only revealed when the object is converted to a class-wide type.

For objects of tagged type, modes on parameters and data dependency contracts have a different meaning depending on the object's static type:

- For objects of a specific (not class-wide) tagged type, the constraints described in *Data Initialization Policy* apply to the visible components of the object only.
- For objects of a class-wide type, the constraints described in *Data Initialization Policy* apply to all components of the object, including invisible ones.

GNATprove checks during flow analysis that no uninitialized data is read in the program, and that the specified data dependencies and flow dependencies are respected in the implementation, based on the semantics above for objects of tagged type. For example, it detects no issues during flow analysis on procedure `Use_Logging` which initializes parameter `Log` and then updates it:

```
1 with Logging; use Logging;
2
3 procedure Use_Logging (Log : out Log_Type) with
```

(continues on next page)

(continued from previous page)

```
  SPARK_Mode
is
begin
   Log.Init_Log;
   Log.Append_To_Log (1);
end Use_Logging;
```

If parameter `Log` is of dynamic type `Logging.Log_Type`, then the call to `Init_Log` initializes all components of `Log` as expected, and the call to `Append_To_Log` can safely read those. If parameter `Log` is of dynamic type `Range_Logging.Log_Type`, then the call to `Init_Log` only initializes those components of `Log` that come from the parent type `Logging.Log_Type`, but since the call to `Append_To_Log` only read those, then there is no read of uninitialized data. This is in contrast with what occurs in procedure `Use_Logging_Classwide`:

```
with Logging; use Logging;

procedure Use_Logging_Classwide (Log : out Log_Type'Class) with
  SPARK_Mode
is
begin
   Log_Type (Log).Init_Log;
   Log.Append_To_Log (2);
end Use_Logging_Classwide;
```

on which GNATprove issues a check message during flow analysis:

```
use_logging_classwide.adb:8:04: high: extension of "Log" is not initialized
    8 |   Log.Append_To_Log (2);
```

Indeed, the call to `Init_Log` (a non-dispatching call to `Logging.Init_Log` due to the conversion on its parameter) only initializes those components of `Log` that come from the parent type `Logging.Log_Type`, but the call to `Append_To_Log` may read other components from `Range_Logging.Log_Type` which may not be initialized.

A consequence of these rules for data initialization policy is that a parameter of a specific tagged type cannot be converted to a class-wide type, for example for a dispatching call. A special aspect `Extensions_Visible` is defined in SPARK to allow this case. When `Extensions_Visible` is specified on a subprogram, the data initialization policy for the subprogram parameters of a specific tagged type requires that the constraints described in *Data Initialization Policy* apply to all components of the object, as if the parameter was of a class-wide type. This allows converting this object to a class-wide type.

5.9 Pointer Support and Dynamic Memory Management

Access types are supported in SPARK but with major restrictions. Here is an overview of the kind of access types supported in SPARK, their restrictions, and what they can be used for.

- Named pool-specific access-to-variable types can only designate data allocated on the heap (this is an Ada rule). SPARK enforces a *Memory Ownership Policy* to retain absence of aliasing, see *Access to Objects and Ownership*. Values of such an access type can be deallocated safely. GNATprove generates verification conditions to ensure that no memory can be leaked.

  ```
  type PS_Int_Acc is access Integer;
  X1 : PS_Int_Acc := new Integer'(15);
  ```

- Named access-to-constant types (using the keyword `constant`) can be used to designate data regardless of where it is allocated (the stack, the heap...), but they cannot be deallocated. Objects of this type are not subject to any ownership checking but the value they designate should be constant all the way down (ie. if such a value has a subcomponent of an access-to-variable type, the value designated by this subcomponent should be constant too).

```
type Cst_Int_Acc is access constant Integer;
C  : aliased constant Integer := 15;
X2 : Cst_Int_Acc := C'Access;
```

- Named general access-to-variable types (using the keyword `all`) can designate data regardless of where it is allocated. Like access-to-constant types, they cannot be deallocated, so GNATprove will flag memory leaks as soon as a value of such a type is allocated on the heap. They are subject to the *Memory Ownership Policy* of SPARK.

```
type Gen_Int_Acc is access all Integer;
V  : aliased Integer := 15;
X3 : Gen_Int_Acc := V'Access;
```

- Anonymous access-to-object types can only be used as the type of stand-alone objects for *Observing* and *Borrowing* and as the return type of *Traversal Functions*. In particular they cannot be stored inside composite types. They are used to grant temporary access to parts of other data-structures (recursive data-structures, composite types, formal containers...).

```
type List;
type List_Acc is access List;
type List is record
  Value : aliased Integer;
  Next  : List_Acc;
end record;

L : List_Acc := new List'(14, new List'(15, new List'(16, null)));
B : access Integer := L.Next.Value'Access;
```

- Access-to-subprogram types can designate functions and procedures. Named access-to-subprogram types can be annotated with a contract, see *Contracts for Subprogram Pointers*, but the designated subprograms cannot currently have global inputs or outputs.

```
type Func_Acc is not null access function (X : Natural) return Natural;
function Id (X : Natural) return Natural is (X);
F : Func_Acc := Id'Access;
```

5.9.1 Access to Objects and Ownership

In SPARK, values of an access-to-variable type are subject to a *Memory Ownership Policy*. The idea is that an object designated by a pointer always has a single owner, which retains the right to either modify it, or (exclusive or) share it with others in a read-only way. Said otherwise, we always have either several copies of the pointer that allow only reading, or only a single copy of the pointer that allows modification.

The main idea used to enforce single ownership for pointers is the move semantics of assignments. When a pointer is copied through an assignment statement, the ownership of the pointer is transferred to the left hand side of the assignment. As a result, the right hand side loses the ownership of the object, and therefore loses the right to access it, both for writing and reading. In the example below, the assignment from `X` to `Y` causes `X` to lose ownership on the value it references:

```
procedure Test is
   type Int_Ptr is access Integer;
   X : Int_Ptr := new Integer'(10);
   Y : Int_Ptr;                 --  Y is null by default
begin
   Y := X;                      --  ownership of X is transferred to Y
   pragma Assert (Y.all = 10); --  Y can be accessed
   Y.all := 11;                 --  both for reading and writing
   pragma Assert (X.all = 11); --  but X cannot, or we would have an alias
end Test;
```

As a result, the last assertion, which reads the value of `X`, is illegal in SPARK, leading to an error message from GNATprove:

```

test.adb:9:21: error: dereference from "X" is not readable
    9 |   pragma Assert (X.all = 11); --  but X cannot, or we would have an alias
      |                   ~~^~~
  object was moved at line 6 [E0010]
    6 |   Y := X;                     --  ownership of X is transferred to Y
      |        ^ here
  launch "gnatprove --explain=E0010" for more information
gnatprove: error during flow analysis and proof
```

In this example, we can see the point of these ownership rules. To correctly reason about the semantics of a program, SPARK needs to know, when a change is made, what are the objects that are potentially impacted. Because it assumes that there can be no aliasing (at least no aliasing of mutable data, see *Absence of Interferences*), the tool can easily determine what are the parts of the environment that are updated by a statement, be it a simple assignment, or for example a procedure call. If we were to break this assumption, we would need to either assume the worst (that all references can be aliases of each other) or require the user to explicitly annotate subprograms to describe which references can be aliased and which cannot. In our example, SPARK can deduce that an assignment to `Y` cannot impact `X`. This is only correct because of ownership rules that prevent us from accessing the value of `X` after the update of `Y`.

Note that a variable which has been moved is not necessarily lost for the rest of the program. Indeed, it is possible to assign it again, restoring ownership. For example, here is a piece of code that swaps the pointers `X` and `Y`:

```
procedure Test is
   type Int_Ptr is access Integer;
   X : Int_Ptr := new Integer'(10);
   Y : Int_Ptr;         --  Y is null by default
   Tmp : Int_Ptr := X; --  ownership of X is moved to Tmp
                        --  X cannot be accessed.
begin
   X := Y; --  ownership of Y is moved to X
           --  Y cannot be accessed
           --  X has full ownership.
   Y := Tmp; --  ownership of Tmp is moved to Y
             --  Tmp cannot be accessed
             --  Y has fullownership.
end Test;
```

This code is accepted by GNATprove. Intuitively, we can see that writing at top-level into `X` after it has been moved is OK, since it will not modify the actual owner of the moved value (here `Tmp`). However, writing in `X.all` is forbidden, as it would affect `Tmp`:

```
procedure Test is
   type Int_Ptr is access Integer;
   X   : Int_Ptr := new Integer'(10);
   Tmp : Int_Ptr := X; --  ownership of X is moved to Tmp
                       --  X cannot be accessed.
begin
   X.all := 0;
end Test;
```

The above variant is rejected by GNATprove:

```

test.adb:7:06: error: dereference from "X" is not writable
    7 |   X.all := 0;
      |   ~~^~~
  object was moved at line 4 [E0010]
    4 |   Tmp : Int_Ptr := X; --  ownership of X is moved to Tmp
      |                   ^ here
  launch "gnatprove --explain=E0010" for more information
gnatprove: error during flow analysis and proof
```

5.9.2 Attribute Access

Let's consider objects `Variable` and `Const`, respectively a variable and constant of type `T`, marked as `aliased` so that it is possible to use attribute `Access` on them:

```
Variable : aliased T;
Const    : aliased constant T := ...;
```

Depending on the type of the attribute reference expression, taking an access value to an object is interpreted differently in SPARK.

- attribute `'Access` of an anonymous access type:

  ```
  Variable_Handle : access T := Variable'Access;
  Const_Handle    : access constant T := Const'Access;
  ```

  The `'Access` attribute of an anonymous access-to-variable type, like for `Variable_Handle` above, allows *Borrowing* a part of an object temporarily, like `Variable` here. The `'Access` attribute of an anonymous access-to-constant type, like for `Const_Handle` above, allows *Observing* a part of an object temporarily, like `Const` here.

- attribute `'Access` of a general access-to-variable type:

  ```
  type General_Ptr is access all T;
  General_Handle : General_Ptr := Variable'Access;
  ```

  The `'Access` attribute of a general access-to-variable type, like for `General_Handle` above, allows moving the ownership of a local object, like `Variable` here, into a pointer. Ownership cannot be reclaimed back by `Variable` which should not be read or written directly afterwards. This is only allowed in SPARK if `Variable` is a local object, i.e. it is declared inside a subprogram.

- attribute `'Access` of a named access-to-constant type:

```
type Const_Ptr is access constant T;
Const_Handle : Const_Ptr := Const'Access;
```

The `'Access` attribute of a named access-to-constant type, like for `Const_Handle` above, allows sharing a read-only access to a constant part of an object, like `Const` here.

5.9.3 Deallocation

At the end of its lifetime, unless the memory it points to is transferred to another owner, an owning pointer should be deallocated. This is typically achieved by instantiating the standard generic procedure `Ada.Unchecked_Deallocation` with the type of the underlying `Object` and the type `Name` of the access type:

```
1  with Ada.Unchecked_Deallocation;
2
3  procedure Test is
4     type Int_Ptr is access Integer;
5
6     procedure Free is new Ada.Unchecked_Deallocation (Object => Integer, Name => Int_Ptr);
7
8     X : Int_Ptr := new Integer'(10);
9     Y : Int_Ptr;
10 begin
11    Y := X;
12    Free (Y);
13 end Test;
```

GNATprove guarantees the absence of memory leak in the above code:

```
1  test.adb:8:04: info: absence of resource or memory leak at end of scope proved
2  test.adb:9:04: info: initialization of "Y" proved
3  test.adb:9:04: info: absence of resource or memory leak at end of scope proved
4  test.adb:11:06: info: absence of resource or memory leak proved
```

Notice that there are three kinds of checks for memory leaks:

1. On each assignment, GNATprove checks that the left-hand side is not leaking memory. That's the case on the assignment to `Y` above on line 11.
2. On each declaration, GNATprove checks that the object is not leaking memory at the end of its lifetime. That's the case for the declarations of `X` and `Y` above on lines 8 and 9.
3. On each call to an instance of `Ada.Unchecked_Deallocation`, GNATprove checks that the underlying memory is not itself owning memory. Above, the object pointed to is an integer, so this holds trivially.

Here is an example of code with all three cases of memory leaks:

```
1  with Ada.Unchecked_Deallocation;
2
3  procedure Test is
4     type Int_Ptr is access Integer;
5     type Int_Ptr_Ptr is access Int_Ptr;
6
7     procedure Free is new Ada.Unchecked_Deallocation (Object => Int_Ptr, Name => Int_Ptr_
   ↪Ptr);
```

(continues on next page)

(continued from previous page)

```
8
9     X : Int_Ptr     := new Integer'(10);  -- memory leak at end of scope
10    Y : Int_Ptr     := new Integer'(11);
11    Z : Int_Ptr_Ptr := new Int_Ptr'(Y);
12 begin
13    Z.all := X;  -- memory leak on assignment
14    X := new Integer'(12);
15    Free (Z);    -- memory leak on deallocation
16 end Test;
```

GNATprove detects all three memory leaks in the above code:

```
1
2  test.adb:9:04: medium: resource or memory leak might occur at end of scope
3      9 |   X : Int_Ptr     := new Integer'(10);  -- memory leak at end of scope
4        |   ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
5
6  test.adb:13:10: medium: resource or memory leak might occur
7     13 |   Z.all := X;  -- memory leak on assignment
8        |   ~~~~~~^~~~
9
10 test.adb:15:04: medium: resource or memory leak might occur
11    15 |   Free (Z);    -- memory leak on deallocation
12       |   ^~~~~~~
```

Finally, in a case like above where a data structure manipulated through pointers also contains pointers, it is customary to define deallocation procedures to take care of deallocating the complete subtree of allocated memory. This is done in the following code by defining a higher-level `Free` procedure applying to arguments of type `Int_Ptr_Ptr`, which is based on instances of `Ada.Unchecked_Deallocation` for deallocating individual memory chunks:

```
1  with Ada.Unchecked_Deallocation;
2
3  procedure Test is
4     type Int_Ptr is access Integer;
5     type Int_Ptr_Ptr is access Int_Ptr;
6
7     procedure Free is new Ada.Unchecked_Deallocation (Object => Integer, Name => Int_Ptr);
8
9     procedure Free (X : in out Int_Ptr_Ptr) with
10      Depends => (X => null,
11                  null => X),
12      Post => X = null
13    is
14       procedure Internal_Free is new Ada.Unchecked_Deallocation
15         (Object => Int_Ptr, Name => Int_Ptr_Ptr);
16    begin
17       if X /= null and then X.all /= null then
18          Free (X.all);
19       end if;
20       Internal_Free (X);
21    end Free;
22
```

(continues on next page)

(continued from previous page)

```
   Y : Int_Ptr     := new Integer'(11);
   Z : Int_Ptr_Ptr := new Int_Ptr'(Y);
begin
   Free (Z);
end Test;
```

Note the contract of the higher-level `Free` procedure, with a postcondition stating that `X` is null on exit, and correct dependences similar to what is defined for the standard `Ada.Unchecked_Deallocation`. GNATprove guarantees that the above code is correctly deallocating memory:

```
test.adb:10:06: info: flow dependencies proved
test.adb:12:14: info: postcondition proved
test.adb:17:31: info: pointer dereference check proved
test.adb:18:18: info: pointer dereference check proved
test.adb:20:07: info: absence of resource or memory leak proved
test.adb:23:04: info: absence of resource or memory leak at end of scope proved
test.adb:24:04: info: absence of resource or memory leak at end of scope proved
```

5.9.4 Observing

The ownership policy of SPARK allows sharing a single reference between several readers. This mechanism is called *observing*. When a variable is observed, both the observed object and the observer retain the right to read the object, but none can modify it. When the observer disappears, the observed object regains the permissions it had before (read-write or read-only).

To declare an observer, it is necessary to use an anonymous access-to-constant type. It is what allows the tool to tell the difference between moving and observing a value. Here is an example. We have a list `L`, defined as a recursive pointer-based data structure in the usual way. We then observe its tail by introducing a local observer `N` using an anonymous access to constant type. We then do it again to observe the tail of `N`:

```
type List;
type List_Acc is access List;
type List is record
   Value : Element;
   Next  : List_Acc;
end record;

L : List := ...;

declare
   N : access constant List := L.Next; -- observe part of L
begin
   declare
      M : access constant List := N.Next; -- observe again part of N
   begin
      pragma Assert (M.Value = 3); --  M can be read
      pragma Assert (N.Value = 2); --  but we can still read N
      pragma Assert (L.Value = 1); --  and even L
   end;
end;
L.Next := null; --  all observers are out of scope, we can modify L
```

We can see that the three variables retain the right to read their content. But it is OK as none of them is allowed to update it. When no more observers exist, it is again possible to modify L.

It is not possible to update a data structure through an observer, but it does not mean that the observer itself is necessarily a constant. It is possible to update it so that it designates another part of a data structure. This is especially useful to traverse recursive data structures using loops. As an example, here is a function which searches for the an element E in a list L:

```
function Contains (L : access constant List; E : Element) return Boolean is
   C : access constant List := L; --  C observes L
begin
   while C /= null loop
      if C.Value = E then
         return True;
      end if;
      C := C.Next; --  C now designates the tail of the list
   end loop;
   return False;
end Contains;
```

A local observer C is used to traverse the list L. At each iteration of the loop, C is shifted so that it designates one element further in the list.

5.9.5 Borrowing

Moving is not the only way to transfer ownership. It is also possible to *borrow* the ownership of (a part of) an object for a period of time. During this period, the part of the object which was borrowed can only be accessed through the borrower. When the borrower disappears (goes out of scope), the borrowed object regains the ownership, and is accessible again. It is what happens for example for mutable parameters of a subprogram when the subprogram is called. The ownership of the actual parameter is transferred to the formal parameter for the duration of the call, and should be returned when the subprogram terminates. In particular, this disallows procedures that move some of their parameters away, as in the following example:

```
1  type Int_Ptr_Holder is record
2     Content : Int_Ptr;
3  end record;
4
5  procedure Move (X : in out Int_Ptr_Holder; Y : in out Int_Ptr_Holder) is
6  begin
7     X := Y; --  ownership of Y.Content is moved to X.Content
8  end Move;
```

```
insufficient permission for "Y" when returning from "Move"
object was moved at line 7
```

Note that borrowing does not occur on subprogram calls for in out parameters of a named access type. Indeed, SPARK RM has a special wording for these parameters, stating that they are not borrowed but moved on entry and on exit of the subprogram. This allows us to move these parameters inside the call, so they can designate something else (or be set to `null`), which otherwise would be forbidden, as borrowed top-level access objects cannot be moved (but parts of such objects can be moved).

The ownership policy of SPARK also allows declaring local borrowers in a nested scope by using an anonymous access-to-variable type (without the `constant` keyword used above for an observer):

```
declare
  Y : access Integer := X;     --  Y borrows the ownership of X
                               --  for the duration of the declare block
begin
  pragma Assert (Y.all = 10); --  Y can be accessed
  Y.all := 11;                --  both for reading and writing
end;
pragma Assert (X.all = 11);    --  The ownership of X is restored,
                               --  it can be accessed again
```

Just like local observers, local borrowers are especially useful to modify a recursive data structure through a loop. In the example below, the procedure `Replace_Element` uses a loop to assign a new value `E` to the element at position `N` in a list `L`.

```
procedure Replace_Element (L : access List; N : Positive; E : Element) is
   X : access List := L; --  X borrows the ownership of L
   P : Positive := N;
begin
   while X /= null loop
      if P = 1 then
         X.Value := E; --  We use X to modify L arbitrarily deeply
         return;
      end if;
      X := X.Next; --  X now designates the tail of the list
      P := P - 1;
   end loop;
end Replace_Element;
```

A local borrower `X` is used to traverse the list and modify it in place. The two assignments to `X` in the loop are different in essence. The first one assigns a part of the structure designated by `X`. It is a modification of the borrowed list `L`. The second one assigns `X` at top-level. It does not modify `L`, but changes `X` so that it designates another the part of L. It is called a *reborrow*. In SPARK, reborrows are only allowed to borrow a part of the borrower. Said otherwise, a borrower can only go deeper in the data structure, it is not allowed to jump to a distinct object or distinct part of the same standalone object.

Borrowers essentially are statically known aliases of their borrowed objects. As a consequence, verifying programs involving borrowers sometimes requires describing the relation between the borrowed object and the borrower. This can be done by using an *Annotation for Referring to a Value at the End of a Local Borrow*.

5.9.6 Traversal Functions

It is possible to write a function which computes and returns an observer or a borrower of an input data structure, provided the traversed data structure is itself an access type. This is called a *traversal function*.

An *observing* traversal function takes an access type as its first parameter and returns an anonymous access-to-constant object which should be a part of this first parameter. As an example, we can write a function which returns a value stored in a list as an anonymous access-to-constant type. To be able to do this, we need to store an access to the value instead of the value itself in the list:

```
type List;
type List_Acc is access List;
type Element_Acc is not null access Element;
type List is record
```

(continues on next page)

(continued from previous page)

```
   Value : Element_Acc;
   Next  : List_Acc;
end record;

function Constant_Access (L : access constant List; N : Positive) return access constant␣
↪Element
is
   C : access constant List := L;
   P : Positive := N;
begin
   while C /= null loop
      if P = 1 then
         return C.Value;
      end if;
      C := C.Next;
      P := P - 1;
   end loop;
   return null;
end Constant_Access;
```

The function `Constant_Access` returns an access designating a value which is already contained in the list `L`. As per the ownership policy of SPARK, if `Constant_Access` was returning a named access type, it would be rejected. However, since it returns an anonymous access-to-constant type, the return statement is considered to create an observer of `L`. Note that an observing traversal function should necessarily observe its first parameter.

```
declare
  C : access constant Element := Constant_Access (L, 3);
  --  C is an observer of L
begin
  pragma Assert (C.all = L.Next.Next.Value.all);
  --  It is OK to read both C and L, but L cannot be modified
end;
L := null; --  L can be modified again
```

It is also possible to return a mutable access inside a data structure using a *borrowing* traversal function. Just like observing traversal functions, their borrowing counterparts take as a first parameter an access type, but they have as a return type an anonymous access-to-variable type. The function `Reference` below is similar to `Constant_Access` except that both its parameter and its return type are mutable:

```
function Reference (L : access List; N : Positive) return access Element
is
   C : access List := L;
   P : Positive := N;
begin
   while C /= null loop
      if P = 1 then
         return C.Value;
      end if;
      C := C.Next;
      P := P - 1;
   end loop;
   return null;
end Reference;
```

A borrowing traversal function returns a borrower of its first parameter. The result of a call to `Reference` can be used to modify its actual parameter arbitrarily deeply. Like for any borrowers, it is illegal to either read or modify the parameter while the object storing the result of the call is still in scope.

Note that it is possible to use pledges to describe the relation between the result of a borrowing traversal function and its parameter in a postcondition, see *Annotation for Referring to a Value at the End of a Local Borrow*.

5.9.7 Subprogram Pointers

Unlike access to objects, access to subprograms are not subject to the ownership policy of SPARK. Both anonymous and named access-to-subprogram types are supported. As an example, the procedure `Update_All` below calls its parameter `Update_One` on all the elements of its array parameter `A`:

```
procedure Update_All
  (A          : in out Nat_Array;
   Update_One : not null access procedure (X : in out Natural))
is
begin
   for E of A loop
      Update_One (E);
   end loop;
end Update_All;
```

It can be called on any procedure with the correct profile:

```
procedure Update_One (X : in out Natural);

procedure Test (A : in out Nat_Array)  is
begin
   Update_All (A, Update_One'Access);
end Test;
```

As GNATprove verifies subprograms modularly, no precondition checks are generated during the analysis of `Update_All`. As a consequence, a check needs to be performed in `Test` to make sure that the parameter supplied for `Update_One` does not have a precondition. For example, if we modify `Update_One` to have a precondition:

```
function In_Range (X : Natural) return Boolean;

procedure Update_One (X : in out Natural) with
  Pre  => In_Range (X);
```

Then GNATprove will complain on the call to `Update_All` that the precondition of `Update_One` might not be satisfied:

```
medium: precondition of target might not be strong enough to imply precondition of␣
↪source, cannot prove In_Range (X)
```

For postconditions, it is the opposite. No postconditions will be assumed when verifying `Update_All`, so it is perfectly OK if `Update_One` has any postconditions. However, it will not be possible to use this postcondition to prove anything on the effect of `Update_All`.

5.9.8 Contracts for Subprogram Pointers

[Ada202X]

The upcoming standard of Ada allows adding contracts to access-to-subprogram types. As an example, here is a named access type `Update_Proc` with a contract:

```
type Update_Proc is not null access procedure (X : in out Natural) with
  Pre  => In_Range (X),
  Post => Relation (X'Old, X);
```

The Ada standard mandates that, when a subprogram designated by an access type with a contract is called, the contract is verified. Thus, it is possible for GNATprove to use this contract on indirect calls. For example, we can use `Update_Proc` as the type of the `Update_One` parameter of `Update_All`. As the call to `Update_One` now has a precondition, we should ensure before a call to `Update_All` that `In_Range` holds for all elements of `A`. We can also prove that `Relation` holds at every index of the array after the call:

```
procedure Update_All
  (A          : in out Nat_Array;
   Update_One : Update_Proc)
with Pre  => (for all E of A => In_Range (E)),
     Post => (for all I in A'Range => Relation (A'Old (I), A (I)))
is
begin
   for K in A'Range loop
      Update_One (A (K));
      pragma Loop_Invariant
        (for all I in A'First .. K => Relation (A'Loop_Entry (I), A (I)));
   end loop;
end Update_All;
```

As the contract of an access type is the only one which is known by GNATprove when checking indirect callers, SPARK requires that this contract is a valid approximation of the contract of every subprogram designated by an access objects of this type. More precisely, each time a value of a given access-to-subprogram type is created, GNATprove makes sur that:

- the precondition of the access-to-subprogram type if any (or the default precondition of True otherwise) is strong enough to imply the precondition of the designated subprogram, and
- the postcondition of the designated subprogram if any (or the default postcondition of True otherwise) is strong enough to imply the postcondition of the subprogram type.

Consider the four procedures below:

```
procedure Update_One_1 (X : in out Natural) with
  Pre  => In_Range (X),
  Post => Relation (X'Old, X);
--  Update_One_1 has exactly the same contract as Update_Proc

procedure Update_One_2 (X : in out Natural) with
  Post => Relation (X'Old, X) and Relation_2 (X'Old, X);
--  Update_Proc safely approximates Update_One_2:
--    * the precondition of Update_Proc is enough to ensure that Update_One_2 can␣
↪execute safely
--    * the postcondition of Update_One_2 implies the postcondition of Update_Proc
```

(continues on next page)

(continued from previous page)

```
procedure Update_One_3 (X : in out Natural) with
  Pre  => In_Range (X);
--  Does Relation hold after a call to Update_One_3?

procedure Update_One_4 (X : in out Natural) with
  Pre  => In_Range (X) and In_Range_2 (X),
  Post => Relation (X'Old, X);
--  Is it safe to call Update_One_4 when we do not check In_Range_2?
```

The procedures `Update_One_1` and `Update_One_2` can be designated by objects of type `Update_Proc`, as their contract can be safely approximated by the contract of `Update_Proc`. The procedures `Update_One_3` and `Update_One_4` on the other hand cannot. For example, if we try to use `Update_One_3` as a parameter of `Update_All`, GNATprove emits a check message stating that the postcondition of `Update_Proc` might not be valid after a call to `Update_One_3`:

```
procedure Test (A : in out Nat_Array) with
  Pre => (for all E of A => In_Range (E))
is
begin
   Update_All (A, Update_One_3'Access);
end Test;
```

```
medium: postcondition of source might not be strong enough to imply postcondition of␣
↪target, cannot prove Relation (X'Old, X)
```

Theoretically, a similar notion of approximation should be used for *Data Dependencies* and *Flow Dependencies* contracts. However, as these contracts are not currently allowed on access-to-subprogram types, SPARK simply disallows taking the Access attribute on a subprogram which has global inputs or outputs.

Note: Annotations specifying whether or not a subprogram returns are not available currently on access-to-subprogram types. As a result, all calls through dereferences are considered to possibly not terminate.

5.10 Concurrency and Ravenscar Profile

Concurrency in SPARK requires enabling the Ravenscar profile (see *Guide for the use of the Ada Ravenscar Profile in high integrity systems* by Alan Burns, Brian Dobbing, and Tullio Vardanega). This profile defines a subset of the Ada concurrency features suitable for hard real-time/embedded systems requiring stringent analysis, such as certification and safety analyses. In particular, it is concerned with determinism, analyzability, and memory-boundedness.

In addition to the subset defined by the Ravenscar profile, concurrency in SPARK also requires that tasks do not start executing before the program has been completely elaborated, which is expressed by setting pragma `Partition_Elaboration_Policy` to the value `Sequential`. Together with the requirement to apply the Ravenscar profile, this means that a concurrent SPARK program should define the following configuration pragmas, either in a configuration pragma file (see *Setting the Default SPARK_Mode* for an example of defining a configuration pragma file in your project file) or at the start of files:

```
pragma Profile (Ravenscar);
pragma Partition_Elaboration_Policy (Sequential);
```

GNATprove also supports the Jorvik profile, as defined in Ada 202X RM, D.13. To use this profile simply replace `Ravenscar` with `Jorvik` in the pragma `Profile` in the above code. The extended profile is intended for hard real-

time/embedded systems that may require schedulability analysis but not the most stringent analyses required for other domains.

In particular, to increase expressive power the Jorvik profile relaxes certain restrictions defined by the standard Ravenscar profile. Notably, these relaxed constraints allow multiple protected entries per protected object, multiple queued callers per entry, and more expressive protected entry barrier expressions. The profile also allows the use of relative delay statements in addition to the absolute delay statements allowed by Ravenscar. The two forms of delay statement are processed by GNATprove based on the type of their expression, as follows (absolute and relative delays, respectively):

- If the expression is of the type Ada.Real_Time.Time then for the purposes of determining global inputs and outputs the absolute delay statement is considered just like the relative delay statement, i.e., to reference the state abstraction Ada.Real_Time.Clock_Time as an input (see SPARK RM 9(17) for details).
- If the expression is of the type Ada.Calendar.Time then it is considered to reference the state abstraction Ada.Calendar.Clock_Time, which is defined similarly to Ada.Real_Time.Clock_Time but represents a different time base.

5.10.1 Tasks and Data Races

[Ravenscar/Jorvik]

Concurrent Ada programs are made of several *tasks*, that is, separate threads of control which share the same address space. In Ravenscar, only library-level, nonterminating tasks are allowed.

Task Types and Task Objects

Like ordinary objects, tasks have a type in Ada and can be stored in composite objects such as arrays and records. The definition of a task type looks like the definition of a subprogram. It is made of two parts: a declaration, usually empty as Ravenscar does not allow tasks to have entries (for task rendezvous), and a body containing the list of statements to be executed by objects of the task type. The body of nonterminating tasks (the only ones allowed in Ravenscar) usually takes the form of an infinite loop. For task objects of a given type to be parameterized, task types can have discriminants. As an example, a task type `Account_Management` can be declared as follows:

```
package Account is
   Num_Accounts : Natural := 0;

   task type Account_Management;
end Account;

package body Account is

   task body Account_Management is
   begin
      loop
         Get_Next_Account_Created;
         Num_Accounts := Num_Accounts + 1;
      end loop;
   end Account_Management;

end Account;
```

Then, tasks of type `Account_Management` can be created at library level, either as complete objects or as components of other objects:

```
package Bank is
   Special_Accounts : Account_Management;

   type Account_Type is (Regular, Premium, Selective);
   type Account_Array is array (Account_Type) of Account_Management;
   All_Accounts : Account_Array;
end Bank;
```

If only one object of a given task type is needed, then the task object can be declared directly giving a declaration and a body. An anonymous task type is then defined implicitly for the declared type object. For example, if we only need one task `Account_Management` then we can write:

```
package Account is
   Num_Accounts : Natural := 0;

   task Account_Management;
end Account;

package body Account is

   task body Account_Management is
   begin
      loop
         Get_Next_Account_Created;
         Num_Accounts := Num_Accounts + 1;
      end loop;
   end Account_Management;

end Account;
```

Preventing Data Races

In Ravenscar, communication between tasks can only be done through shared objects (tasks cannot communicate through rendezvous as task entries are not allowed in Ravenscar). In SPARK, the language is further restricted to avoid the possibility of erroneous concurrent access to shared data (a.k.a. data races). More precisely, tasks can only share *synchronized* objects, that is, objects that are protected against concurrent accesses. These include atomic objects, protected objects (see *Protected Objects and Deadlocks*), and suspension objects (see *Suspension Objects*). As an example, our previous definition of the `Account_Management` task type was not in SPARK. Indeed, data races could occur when accessing the global variable `Num_Accounts`, as detected by GNATprove:

```
account1.adb:15:39: medium: overflow check might fail, cannot prove upper bound for Num_
↪Accounts + 1
   15 |         Num_Accounts := Num_Accounts + 1;
      |                         ~~~~~~~~~~~~~~^~~
  reason for check: result of addition must fit in a 32-bits machine integer
  possible fix: loop at line 13 should mention Num_Accounts in a loop invariant
   13 |      loop
      |      ^ here

bank1.ads:5:04: high: possible data race when accessing variable "account1.num_accounts"
    5 |   Special_Accounts : Account_Management;
```

(continues on next page)

(continued from previous page)

```
      |   ^~~~~~~~~~~~~~~
  task "bank1.special_accounts" accesses "account1.num_accounts"
  task "bank1.all_accounts" accesses "account1.num_accounts"
```

To avoid this problem, shared variable `Num_Account` can be declared atomic:

```
package Account is
   Num_Accounts : Natural := 0 with Atomic;

   task type Account_Management;
end Account;
```

With this modification, GNATprove now alerts us that the increment of `Num_Account` is not legal, as a volatile variable (which is the case of atomic variables) cannot be read as a subexpression of a larger expression in SPARK:

```
account2.adb:15:26: error: volatile object in interfering context is not allowed in␣
↪SPARK (SPARK RM 7.1.3(9)) [E0004]
   15 |         Num_Accounts := Num_Accounts + 1;
      |                         ^~~~~~~~~~~~
  launch "gnatprove --explain=E0004" for more information
  violation of aspect SPARK_Mode at line 2
    2 |  SPARK_Mode
      |  ^ here
gnatprove: error during flow analysis and proof
```

This can be fixed by copying the current value of `Num_Account` in a temporary before the increment:

```
declare
   Tmp : constant Natural := Num_Accounts;
begin
   Num_Accounts := Tmp + 1;
end;
```

But note that even with that fix, there is no guarantee that `Num_Accounts` is incremented by one each time an account is created. Indeed, two tasks may read the same value of `Num_Accounts` and store this value in `Tmp` before both updating it to `Tmp + 1`. In such a case, two accounts have been created but `Num_Accounts` has been increased by 1 only. There is no *data race* in this program, which is confirmed by running GNATprove with no error, but there is by design a *race condition* on shared data that causes the program to malfunction. The correct way to fix this in SPARK is to use *Protected Types and Protected Objects*.

As they cannot cause data races, constants and variables that are constant after elaboration (see *Aspect Constant_After_Elaboration*) are considered as synchronized and can be accessed by multiple tasks. For example, we can declare a global constant `Max_Accounts` and use it inside `Account_Management` without risking data races:

```
package Account is
   Num_Accounts : Natural := 0 with Atomic;
   Max_Accounts : constant Natural := 100;

   task type Account_Management;
end Account;

package body Account is
```

(continues on next page)

(continued from previous page)

```
   task body Account_Management is
   begin
      loop
         Get_Next_Account_Created;
         declare
            Tmp : constant Natural := Num_Accounts;
         begin
            if Tmp < Max_Accounts then
               Num_Accounts := Tmp + 1;
            end if;
         end;
      end loop;
   end Account_Management;

end Account;
```

It is possible for a task to access an unsynchronized global variable only if this variable is declared in the same package as the task and if there is a single task accessing this variable. To allow this property to be statically verified, only tasks of an anonymous task type are allowed to access unsynchronized variables and the variables accessed should be declared to belong to the task using aspect `Part_Of`. Global variables declared to belong to a task are handled just like local variables of the task, that is, they can only be referenced from inside the task body. As an example, we can state that `Num_Accounts` is only accessed by the task object `Account_Management` in the following way:

```
package Account is
   task Account_Management;

   Num_Accounts : Natural := 0 with Part_Of => Account_Management;
end Account;
```

5.10.2 Task Contracts

[SPARK]

Dependency contracts can be specified on tasks. As tasks should not terminate in SPARK, such contracts specify the dependencies between outputs and inputs of the task *updated while the task runs*:

- The *data dependencies* introduced by aspect `Global` specify the global data read and written by the task.
- The *flow dependencies* introduced by aspect `Depends` specify how task outputs depend on task inputs.

This is a difference between tasks and subprograms, for which such contracts describe the dependencies between outputs and inputs *when the subprogram returns*.

Data Dependencies on Tasks

Data dependencies on tasks follow the same syntax as the ones on subprograms (see *Data Dependencies*). For example, data dependencies can be specified for task (type or object) `Account_Management` as follows:

```
package Account is
   Num_Accounts : Natural := 0 with Atomic;

   task type Account_Management with
     Global => (In_Out => Num_Accounts);
end Account;
```

Flow Dependencies on Tasks

Flow dependencies on tasks follow the same syntax as the ones on subprograms (see *Flow Dependencies*). For example, flow dependencies can be specified for task (type or object) `Account_Management` as follows:

```
package Account is
   Num_Accounts : Natural := 0 with Atomic;

   task type Account_Management with
     Depends => (Account_Management => Account_Management,
                 Num_Accounts       => Num_Accounts);
end Account;
```

Notice that the task unit itself is both an input and an output of the task:

- It is an input because task discriminants (if any) and task attributes may be read in the task body.
- It is an output so that the task unit may be passed as in out parameter in a subprogram call. But note that the task object cannot be modified once created.

The dependency of the task on itself can be left implicit as well, as follows:

```
package Account is
   Num_Accounts : Natural := 0 with Atomic;

   task type Account_Management with
     Depends => (Num_Accounts => Num_Accounts);
end Account;
```

5.10.3 Protected Objects and Deadlocks

[Ravenscar/Jorvik]

In Ada, protected objects are used to encapsulate shared data and protect it against data races (low-level unprotected concurrent access to data) and race conditions (lack of proper synchronization between reads and writes of shared data). They coordinate access to the protected data guaranteeing that read-write accesses are always exclusive while allowing concurrent read-only accesses. In Ravenscar, only library-level protected objects are allowed.

Protected Types and Protected Objects

Definitions of protected types resemble package definitions. They are made of two parts, a declaration (divided into a public part and a private part) and a body. The public part of a protected type's declaration contains the declarations of the subprograms that can be used to access the data declared in its private part. The body of these subprograms are located in the protected type's body. In Ravenscar, protected objects should be declared at library level, either as complete objects or as components of other objects. As an example, here is how a protected type can be used to coordinate concurrent accesses to the global variable `Num_Accounts`:

```
package Account is

   protected type Protected_Natural is
      procedure Incr;
      function Get return Natural;
   private
      The_Data : Natural := 0;
   end Protected_Natural;

   Num_Accounts : Protected_Natural;
   Max_Accounts : constant Natural := 100;

   task type Account_Management;
end Account;

package body Account is

   protected body Protected_Natural is
      procedure Incr is
      begin
         The_Data := The_Data + 1;
      end Incr;

      function Get return Natural is (The_Data);
   end Protected_Natural;

   task body Account_Management is
   begin
      loop
         Get_Next_Account_Created;
         if Num_Accounts.Get < Max_Accounts then
            Num_Accounts.Incr;
         end if;
      end loop;
   end Account_Management;

end Account;
```

Contrary to the previous version using an atomic global variable (see *Preventing Data Races*), this version prevents also any race condition when incrementing the value of `Num_Accounts`. But note that there is still a possible race condition between the time the value of `Num_Accounts` is read and checked to be less than `Max_Accounts` and the time it is incremented. So this version does not guarantee that `Num_Accounts` stays below `Max_Accounts`. The correct way to fix this in SPARK is to use protected entries (see *Protected Subprograms*).

Note that, in SPARK, to avoid initialization issues on protected objects, both private variables and variables belonging to a protected object must be initialized at declaration (either explicitly or through default initialization).

Just like for tasks, it is possible to directly declare a protected object if it is the only one of its type. In this case, an anonymous protected type is implicitly declared for it. For example, if `Num_Account` is the only `Protected_Natural` we need, we can directly declare:

```
package Account is

   protected Num_Accounts is
      procedure Incr;
      function Get return Natural;
   private
      The_Data : Natural := 0;
   end Num_Accounts;

end Account;

package body Account is

   protected body Num_Accounts is
      procedure Incr is
      begin
         The_Data := The_Data + 1;
      end Incr;

      function Get return Natural is (The_Data);
   end Num_Accounts;

end Account;
```

Protected Subprograms

The access mode granted by protected subprograms depends on their kind:

- Protected procedures provide exclusive read-write access to the private data of a protected object.
- Protected functions offer concurrent read-only access to the private data of a protected object.
- Protected *entries* are conceptually procedures with a *barrier*. When an entry is called, the caller waits until the condition of the barrier is true to be able to access the protected object.

So that scheduling is deterministic, Ravenscar requires that at most one entry is specified in a protected unit and at most one task is waiting on a given entry at every time. To ensure this, GNATprove checks that no two tasks can call the same protected object's entry. As an example, we could replace the procedure `Incr` of `Protected_Natural` to wait until `The_Data` is smaller than `Max_Accounts` before incrementing it. As only simple Boolean variables are allowed as entry barriers in Ravenscar, we add such a Boolean flag `Not_Full` as a component of the protected object:

```
package Account is

   protected type Protected_Natural is
      entry Incr;
      function Get return Natural;
   private
      The_Data : Natural := 0;
      Not_Full : Boolean := True;
   end Protected_Natural;
```

(continues on next page)

(continued from previous page)

```
   Num_Accounts : Protected_Natural;
   Max_Accounts : constant Natural := 100;

   task type Account_Management;
end Account;

package body Account is

   protected body Protected_Natural is
      entry Incr when Not_Full is
      begin
         The_Data := The_Data + 1;
         if The_Data = Max_Accounts then
            Not_Full := False;
         end if;
      end Incr;

      function Get return Natural is (The_Data);
   end Protected_Natural;

   task body Account_Management is
   begin
      loop
         Get_Next_Account_Created;
         Num_Accounts.Incr;
      end loop;
   end Account_Management;

end Account;
```

This version fixes the remaining race condition on this example, thus ensuring that every new account created bumps the value of `Num_Accounts` by 1, and that `Num_Accounts` stays below `Max_Accounts`.

To avoid data races, protected subprograms should not access unsynchronized objects (see *Tasks and Data Races*). Like for tasks, it is still possible for subprograms of a protected object of an anonymous protected type to access an unsynchronized object declared in the same package as long as it is not accessed by any task or subprogram from other protected objects. In this case, the unsynchronized object should have a `Part_Of` aspect referring to the protected object. It is then handled as if it was a private variable of the protected object. This is typically done so that the address in memory of the variable can be specified, using either aspect `Address` or a corresponding representation clause. Here is how this could be done with `Num_Account`:

```
package Account is
   protected Protected_Num_Accounts is
      procedure Incr;
      function Get return Natural;
   end Protected_Num_Accounts;

   Num_Accounts : Natural := 0 with
     Part_Of => Protected_Num_Accounts,
     Address => ...
end Account;
```

As it can prevent access to a protected object for an unbounded amount of time, a task should not be blocked or delayed while inside a protected subprogram. Actions that can block a task are said to be *potentially blocking*. For example, calling a protected entry, explicitly waiting using a `delay_until` statement (note that `delay` statements are forbidden in Ravenscar), or suspending on a suspension object (see *Suspension Objects*) are potentially blocking actions. In Ada, it is an error to do a potentially blocking action while inside a protected subprogram. Note that a call to a function or a procedure on another protected object is not considered to be potentially blocking. Indeed, such a call cannot block a task in the absence of deadlocks (which is enforced in Ravenscar using the priority ceiling protocol, see *Avoiding Deadlocks and Priority Ceiling Protocol*).

GNATprove verifies that no potentially blocking action is performed from inside a protected subprogram in a modular way on a per subprogram basis. Thus, if a subprogram can perform a potentially blocking operation, every call to this subprogram from inside a protected subprogram will be flagged as a potential error. As an example, the procedure Incr_Num_Accounts is potentially blocking and thus should not be called, directly or indirectly, from a protected subprogram:

```
package Account is

   protected type Protected_Natural is
      entry Incr;
   private
      The_Data : Natural := 0;
   end Protected_Natural;

   Num_Accounts : Protected_Natural;

   procedure Incr_Num_Accounts;

end Account;

package body Account is

   procedure Incr_Num_Accounts is
   begin
      Num_Accounts.Incr;
   end Incr_Num_Accounts;

end Account;
```

Avoiding Deadlocks and Priority Ceiling Protocol

To ensure exclusivity of read-write accesses, when a procedure or an entry of a protected object is called, the protected object is locked so that no other task can access it, be it in a read-write or a read-only mode. In the same way, when a protected function is called, no other task can access the protected object in read-write mode. A *deadlock* happens when two or more tasks are unable to run because each of them is trying to access a protected object that is currently locked by another task.

To ensure absence of deadlocks on a single core, Ravenscar requires the use of the Priority Ceiling Protocol. This protocol ensures that no task can be blocked trying to access a protected object locked by another task. It relies on task's *priorities*. The priority of a task is a number encoding its urgency. On a single core, scheduling ensures that the current running task can only be preempted by another task if it has a higher priority. Using this property, the Priority Ceiling Protocol works by increasing the priorities of tasks accessing a protected object to a priority that is at least as high as the priorities of other tasks accessing this object. This ensures that, while holding a lock, the currently running task cannot be preempted by a task which could later be blocked by this lock.

To enforce this protocol, every task is associated with a *base priority*, either given at declaration using the `Priority` aspect or defaulted. This base priority is static and cannot be modified after the task's declaration. A task also has an *active priority* which is initially the task's base priority but will be increased when the task enters a protected action. For example, we can set the base priority of `Account_Management` to 5 at declaration:

```
package Account is
   task type Account_Management with Priority => 5;
end Account;
```

Likewise, each protected object is associated at declaration with a *ceiling priority* which should be equal or higher than the active priority of any task accessing it. The ceiling priority of a protected object does not need to be static, it can be set using a discriminant for example. Still, like for tasks, Ravenscar requires that it is set once and for all at the object's declaration and cannot be changed afterwards. As an example, let us attach a ceiling priority to the protected object `Num_Accounts`. As `Num_Accounts` will be used by `Account_Management`, its ceiling priority should be no lower than 5:

```
package Account is

   protected Num_Accounts with Priority => 7 is
      procedure Incr;
      function Get return Natural;
   private
      The_Data : Natural := 0;
   end Num_Accounts;

   task type Account_Management with Priority => 5;

end Account;
```

5.10.4 Suspension Objects

[Ravenscar/Jorvik]

The language-defined package `Ada.Synchronous_Task_Control` provides a type for semaphores called *suspension objects*. They allow lighter synchronization mechanisms than protected objects (see *Protected Objects and Deadlocks*). More precisely, a suspension object has a Boolean state which can be set atomically to True using the `Set_True` procedure. When a task suspends on a suspension object calling the `Suspend_Until_True` procedure, it is blocked until the state of the suspension object is True. At that point, the state of the suspension object is set back to False and the task is unblocked. Note that `Suspend_Until_True` is potentially blocking and therefore should not be called directly or indirectly from within *Protected Subprograms*. In the following example, the suspension object `Semaphore` is used to make sure `T1` has initialized the shared data by the time `T2` begins processing it:

```
Semaphore : Suspension_Object;
task T1;
task T2;

task body T1 is
begin
  Initialize_Shared_Data;
  Set_True (Semaphore);
  loop
    ...
  end loop;
```

(continues on next page)

(continued from previous page)

```
end T1;

task body T2 is
begin
  Suspend_Until_True (Semaphore);
  loop
    ...
  end loop;
end T2;
```

In Ada, an exception is raised if a task tries to suspend on a suspension object on which another task is already waiting on that same suspension object. Like for verifying that no two tasks can be queued on a protected entry, this verification is done by GNATprove by checking that no two tasks ever suspend on the same suspension object. In the following example, the suspension objects `Semaphore1` and `Semaphore2` are used to ensure that `T1` and `T2` never call `Enter_Protected_Region` at the same time. GNATprove will successfully verify that only one task can suspend on each suspension object:

```
Semaphore1, Semaphore2 : Suspension_Object;
task T1;
task T2;

task body T1 is
begin
  loop
    Suspend_Until_True (Semaphore1);
    Enter_Protected_Region;
    Set_True (Semaphore2);
  end loop;
end T1;

task body T2 is
begin
  loop
    Suspend_Until_True (Semaphore2);
    Enter_Protected_Region;
    Set_True (Semaphore1);
  end loop;
end T2;
```

5.10.5 State Abstraction and Concurrency

[SPARK]

Protected objects, as well as suspension objects, are *effectively volatile* which means that their value as seen from a given task may change at any time due to some other task accessing the protected object or suspension object. If they are part of a state abstraction, the volatility of the abstract state must be specified by using the `External` aspect (see *External State Abstraction*). Note that task objects, though they can be part of a package's hidden state, are not effectively volatile and can therefore be components of normal state abstractions. For example, the package `Synchronous_Abstractions` defines two abstract states, one for external objects, containing the atomic variable `V`, the suspension object `S`, and the protected object `P`, and one for normal objects, containing the task `T`:

```
package Synchronous_Abstractions with
  Abstract_State => (Normal_State, (Synchronous_State with External))
is
end Synchronous_Abstractions;

package body Synchronous_Abstractions with
  Refined_State => (Synchronous_State => (P,V,S), Normal_State => T)
is
  task T;

  S : Suspension_Object;

  V : Natural := 0 with Atomic, Async_Readers, Async_Writers;

  protected P is
    function Read return Natural;
  private
    V : Natural := 0;
  end P;

  protected body P is
    function Read return Natural is (V);
  end P;

  task body T is ...
end  Synchronous_Abstractions;
```

To avoid data races, task bodies, as well as protected subprograms, should only access synchronized objects (see *Preventing Data Races*). State abstractions containing only synchronized objects can be specified to be synchronized using the `Synchronous` aspect. Only synchronized state abstractions can be accessed from task bodies and protected subprograms. For example, if we want the procedure `Do_Something` to be callable from the task `Use_Synchronized_State`, then the state abstraction `Synchronous_State` must be annotated using the `Synchronous` aspect:

```
package Synchronous_Abstractions with
  Abstract_State => (Normal_State,
                     (Synchronous_State with Synchronous, External))
is
  procedure Do_Something with Global => (In_Out => Synchronous_State);
end Synchronous_Abstractions;

task body Use_Synchronized_State is
begin
  loop
    Synchronous_Abstractions.Do_Something;
  end loop;
end Use_Synchronized_State;
```

5.10.6 Project-wide Tasking Analysis

Tasking-related analysis, as currently implemented in GNATprove, is subject to two limitations:

First, the analysis is always done when processing a source file with task objects or with a subprogram that can be used as a main subprogram of a partition (i.e. is at library level, has no parameters, and is either a procedure or a function returning an integer type).

In effect, you might get spurious checks when:

- a subprogram satisfies conditions for being used as a main subprogram of a partition but is not really used that way, i.e. it is not specified in the Main attribute of the GNAT project file you use to build executables, and
- it "withs" or is "withed" (directly or indirectly) from a library-level package that declares some task object, and
- both the fake "main" subprogram and the task object access the same resource in a way that violates tasking-related rules (e.g. suspends on the same suspension object).

As a workaround, either wrap the fake "main" subprogram in a library-level package or give it a dummy parameter.

Second, the analysis is only done in the context of all the units "withed" (directly and indirectly) by the currently analyzed source file.

In effect, you might miss checks when:

- building a library that declares tasks objects in unrelated source files, i.e. files that are never "withed" (directly or indirectly) from the same file, and those tasks objects access the same resource in a way that violates tasking-related rules, or
- using a library that internally declares some tasks objects, they access some tasking-sensitive resource, and your main subprogram also accesses this resource.

As a workaround, when building library projects add a dummy main subprogram that "withs" all the library-level packages of your project.

5.10.7 Interrupt Handlers

SPARK puts no restrictions on the Ada interrupt handling and GNATprove merely checks that interrupt handlers will be safely executed. In Ada interrupts handlers are defined by annotating protected procedures, for example:

```
with Ada.Interrupts.Names; use Ada.Interrupts.Names;

protected P is
   procedure Signal with Attach_Handler => SIGINT;
end P;
```

Currently GNATprove emits a check for each handler declaration saying that the corresponding interrupt might be already reserved. In particular, it might be reserved by either the system or the Ada runtime; see GNAT pragmas Interrupt_State and Unreserve_All_Interrupts for details. Once examined, those checks can be suppressed with pragma Annotate.

If pragma Priority or Interrupt_Priority is explicitly specified for a protected type, then GNATprove will check that its value is in the range of the System.Any_Priority or System.Interrupt_Priority, respectively; see Ada RM D.3(6.1/3).

For interrupt handlers whose bodies are annotated with SPARK_Mode => On, GNATprove will additionally check that:

- the interrupt handler does not call (directly or indirectly) the Ada.Task_Identification.Current_Task routine, which might cause a Program_Error runtime exception; see Ada RM C.7.1(17/3);
- all global objects read (either as an Input or a Proof_In) by the interrupt handler are initialized at elaboration;

- there are no unsynchronized objects accessed both by the interrupt handler and by some task (or by some other interrupt handler);
- there are no protected objects locked both by the interrupt handler and by some task (or by some other interrupt handler).

5.11 SPARK Libraries

The units described here have their spec in SPARK (with `SPARK_Mode => On` specified on the spec), more rarely their body in SPARK as well.

Subprograms in these units fall into one of the following categories:

- Subprograms which should always return without error or exception if their precondition is respected.
- Procedures marked with the annotation `Exceptional_Cases`. This corresponds to the possibility of exception in the procedure, even when its precondition is respected.
- Functions marked with `SPARK_Mode => Off` which cannot be called from SPARK code.

5.11.1 SPARK Library

As part of the SPARK product, several libraries are available through the project file `<spark-install>/lib/gnat/sparklib.gpr` (or through the project file `<spark-install>/lib/gnat/sparklib_light.gpr` in an environment without units `Ada.Numerics.Big_Numbers.Big_Integers` and `Ada.Numerics.Big_Numbers.Big_Reals`). Header files of the SPARK library are available through *Help* → *SPARK* → *SPARKlib* menu item in GNAT Studio. To use this library in a program, you need to copy the project template that corresponds to your runtime (either `sparklib.gpr` or `sparklib_light.gpr`) and adapt it by providing appropriate values for the object directory (attribute `Object_Dir` in the project file) and the list of excluded source files (attribute `Excluded_Source_Files` in the project file). The simplest is just to provide a value for `Object_Dir` and inherit `Excluded_Source_Files` from the parent project:

```
project SPARKlib extends "sparklib_external" is
   for Object_Dir use "sparklib_obj";
   for Source_Dirs use SPARKlib_External'Source_Dirs;
   for Excluded_Source_Files use SPARKlib_External'Excluded_Source_Files;
end SPARKlib;
```

Then, add a corresponding dependency in your project file, for example:

```
with "sparklib";
project My_Project is
   ...
end My_Project;
```

You may need to update the environment variable `GPR_PROJECT_PATH` for the lemma library project to be found by GNAT compiler, as described in *Installation of GNATprove*.

Finally, if you instantiate in your code a generic from the SPARK library, you may also need to pass `-gnateDSPARK_BODY_MODE=Off` as a compilation switch for the units with these instantiations.

5.11.2 Big Numbers Library

Annotations such as preconditions, postconditions, assertions, loop invariants, are analyzed by GNATprove with the exact same meaning that they have during execution. In particular, evaluating the expressions in an annotation may raise a run-time error, in which case GNATprove will attempt to prove that this error cannot occur, and report a warning otherwise.

In SPARK, scalar types such as integer and floating point types are bounded machine types, so arithmetic computations over them can lead to overflows when the result does not fit in the bounds of the type used to hold it. In some cases, it is convenient to express properties in annotations as they would be expressed in mathematics, where quantities are unbounded, for example:

```
function Add (X, Y : Integer) return Integer with
  Pre  => X + Y in Integer,
  Post => Add'Result = X + Y;
```

The precondition of `Add` states that the result of adding its two parameters should fit in type `Integer`. Unfortunately, evaluating this expression will fail an overflow check, because the result of `X + Y` is stored in a temporary of type `Integer`.

To alleviate this issue, it is possible to use the standard library for big numbers. It contains support for:

- Unbounded integers in `SPARK.Big_Integers`.
- Unbounded rational numbers in `SPARK.Big_Reals`.

Theses libraries define representations for big numbers and basic arithmetic operations over them, as well as conversions from bounded scalar types such as floating point numbers or integer types. Conversion from an integer to a big integer is provided by:

- function `To_Big_Integer` in `SPARK.Big_Integers` for type `Integer`
- function `To_Big_Integer` in generic package `Signed_Conversions` in `SPARK.Big_Integers` for all other signed integer types
- function `To_Big_Integer` in generic package `Unsigned_Conversions` in `SPARK.Big_Integers` for modular integer types

Similarly, the same packages define a function `From_Big_Integer` to convert from a big integer to an integer. A function `To_Real` in `SPARK.Big_Reals` converts from type `Integer` to a big real and function `To_Big_Real` in the same package converts from a big integer to a big real.

Though these operations do not have postconditions, they are interpreted by GNATprove as the equivalent operations on mathematical integers and real numbers. This allows to benefit from precise support on code using them. Note that the corresponding Ada libraries `Ada.Numerics.Big_Numbers.Big_Integers` and `Ada.Numerics.Big_Numbers.Big_Reals` will be handled in the same way, but might be not available under specific runtimes. It is preferable to use the units from the SPARK library instead, or use `Ada.Numerics.Big_Numbers.Big_Integer_Ghost`.

Note: Some functionality of the library are not precisely supported. This includes in particular conversions to and from strings, conversions of `Big_Real` to fixed-point or floating-point types, and `Numerator` and `Denominator` functions.

The big number library can be used both in annotations and in actual code, as it is executable, though of course, using it in production code means incurring its runtime costs. It can be considered a good trade-off to only use it in contracts, if they are disabled in production builds. For example, we can rewrite the precondition of our `Add` function with big integers to avoid overflows:

```
function Add (X, Y : Integer) return Integer with
  Pre  => In_Range (To_Big_Integer (X) + To_Big_Integer (Y),
                    Low  => To_Big_Integer (Integer'First),
                    High => To_Big_Integer (Integer'Last)),
  Post => Add'Result = X + Y;
```

As a more advanced example, it is also possible to introduce a ghost model for numerical computations on floating point numbers as a mathematical real number so as to be able to express properties about rounding errors. In the following snippet, we use the ghost variable `M` as a model of the floating point variable `Y`, so we can assert that the result of our floating point calculations are not too far from the result of the same computations on real numbers.

```
declare
   package Float_Convs is new Float_Conversions (Num => Float);
   --  Introduce conversions to and from values of type Float

   subtype Small_Float is Float range -100.0 .. 100.0;

   function Init return Small_Float with Import;
   --  Unknown initial value of the computation

   X : constant Small_Float := Init;
   Y : Float := X;
   M : Big_Real := Float_Convs.To_Big_Real (X) with Ghost;
   --  M is used to mimic the computations done on Y on real numbers

begin
   Y := Y * 100.0;
   M := M * Float_Convs.To_Big_Real (100.0);
   Y := Y + 100.0;
   M := M + Float_Convs.To_Big_Real (100.0);

   pragma Assert
     (In_Range (Float_Convs.To_Big_Real (Y) - M,
                Low  => Float_Convs.To_Big_Real (- 0.001),
                High => Float_Convs.To_Big_Real (0.001)));
   --  The rounding errors introduced by the floating-point computations
   --  are not too big.
end;
```

5.11.3 Functional Containers Library

To model complex data structures, one often needs simpler, mathematical like containers. The mathematical containers provided in the SPARK library (see the *SPARK Library*) are unbounded and may contain indefinite elements. However, they are controlled and thus not usable in every context. So that these containers can be used safely, we have made them functional, that is, no primitives are provided which would allow modifying an existing container. Instead, their API features functions creating new containers from existing ones. As an example, functional containers provide no `Insert` procedure but rather a function `Add` which creates a new container with one more element than its parameter:

```
function Add (C : Container; E : Element_Type) return Container;
```

As a consequence, these containers are highly inefficient. Thus, they should in general be used in ghost code and annotations so that they can be removed from the final executable.

There are 5 functional containers, which are part of the SPARK library:

- `SPARK.Containers.Functional.Infinite_Sequences`
- `SPARK.Containers.Functional.Maps`
- `SPARK.Containers.Functional.Multisets`
- `SPARK.Containers.Functional.Sets`
- `SPARK.Containers.Functional.Vectors`

Sequences defined in `Functional.Vectors` are no more than ordered collections of elements. In an Ada like manner, the user can choose the range used to index the elements:

```
function Length (S : Sequence) return Count_Type;
function Get (S : Sequence; N : Index_Type) return Element_Type;
```

The sequences defined in `Functional.Infinite_Sequences` behave as the one of `Functional.Vectors`. The difference between them lies in the fact that the inifinte one are indexed by mathematical integers.

```
function Length (Container : Sequence) return Big_Natural;
function Get (Container : Sequence; Position  : Big_Integer) return Element_Type;
```

Functional sets offer standard mathematical set functionalities such as inclusion, union, and intersection. They are neither ordered nor hashed:

```
function Contains (S : Set; E : Element_Type) return Boolean;
function "<=" (Left : Set; Right : Set) return Boolean;
```

Functional maps offer a dictionary between any two types of elements:

```
function Has_Key (M : Map; K : Key_Type) return Boolean;
function Get (M : Map; K : Key_Type) return Element_Type;
```

Multisets are mathematical sets associated with a number of occurrences:

```
function Nb_Occurence (S : Multiset; E : Element_Type) return Big_Natural;
function Cardinality (S : Multiset) return Big_Natural;
```

Each functional container type supports quantification over its elements (or keys for functional maps).

These containers can easily be used to model user defined data structures. They were used to this end to annotate and verify a package of allocators (see the *allocators* example provided with a SPARK installation). In this example, an allocator featuring a free list implemented in an array is modeled by a record containing a set of allocated resources and a sequence of available resources:

```
type Status is (Available, Allocated);
type Cell is record
   Stat : Status;
   Next : Resource;
end record;
type Allocator is array (Valid_Resource) of Cell;
type Model is record
   Available : Sequence;
   Allocated : Set;
end record;
```

Note: Instances of container packages, both functional and formal, are subject to particular constraints which are necessary for the contracts on the instance to be correct. For example, container primitives don't comply with the ownership policy of SPARK if element or key types are ownership types. These constraints are verified specifically each time a container package is instantiated. For some of these checks, it is possible for the user to help the proof tool by providing some lemmas at instantiation. It is the case in particular for the constraints coming from the Ada reference manual on the container packages (that "=" is an equivalence relation, or that "<" is a strict weak order in particular). These lemmas appear in the library as additional ghost generic formal parameters.

Note: Functional sets, maps and multisets operate with a user-provided equivalence relation, which might be different from the logical equality. In this case, all elements or keys of an equivalence class are removed or included together in the container. This can sometimes have surprising results. For example, `Contains` can return `True` if an equivalent (but not equal) element has been added to a set. Similarly, the quantified expression `for Some E of S => Cond (E)` might be proved if Cond is `False` for all elements that were explicitely added to the set, but `True` for an object equivalent to such an element.

The functional sets, maps, sequences, and vectors have child packages providing higher order functions:

- `SPARK.Containers.Functional.Infinite_Sequences.Higher_Order`
- `SPARK.Containers.Functional.Maps.Higher_Order`
- `SPARK.Containers.Functional.Sets.Higher_Order`
- `SPARK.Containers.Functional.Vectors.Higher_Order`

These functions take as parameters access-to-functions that compute some information about an element of the container and apply it to all elements in a generic way. As an example, here is the function `Count` for functional sets. It counts the number of elements in the set with a given property. The property is provided by its input access-to-function parameter `Test`:

```
function Count
  (S    : Set;
   Test : not null access function (E : Element_Type) return Boolean)
   return Big_Natural
--  Count the number of elements on which the input Test function returns
--  True. Count can only be used with Test functions which return the same
--  value on equivalent elements.

with
  Global   => null,
  Annotate => (GNATprove, Higher_Order_Specialization),
  Pre      => Eq_Compatible (S, Test),
  Post     => Count'Result <= Length (S);
```

All the higher order functions are annotated with `Higher_Order_Specialization` (see *Annotation for Handling Specially Higher Order Functions*) so they can be used even with functions which read global data as parameters.

5.11.4 Formal Containers Library

Containers are generic data structures offering a high-level view of collections of objects, while guaranteeing fast access to their content to retrieve or modify it. The most common containers are lists, vectors, sets and maps, which are defined as generic units in the Ada Standard Library. In critical software where verification objectives severely restrict the use of pointers, containers offer an attractive alternative to pointer-intensive data structures.

The Ada Standard Library defines two kinds of containers:

- The controlled containers using dynamic allocation, for example `SPARK.Containers.Formal.Unbounded_Sets`. They define containers as controlled tagged types, so that memory for the container is automatic reallocated during assignment and automatically freed when the container object's scope ends.
- The bounded containers not using dynamic allocation, for example `SPARK.Containers.Fromal.Vectors`. They define containers as discriminated tagged types, so that the memory for the container can be reserved at initialization.

Although bounded containers are better suited to critical software development, neither controlled containers nor bounded containers can be used in SPARK, because their API does not lend itself to adding suitable contracts (in particular preconditions) ensuring correct usage in client code.

The formal containers are a variation of the standard containers with API changes that allow adding suitable contracts, so that GNATprove can prove that client code manipulates containers correctly. There are 12 formal containers, which are part of the SPARK library.

Among them, 6 are bounded and definite:

- `SPARK.Containers.Formal.Vectors`
- `SPARK.Containers.Formal.Doubly_Linked_Lists`
- `SPARK.Containers.Formal.Hashed_Sets`
- `SPARK.Containers.Formal.Ordered_Sets`
- `SPARK.Containers.Formal.Hashed_Maps`
- `SPARK.Containers.Formal.Ordered_Maps`

The 6 others are unbounded and indefinite but are controlled:

- `SPARK.Containers.Formal.Unbounded_Vectors`
- `SPARK.Containers.Formal.Unbounded_Doubly_Linked_Lists`
- `SPARK.Containers.Formal.Unbounded_Hashed_Sets`
- `SPARK.Containers.Formal.Unbounded_Ordered_Sets`
- `SPARK.Containers.Formal.Unbounded_Hashed_Maps`
- `SPARK.Containers.Formal.Unbounded_Ordered_Maps`

Bounded definite formal containers can only contain definite objects (objects for which the compiler can compute the size in memory, hence not `String` nor `T'Class`). They do not use dynamic allocation. In particular, they cannot grow beyond the bound defined at object creation.

Unbounded indefinite formal containers can contain indefinite objects. They use dynamic allocation both to allocate memory for their elements, and to expand their internal block of memory when it is full.

Note: The capacity of unbounded containers is not set using a discriminant. Instead, it is implicitly set to it maximum value. All the required memory is not reserved at declaration. As all the formal containers are internally indexed by `Count_Type`, their maximum size is `Count_Type'Last`.

Modified API of Formal Containers

The visible specification of formal containers is in SPARK, with suitable contracts on subprograms to ensure correct usage, while their private part and implementation is not in SPARK. Hence, GNATprove can be used to prove correct usage of formal containers in client code, but not to prove that formal containers implement their specification.

Procedures `Update_Element` or `Query_Element` that iterate over a container are not defined on formal containers. The effect of these procedures could not be precisely described in their contracts as there is no way to refer to the contract of their access-to-procedure parameter.

Procedures and functions that query the content of a container take the container in parameter. For example, function `Has_Element` that queries if a container has an element at a given position is declared as follows:

```
function Has_Element (Container : T; Position : Cursor) return Boolean;
```

This is different from the API of standard containers, where it is sufficient to pass a cursor to these subprograms, as the cursor holds a reference to the underlying container:

```
function Has_Element (Position : Cursor) return Boolean;
```

Cursors of formal containers do not hold a reference to a specific container, as this would otherwise introduce aliasing between container and cursor variables, which is not supported in SPARK. See *Absence of Interferences*. As a result, the same cursor can be applied to multiple container objects.

For each container type, the library provides model functions that are used to annotate subprograms from the API. The different models supply different levels of abstraction of the container's functionalities. These model functions are grouped in *Ghost Packages* named `Formal_Model`.

The higher level view of a container is usually the mathematical structure of element it represents. We use a sequence for ordered containers such as lists and vectors and a mathematical map for imperative maps. This allows us to specify the effects of a subprogram in a very high level way, not having to consider cursors nor order of elements in a map:

```
procedure Increment_All (L : in out List) with
  Post =>
    (for all N in 1 .. Length (L) =>
       Element (Model (L), N) = Element (Model (L)'Old, N) + 1);

procedure Increment_All (S : in out Map) with
  Post =>
    (for all K of Model (S)'Old => Has_Key (Model (S), K))
       and
    (for all K of Model (S) =>
       Has_Key (Model (S)'Old, K)
         and Get (Model (S), K) = Get (Model (S)'Old, K) + 1);
```

For sets and maps, there is a lower level model representing the underlying order used for iteration in the container, as well as the actual values of elements/keys. It is a sequence of elements/keys. We can use it if we want to specify in `Increment_All` on maps that the order and actual values of keys are preserved:

```
procedure Increment_All (S : in out Map) with
  Post =>
    Keys (S) = Keys (S)'Old
      and
    (for all K of Model (S) =>
       Get (Model (S), K) = Get (Model (S)'Old, K) + 1);
```

Finally, cursors are modeled using a functional map linking them to their position in the container. For example, we can state that the positions of cursors in a list are not modified by a call to `Increment_All`:

```
procedure Increment_All (L : in out List) with
  Post =>
    Positions (L) = Positions (L)'Old
      and
    (for all N in 1 .. Length (L) =>
       Element (Model (L), N) = Element (Model (L)'Old, N) + 1);
```

Switching between the different levels of model functions allows to express precise considerations when needed without polluting upper level specifications. For example, consider a variant of the `List.Find` function defined in the API of formal containers, which returns a cursor holding the value searched if there is one, and the special cursor `No_Element` otherwise:

```
with Element_Lists; use Element_Lists; use Element_Lists.Lists;
with Ada.Containers; use Ada.Containers; use Element_Lists.Lists.Formal_Model;

function My_Find (L : List; E : Element_Type) return Cursor with
  SPARK_Mode,
  Contract_Cases =>
    (Contains (L, E)     => Has_Element (L, My_Find'Result) and then
                            Element (L, My_Find'Result) = E,
     not Contains (L, E) => My_Find'Result = No_Element);
```

The ghost functions mentioned above are specially useful in *Loop Invariants* to refer to cursors, and positions of elements in the containers. For example, here, ghost function `Positions` is used in the loop invariant to query the position of the current cursor in the list, and `Model` is used to specify that the value searched is not contained in the part of the container already traversed (otherwise the loop would have exited):

```
function My_Find (L : List; E : Element_Type) return Cursor with
  SPARK_Mode
is
   Cu : Cursor := First (L);

begin
   while Has_Element (L, Cu) loop
      pragma Loop_Variant (Increases => P.Get (Positions (L), Cu));
      pragma Loop_Invariant (for all I in 1 .. P.Get (Positions (L), Cu) - 1 =>
                                Element (Model (L), I) /= E);

      if Element (L, Cu) = E then
         return Cu;
      end if;

      Next (L, Cu);
   end loop;

   return No_Element;
end My_Find;
```

GNATprove proves that function `My_Find` implements its specification:

```
my_find.adb:8:28: info: loop variant proved
```

(continues on next page)

(continued from previous page)

```
my_find.adb:8:42: info: precondition proved
my_find.adb:9:30: info: loop invariant initialization proved
my_find.adb:9:30: info: loop invariant preservation proved
my_find.adb:9:49: info: precondition proved
my_find.adb:10:33: info: precondition proved
my_find.adb:12:10: info: precondition proved
my_find.adb:16:07: info: precondition proved
my_find.ads:4:10: info: implicit aspect Always_Terminates on "My_Find" has been proved,␣
↪subprogram will terminate
my_find.ads:6:03: info: disjoint contract cases proved
my_find.ads:6:03: info: complete contract cases proved
my_find.ads:7:26: info: contract case proved
my_find.ads:8:29: info: precondition proved
my_find.ads:9:26: info: contract case proved
```

Note: Just like functional containers, the formal containers do not comply with the ownership policy of SPARK if element or key types are ownership types. These constraints are verified specifically each time a container package is instantiated.

Quantification over Formal Containers

Quantified Expressions can be used over the content of a formal container to express that a property holds for all elements of a container (using `for all`) or that a property holds for at least one element of a container (using `for some`).

For example, we can express that all elements of a formal list of integers are prime as follows:

```
(for all Cu in My_List => Is_Prime (Element (My_List, Cu)))
```

On this expression, the GNAT compiler generates code that iterates over `My_List` using the functions `First`, `Has_Element` and `Next` given in the `Iterable` aspect applying to the type of formal lists, so the quantified expression above is equivalent to:

```
declare
   Cu     : Cursor_Type := First (My_List);
   Result : Boolean := True;
begin
   while Result and then Has_Element (My_List, Cu) loop
      Result := Is_Prime (Element (My_List, Cu));
      Cu     := Next (My_List, Cu);
   end loop;
end;
```

where `Result` is the value of the quantified expression. See GNAT Reference Manual for details on aspect `Iterable`.

5.11.5 Containers and Executablity

Some features of the container library (both functional and formal) might not have executable semantics. To ensure that user code never attempts to execute them, these subprograms call the `Check_Or_Fail` procedure declared in `SPARK.Containers`. This procedure is marked as `Import`, so an error will occur at link time if these features are used in normal code (or in enabled ghost code or assertions). These non-executable features include quantified expressions over functional maps, sets, and multisets and logical equality in nearly all functional and formal containers.

Quantified Expressions over Functional Maps, Sets, and Multisets

Functional maps, sets, and multisets take as parameters an equivalence relation. Inclusion in the container works modulo equivalence: when `Add` or `Remove` is called, the whole equivalence class is included or excluded at once. As equivalence classes might be infinite, quantified expressions over elements of a set or multiset or keys of a map could fail to terminate.

To replace quantified expressions over a functional map, set, or multisets occurring in the code, it is possible to use a loop over the `Iterable_Map`, `Iterable_Set`, or `Iterable_Multiset` types instead. They use the `Choose` function to get an unspecified element of the container from a different equivalence class at each iteration of the loop. As an example:

```
B := (for some E of S => P (E));
```

can be replaced by:

```
B := False;
for C in Iterate (S) loop
   pragma Loop_Invariant (for all E of S => (if not Contains (C, E) then not P (E)));
   if P (Choose (C)) then
      B := True;
      exit;
   end if;
end loop;
```

Logical Equality

The specifications of most formal and functional containers use *logical equality* to specify that all properties of elements placed in the container are preserved (see *Annotation for Accessing the Logical Equality for a Type*). Using logical equality to express such properties increases provability of user code, as it is optimally precise and natively handled by the automated solvers at the background of SPARK. However, logical equality does not always correspond to Ada equality, and there are even types for which it is not possible to write a valid logical equality in Ada, due to how things are encoded in the backend of the tool. As a result, logical equality functions used in the specification of formal and functional containers are not executable.

For formal containers, as the logical equality is given as a parameter to the functional containers used as models, the model themselves are not executable. As a result, it is not possible to execute ghost code or assertions that mention these model functions.

5.11.6 SPARK Lemma Library

As part of the SPARK library (see *SPARK Library*), packages declaring a set of ghost null procedures with contracts (called *lemmas*) are distributed. Here is an example of such a lemma:

```
procedure Lemma_Div_Is_Monotonic
  (Val1  : Int;
   Val2  : Int;
   Denom : Pos)
with
  Global => null,
  Pre  => Val1 <= Val2,
  Post => Val1 / Denom <= Val2 / Denom;
```

whose body is simply a null procedure:

```
procedure Lemma_Div_Is_Monotonic
  (Val1  : Int;
   Val2  : Int;
   Denom : Pos)
is null;
```

This procedure is ghost (as part of a ghost package), which means that the procedure body and all calls to the procedure are compiled away when producing the final executable without assertions (when switch *-gnata* is not set). On the contrary, when compiling with assertions for testing (when switch *-gnata* is set) the precondition of the procedure is executed, possibly detecting invalid uses of the lemma. However, the main purpose of such a lemma is to facilitate automatic proof, by providing the prover specific properties expressed in the postcondition. In the case of `Lemma_Div_Is_Monotonic`, the postcondition expresses an inequality between two expressions. You may use this lemma in your program by calling it on specific expressions, for example:

```
R1 := X1 / Y;
R2 := X2 / Y;
Lemma_Div_Is_Monotonic (X1, X2, Y);
--  at this program point, the prover knows that R1 <= R2
--  the following assertion is proved automatically:
pragma Assert (R1 <= R2);
```

Note that the lemma may have a precondition, stating in which contexts the lemma holds, which you will need to prove when calling it. For example, a precondition check is generated in the code above to show that `X1 <= X2`. Similarly, the types of parameters in the lemma may restrict the contexts in which the lemma holds. For example, the type `Pos` for parameter `Denom` of `Lemma_Div_Is_Monotonic` is the type of positive integers. Hence, a range check may be generated in the code above to show that `Y` is positive.

All the lemmas provided in the SPARK lemma library have been proved either automatically or using Coq interactive prover. The Why3 session file recording all proofs, as well as the individual Coq proof scripts, are available as part of the SPARK product under directory `<spark-install>/lib/gnat/proof`. For example, the proof of lemma `Lemma_Div_Is_Monotonic` is a Coq proof of the mathematical property (in Coq syntax):

```
1 subgoals
h1 : in_range val1
h2 : in_range val2
h3 : in_range1 denom
h4 : (val1 <= val2)%Z
                                        (1/1)
(val1 ÷ denom <= val2 ÷ denom)%Z
```

Currenly, the SPARK lemma library provides the following lemmas:

- Lemmas on signed integer arithmetic in file `spark-lemmas-arithmetic.ads`, that are instantiated for 32 bits signed integers (`Integer`) in file `spark-lemmas-integer_arithmetic.ads` and for 64 bits signed integers (`Long_Integer`) in file `spark-lemmas-long_integer_arithmetic.ads`.
- Lemmas on modular integer arithmetic in file `spark-lemmas-mod_arithmetic.ads`, that are instantiated for 32 bits modular integers (`Interfaces.Unsigned_32`) in file `spark-lemmas-mod32_arithmetic.ads` and for 64 bits modular integers (`Interfaces.Unsigned_64`) in file `spark-lemmas-mod64_arithmetic.ads`.
- GNAT-specific lemmas on fixed-point arithmetic in file `spark-lemmas-fixed_point_arithmetic.ads`, that need to be instantiated by the user for her specific fixed-point type.
- Lemmas on floating point arithmetic in file `spark-lemmas-floating_point_arithmetic.ads`, that are instantiated for single-precision floats (`Float`) in file `spark-lemmas-float_arithmetic.ads` and for double-precision floats (`Long_Float`) in file `spark-lemmas-long_float_arithmetic.ads`.
- Lemmas on unconstrained arrays in file `spark-lemmas-unconstrained_array.ads`, that need to be instantiated by the user for her specific type of index and element, and specific ordering function between elements.

To apply lemmas to signed or modular integers of different types than the ones used in the instances provided in the library, just convert the expressions passed in arguments, as follows:

```
R1 := X1 / Y;
R2 := X2 / Y;
Lemma_Div_Is_Monotonic (Integer(X1), Integer(X2), Integer(Y));
--  at this program point, the prover knows that R1 <= R2
--  the following assertion is proved automatically:
pragma Assert (R1 <= R2);
```

5.11.7 Higher Order Function Library

The SPARK product also includes a library of higher order functions for unconstrained arrays. It is available using the SPARK library (see *SPARK Library*).

This library consists in a set of generic entities defining usual operations on arrays. As an example, here is a generic function for the map higher-level function on arrays. It applies a given function `F` to each element of an array, returning an array of results in the same order.

```
generic
   type Index_Type is range <>;
   type Element_In is private;
   type Array_In is array (Index_Type range <>) of Element_In;

   type Element_Out is private;
   type Array_Out is array (Index_Type range <>) of Element_Out;

   with function Init_Prop (A : Element_In) return Boolean;
   --  Potential additional constraint on values of the array to allow Map

   with function F (X : Element_In) return Element_Out;
   --  Function that should be applied to elements of Array_In

function Map (A : Array_In) return Array_Out with
  Pre  => (for all I in A'Range => Init_Prop (A (I))),
  Post => Map'Result'First = A'First
```

(continues on next page)

(continued from previous page)

```
    and then Map'Result'Last = A'Last
    and then (for all I in A'Range =>
                Map'Result (I) = F (A (I)));
```

This function can be instantiated by providing two unconstrained array types ranging over the same index type and a function `F` mapping a component of the first array type to a component of the second array type. Additionally, a constraint `Init_Prop` can be supplied for the components of the first array to be allowed to apply `F`. If no such constraint is needed, `Init_Prop` can be instantiated with an always `True` function.

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

function Small_Enough (X : Natural) return Boolean is
  (X < Integer'Last);

function Increment_One (X : Integer) return Integer is (X + 1) with
  Pre => X < Integer'Last;

function Increment_All is new SPARK.Higher_Order.Map
  (Index_Type  => Positive,
   Element_In  => Natural,
   Array_In    => Nat_Array,
   Element_Out => Natural,
   Array_Out   => Nat_Array,
   Init_Prop   => Small_Enough,
   F           => Increment_One);
```

The `Increment_All` function above will take as an argument an array of natural numbers small enough to be incremented and will return an array containing the result of incrementing each number by one:

```
function Increment_All (A : Nat_Array) return Nat_Array with
  Pre  => (for all I in A'Range => Small_Enough (A (I))),
  Post => Increment_All'Result'First = A'First
  and then Increment_All'Result'Last = A'Last
  and then (for all I in A'Range =>
              Increment_All'Result (I) = Increment_One (A (I)));
```

Currently, the higher-order function library provides the following functions:

- Map functions over unconstrained one-dimensional arrays in file `spark-higher_order.ads`. These include both in place and functional map subprograms, with and without an additional position parameter.
- Fold functions over unconstrained one-dimensional and two-dimensional arrays in file `spark-higher_order-fold.ads`. Both left to right and right to left fold functions are available for one-dimensional arrays. For two-dimensional arrays, fold functions go on a line by line, left to right, top-to-bottom way. For ease of use, these functions have been instantiated for the most common cases. `Sum` and `Sum_2` respectively compute the sum of all the elements of a one-dimensional or two-dimensional array, and `Count` and `Count_2` the number of elements with a given `Choose` property.

Note: Unlike the *SPARK Lemma Library*, these generic functions are not verified once and for all as their correction depends on the functions provided at each instance. As a result, each instance should be verified by running the SPARK tools.

5.11.8 Input-Output Libraries

The following text is about `Ada.Text_IO` and its child packages, `Ada.Text_IO.Integer_IO`, `Ada.Text_IO.Modular_IO`, `Ada.Text_IO.Float_IO`, `Ada.Text_IO.Fixed_IO`, `Ada.Text_IO.Decimal_IO` and `Ada.Text_IO.Enumeration_IO`.

The effect of functions and procedures of input-output units is partially modelled. This means in particular:

- that SPARK functions cannot directly call procedures that do input-output. The solution is either to transform them into procedures, or to hide the effect from GNATprove (if not relevant for analysis) by wrapping the standard input-output procedures in procedures with an explicit `Global => null` and body with `SPARK_Mode => Off`.

```
with Ada.Text_IO;

function Foo return Integer is

   procedure Put_Line (Item : String) with
     Global => null;

   procedure Put_Line (Item : String) with
     SPARK_Mode => Off
   is
   begin
      Ada.Text_IO.Put_Line (Item);
   end Put_Line;

begin
   Put_Line ("Hello, world!");
   return 0;
end Foo;
```

- SPARK procedures that call input-output subprograms need to reflect these effects in their Global/Depends contract if they have one.

```
with Ada.Text_IO;

procedure Foo with
  Global => (Input  => Var,
             In_Out => Ada.Text_IO.File_System)
is
begin
   Ada.Text_IO.Put_Line (Var);
end Foo;

procedure Bar is
begin
   Ada.Text_IO.Put_Line (Var);
end Bar;
```

In the examples above, procedure `Foo` and `Bar` have the same body, but their declarations are different. Global contracts have to be complete or not present at all. In the case of `Foo`, it has an `Input` contract on `Var` and an `In_Out` contract on `File_System`, an abstract state from `Ada.Text_IO`. Without the latter contract, a high message would be raised when running GNATprove. Global contracts will be automatically generated for `Bar` by flow analysis if this is user code. Both declarations are accepted by SPARK.

State Abstraction and Global Contracts

The abstract state `File_System` is used to model the memory on the system and the file handles (`Line_Length`, `Col`, etc.). This is explained by the fact that almost every procedure in `Text_IO` that actually modifies attributes of the `File_Type` parameter has `in File_Type` as a parameter and not `in out`. This would be inconsistent with SPARK rules without the abstract state.

All functions and procedures are annoted with Global, and Pre, Post if necessary. The Global contracts are most of the time `In_Out` for `File_System`, even in `Put` or `Get` procedures that update the current column and/or line. Functions have an `Input` global contract. The only functions with `Global => null` are the functions `Get` and `Put` in the generic packages that have a similar behaviour as sprintf and sscanf.

Functions and Procedures Removed in SPARK

Some functions and procedures are removed from SPARK usage because they are not consistent with SPARK rules:

1. Aliasing

   The functions `Current_Input`, `Current_Output`, `Current_Error`, `Standard_Input`, `Standard_Output` and `Standard_Error` are turned off in `SPARK_Mode` because they create aliasing, by returning the corresponding file.

   `Set_Input`, `Set_Output` and `Set_Error` are turned off because they also create aliasing, by assigning a `File_Type` variable to `Current_Input` or the other two.

   It is still possible to use `Set_Input` and the 3 others to make the code clearer. This is doable by calling `Set_Input` in a different subprogram whose body has `SPARK_Mode => Off`. However, it is necessary to check that the file is open and the mode is correct, because there are no checks made on procedures that do not take a file as a parameter (i.e. implicit, so it will write to/read from the current output/input).

2. `Get_Line` function

   The function `Get_Line` is disabled in SPARK because it is a function with side effects. Even with the `Volatile_Function` attribute, it is not possible to model its action on the files and global variables in SPARK. The function is very convenient because it returns an unconstrained string, but a workaround is possible by constructing the string with a buffer:

```
with Ada.Text_IO;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;

procedure Echo is
   Unb_Str : Unbounded_String := Null_Unbounded_String;
   Buffer  : String (1 .. 1024);
   Last    : Natural := 1024;
begin

   while Last = 1024 loop
      Ada.Text_IO.Get_Line (Buffer, Last);
      exit when Last > Natural'Last - Length (Unb_Str);
      Unb_Str := Unb_Str & Buffer (1 .. Last);
   end loop;

   declare
      Str : String := To_String (Unb_Str);
   begin
      Ada.Text_IO.Put_Line (Str);
```

(continues on next page)

(continued from previous page)

```
   end;
end Echo;
```

Errors Handling

`Status_Error` (due to a file already open/not open) and `Mode_Error` are fully handled.

Except for `Layout_Error`, which is a special case of a partially handled error and explained in a few lines below, all other errors are not handled:

- `Use_Error` is related to the external environment.
- `Name_Error` would require to check availability on disk beforehand.
- `End_Error` is raised when a file terminator is read while running the procedure.

For an `Out_File`, it is possible to set a `Line_Length` and `Page_Length`. When writing in this file, the procedures will add Line markers and Page markers each `Line_Length` characters or `Page_Length` lines respectively. `Layout_Error` occurs when trying to set the current column or line to a value that is greater than `Line_Length` or `Page_Length` respectively. This error is handled when using `Set_Col` or `Set_Line` procedures.

However, this error is not handled when no `Line_Length` or `Page_Length` has been specified, e.g, if the lines are unbounded, it is possible to have a `Col` greater than `Count'Last` and therefore have a `Layout_Error` raised when calling `Col`.

Not only the handling is partial, but it is also impossible to prove preconditions when working with two files or more. Since `Line_Length` etc. attributes are stored in the `File_System`, it is not posible to prove that the `Line_Length` of `File_2` has not been modified when running any procedure that do input-output on `File_1`.

Finally, `Layout_Error` may be raised when calling `Put` to display the value of a real number (floating-point or fixed-point) in a string output parameter, which is not reflected currently in the precondition of `Put` as no simple precondition can describe the required length in such a case.

5.11.9 Strings Libraries

The following text is about `Ada.Strings.Maps`, `Ada.Strings.Fixed`, `Ada.Strings.Bounded` and `Ada.Strings.Unbounded`.

Global contracts were added to non-pure packages, and pre/postconditions were added to all SPARK subprograms to partially model their effects. In particular:

- Effects of subprograms from `Ada.Strings.Maps`, as specified in the Ada RM (A.4.2), are fully modeled through pre- and postconditions.
- Effects of most subprograms from `Ada.Strings.Fixed` are fully modeled through pre- and postconditions. Preconditions protect from exceptions specified in the Ada RM (A.4.3). Some procedures are not annotated with sufficient preconditions and may raise `Length_Error` when called with inconsistent parameters. They are annotated with an exceptional contract.

  Under their respective preconditions, the implementation of subprograms from `Ada.Strings.Fixed` is proven with GNATprove to be free from run-time errors and to comply with their postcondition, except for procedure `Move` and those procedures based on `Move`: `Delete`, `Head`, `Insert`, `Overwrite`, `Replace_Slice`, `Tail` and `Trim` (but the corresponding functions are proved).
- Effects of subprograms from `Ada.Strings.Bounded` are fully modeled through pre- and postconditions. Preconditions protect from exceptions specified in the Ada RM (A.4.4).

Under their respective preconditions, the implementation of subprograms from `Ada.Strings.Bounded` is proven with GNATprove to be free from run-time errors, and except for subprograms `Insert`, `Overwrite` and `Replace_Slice`, to comply with their postcondition.

- Effects of subprograms from `Ada.Strings.Unbounded` are partially modeled. Postconditions state properties on the Length of the strings only and not on their content. Preconditions protect from exceptions specified in the Ada RM (A.4.5).
- The procedure `Free` in `Ada.Strings.Unbounded` is not in SPARK as it could be wrongly called by the user on a pointer to the stack.

Inside these packages, `Translation_Error` (in `Ada.Strings.Maps`), `Index_Error` and `Pattern_Error` are fully handled.

`Length_Error` is fully handled in `Ada.Strings.Bounded` and `Ada.Strings.Unbounded` and in functions from `Ada.Strings.Fixed`.

However, in the procedure `Move` and the procedures based on it except for `Delete` and `Trim` (`Head`, `Insert`, `Overwrite`, `Replace_Slice` and `Tail`) from `Ada.Strings.Fixed`, `Length_Error` may be raised under certain conditions. This is related to the call to `Move`. Each call of these subprograms can be preceded with a pragma Assert to check that the actual parameters are consistent, when parameter `Drop` is set to `Error` and the `Source` is longer than `Target`.

```
--  From the Ada RM for Move: "The Move procedure copies characters from
--  Source to Target.
--
--  ...
--
--  If Source is longer than Target, then the effect is based on Drop.
--
--  ...
--
--  * If Drop=Error, then the effect depends on the value of the Justify
--    parameter and also on whether any characters in Source other than Pad
--    would fail to be copied:
--
--    * If Justify=Left, and if each of the rightmost
--      Source'Length-Target'Length characters in Source is Pad, then the
--      leftmost Target'Length characters of Source are copied to Target.
--
--    * If Justify=Right, and if each of the leftmost
--      Source'Length-Target'Length characters in Source is Pad, then the
--      rightmost Target'Length characters of Source are copied to Target.
--
--    * Otherwise, Length_Error is propagated.".
--
--  Here, Move will be called with Drop = Error, Justify = Left and
--  Pad = Space, so we add the following assertion before the call to Move.

pragma Assert
 (if Source'Length > Target'Length then
    (for all J in 1 .. Source'Length - Target'Length =>
       (Source (Source'Last - J + 1) = Space)));

Move (Source  => Source,
      Target  => Target,
```

(continues on next page)

(continued from previous page)

```
      Drop    => Error,
      Justify => Left,
      Pad     => Space);
```

5.11.10 C Strings Interface

`Interfaces.C.Strings` is a library that provides an Ada interface to allocate, reference, update and free C strings.

The provided preconditions protect users from getting `Dereference_Error` and `Update_Error`. However, those preconditions do not protect against `Storage_Error`.

All subprograms are annotated with Global contracts. To model the effects of the subprograms on the allocated memory, an abstract state `C_Memory` is defined. Since `chars_ptr` is an access type that is hidden from SPARK (it is a private type and the private part of `Interfaces.C.Strings` has `SPARK_Mode => Off`), the user could create aliases that SPARK would not be able to see. Hence, we consider that calling `Update` on any `chars_ptr` modifies the allocated memory, `C_Memory`, so that the effects of potential aliases are modelled correctly.

Additionally, some subprograms are annotated with `SPARK_Mode => Off`:

- `To_Chars_Ptr`: This function creates an alias, thus it is not compatible with SPARK.
- `Free`: There is no way for SPARK to know whether or not it is safe to deallocate these pointers. They might not be allocated on the heap or there might be some aliases, which could lead to dangling pointers.

Finally, the two functions used to allocate memory to create `chars_ptr` objects are annotated with the `Volatile_Function` attribute. Indeed, calling those functions twice in a row with the same parameters would return different objects.

5.11.11 Addresses to Access Conversions

The run-time library `System.Address_To_Access_Conversions` enables the user to convert `System.Address_Type` values to general access-to-object types. The conversions are subject to the same rules as `Unchecked_Conversion` between such types (see *Data Validity*), that is:

- `To_Pointer` is allowed in SPARK and annotated with `Global => null`. On a call to this function, GNATprove will emit warnings to ensure that the designated data has no aliases and is initialized.
- `To_Address` is forbidden in SPARK because it does not handle addresses.

5.11.12 Cut Operations

The SPARK product also includes boolean cut operations that can be used to manually help the proof of complicated assertions. These operations are provided as functions in a library but are handled in a specific way by GNATprove. They can be found in the `SPARK.Cut_Operations` package which is available using the same project file as the *SPARK Lemma Library*.

This library provides two functions named `By` and `So` which are handled in the following way:

- If `A` and `B` are two boolean expressions, proving `By (A, B)` requires proving `B`, the premise, and then `A` assuming `B`, the side-condition. When `By (A, B)` is assumed on the other hand, GNATprove only assumes `A`. The proposition `B` is used for the proof of `A`, but is not visible afterward.
- If `A` and `B` are two boolean expressions, proving `So (A, B)` requires proving `A`, the premise, and then `B` assuming `A`, the side-condition. When `So (A, B)` is assumed both `A` and `B` are assumed to be true.

This allows to introduce intermediate assertions to help the proof of some part of an assertion by still controlling in a precise way what is added to the enclosing context. This is interesting when doing complex proofs when the size of the proof context (the amount of information known at a given program point) is an issue.

In the example below, `By` is used to add the intermediate property `B (I)` in the proof of `C (I)` from `A (I)`:

```
with SPARK.Cut_Operations; use SPARK.Cut_Operations;

procedure Main with SPARK_Mode is
   function A (I : Integer) return Boolean with Import;
   function B (I : Integer) return Boolean with Import,
     Post => B'Result and then (if A (I) then C (I));
   function C (I : Integer) return Boolean with Import;
begin
   pragma Assume (for all I in 1 .. 100 => A (I));
   pragma Assert (for all I in 1 .. 100 => By (C (I), B (I)));
end;
```

To prove the assertion, GNATprove will attempt to verify both that `B (I)` is true for all `I` and that `C (I)` can be deduced from `B (I)`. After the assertion, the call to `B` does not occur in the context anymore, it is as if the assertion `(for all I in 1 .. 100 => C (I))` had been written directly. Remark that here, `B` is really a lemma. Its result does not matter in itself (it always returns True) but its postcondition gives additional information for the proof (see the *SPARK Lemma Library* for more information about lemmas).

As can be seen in the example above, `By` and `So` may not necessarily occur at top-level in an assertion. However, because of their specific treatment, they are only allowed in specific contexts. We define these supported contexts in a recursive way:

- As the expression of a `pragma Assert` or `Assert_And_Cut`;
- As an operand of a AND, OR, AND THEN, or OR ELSE operation which itself occurs in a supported context;
- As the THEN or ELSE branch of a IF expression which itself occurs in a supported context;
- As an alternative of a CASE expression which itself occurs in a supported context;
- As the condition of a quantified expression which itself occurs in a supported context;
- As a parameter to a call to either `By` or `So` which itself occurs in a supported context;
- As the body expression of a DECLARE expression which itself occurs in a supported context.

CHAPTER

SIX

SPARK TUTORIAL

This chapter describes a simple use of the SPARK toolset on a program written completely in SPARK, within the GNAT Studio integrated development environment. All the tools may also be run from the command-line, see *Command Line Invocation*.

Note: If you're using SPARK Discovery instead of SPARK Pro, some of the proofs in this tutorial may not be obtained automatically. See the section on *Alternative Provers* to install additional provers that are not present in SPARK Discovery.

6.1 Writing SPARK Programs

As a running example, we consider a naive searching algorithm for an unordered collection of elements. The algorithm returns whether the collection contains the desired value, and if so, at which index. The collection is implemented here as an array. We deliberately start with an incorrect program for the `Search` function, in order to explain how the SPARK toolset can help correct these errors. The final version of the `linear_search` example is part of the examples in the distribution via the *Help* → *SPARK* → *Examples* menu item.

We start with creating a GNAT project file:

```
project linear_search_ada is
   for Source_Dirs use (".");

   package Compiler is
      for Switches ("Ada") use ("-gnatwa");
   end Compiler;
end linear_search_ada;
```

It specifies that the source code to inspect is in the current directory, and that the code should be compiled at maximum warning level (switch `-gnatwa`). GNAT projects are used by most tools in the GNAT toolsuite; for in-depth documentation of this technology, consult the GNAT User's Guide. Documentation and examples for the SPARK language and tools are also available via the *Help* → *SPARK* menu in GNAT Studio.

The obvious specification of `Linear_Search` is given in file `linear_search.ads`, where we specify that the spec is in SPARK by using aspect `SPARK_Mode`.

```
package Linear_Search
  with SPARK_Mode
is

```

(continues on next page)

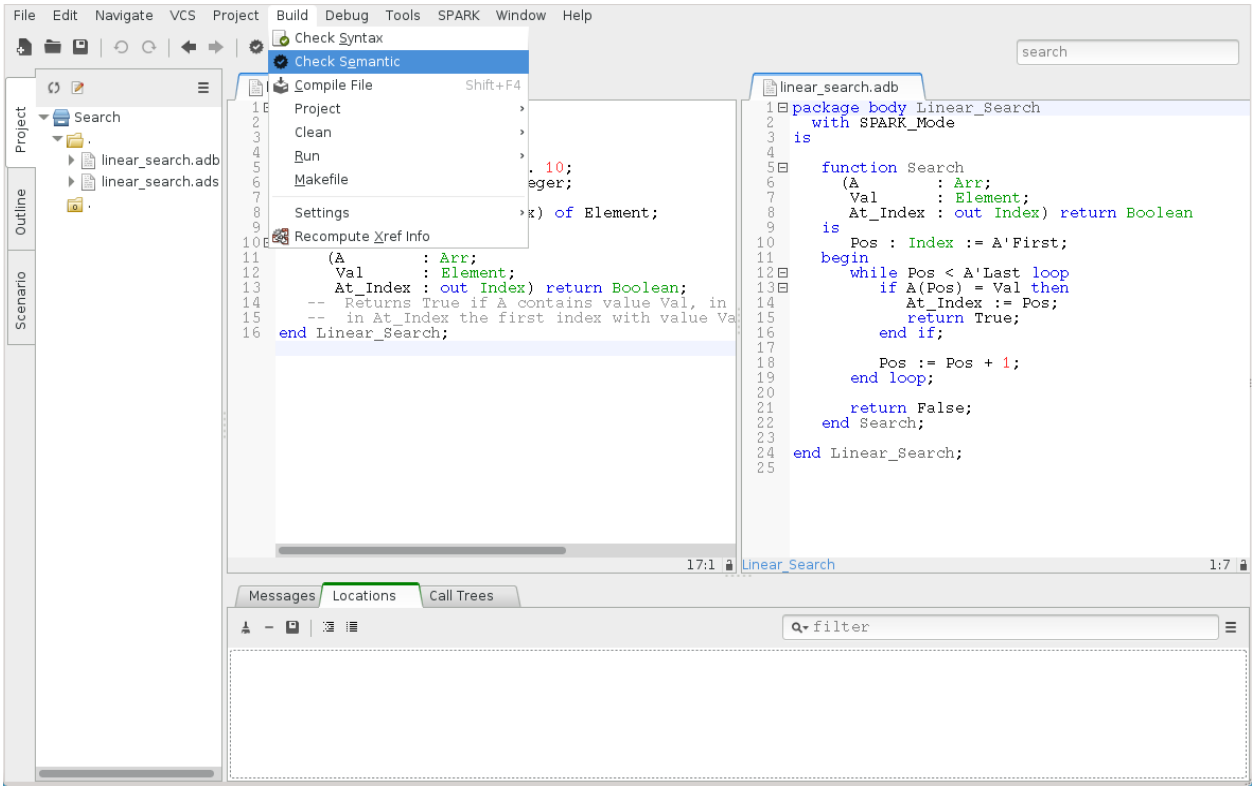
(continued from previous page)

```
   type Index is range 1 .. 10;
   type Element is new Integer;

   type Arr is array (Index) of Element;

   function Search
     (A        : Arr;
      Val      : Element;
      At_Index : out Index) return Boolean;
   --  Returns True if A contains value Val, in which case it also returns
   --  in At_Index the first index with value Val. Returns False otherwise.
end Linear_Search;
```

The implementation of `Linear_Search` is given in file `linear_search.adb`, where we specify that the body is in SPARK by using aspect `SPARK_Mode`. It is as obvious as its specification, using a loop to go through the array parameter `A` and looking for the first index at which `Val` is found, if there is such an index.

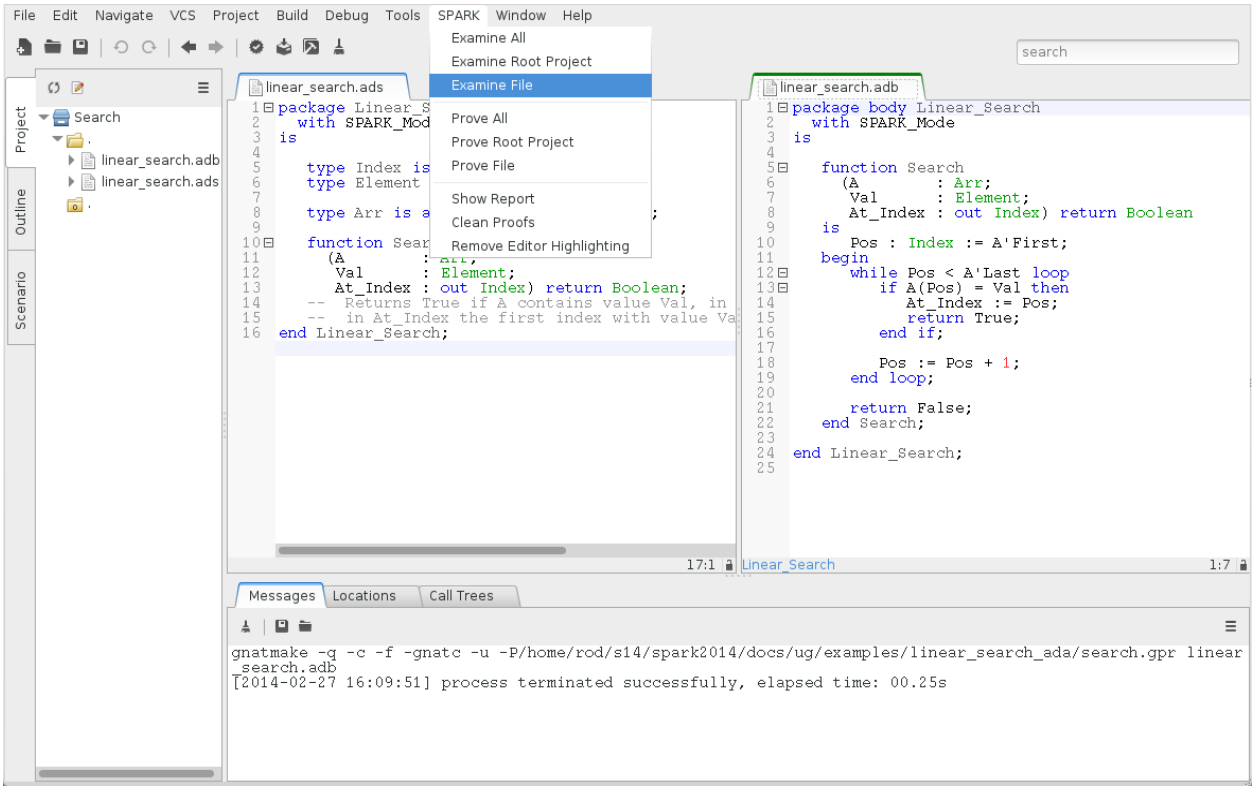
```
package body Linear_Search
  with SPARK_Mode
is

   function Search
     (A        : Arr;
      Val      : Element;
      At_Index : out Index) return Boolean
   is
      Pos : Index := A'First;
   begin
      while Pos < A'Last loop
         if A(Pos) = Val then
            At_Index := Pos;
            return True;
         end if;

         Pos := Pos + 1;
      end loop;

      return False;
   end Search;

end Linear_Search;
```

We can check that the above code is valid Ada by using the `Build > Check Semantic` menu, which completes without any errors or warnings:

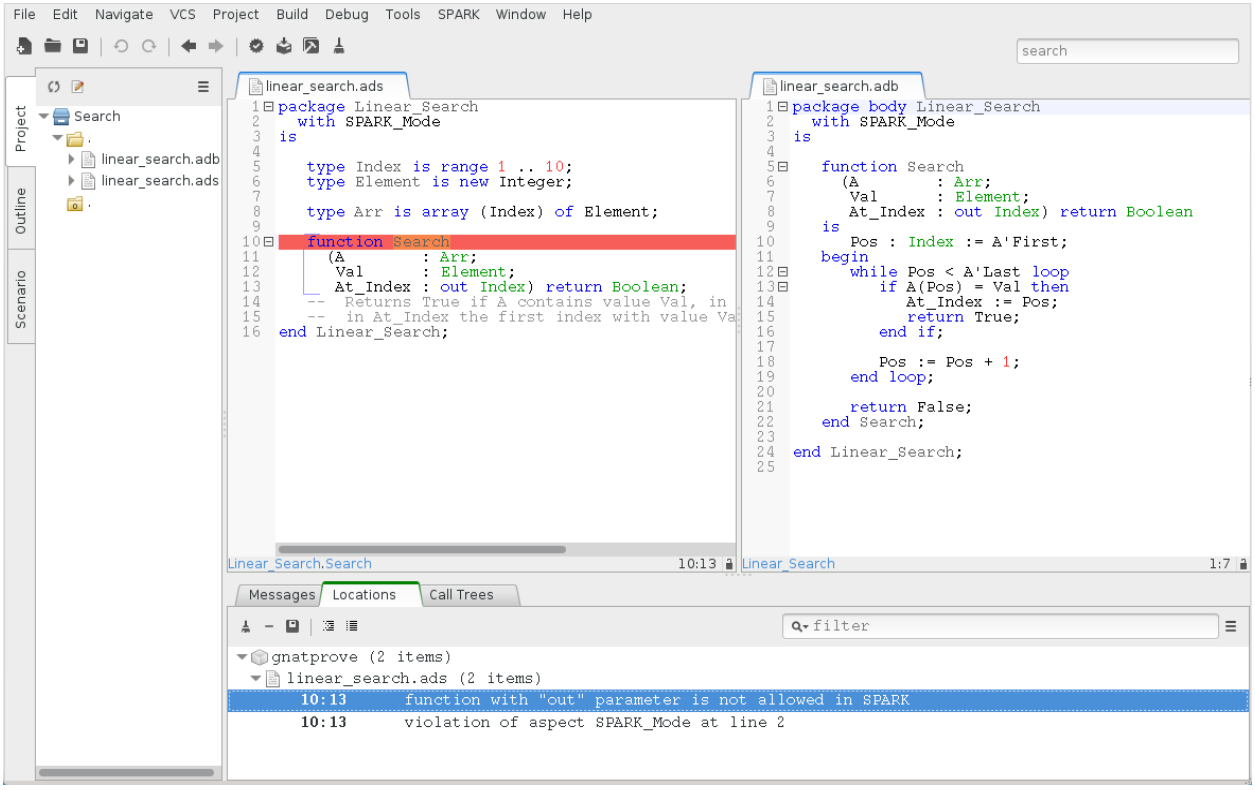


6.1.1 Checking SPARK Legality Rules

Now, let us run GNATprove on this unit, using the *SPARK → Examine File* menu, so that it issues errors on SPARK code that violates SPARK rules:



It detects here that function `Search` is not in SPARK, because it has an `out` parameter:



The permission in Ada 2012 to have `out` parameters to functions is not allowed in SPARK, because it causes calls to have side-effects (assigning to their `out` parameters), which means that various calls in the same expression may be

conflicting, yielding different results depending on the order of evaluation of the expression.

We correct this problem by defining a record type `Search_Result` in `linear_search.ads` holding both the Boolean result and the index for cases when the value is found, and making `Search` return this type:

```
package Linear_Search
  with SPARK_Mode
is

   type Index is range 1 .. 10;
   type Element is new Integer;

   type Arr is array (Index) of Element;

   type Search_Result is record
      Found    : Boolean;
      At_Index : Index;
   end record;

   function Search
     (A   : Arr;
      Val : Element) return Search_Result;

end Linear_Search;
```

The implementation of `Search` in `linear_search.adb` is modified to use this type:

```
package body Linear_Search
  with SPARK_Mode
is

   function Search
     (A   : Arr;
      Val : Element) return Search_Result
   is
      Pos : Index := A'First;
      Res : Search_Result;
   begin
      while Pos < A'Last loop
         pragma Loop_Variant (Increases => Pos);

         if A(Pos) = Val then
            Res.At_Index := Pos;
            Res.Found := True;
            return Res;
         end if;

         Pos := Pos + 1;
      end loop;

      Res.Found := False;
      return Res;
   end Search;

```

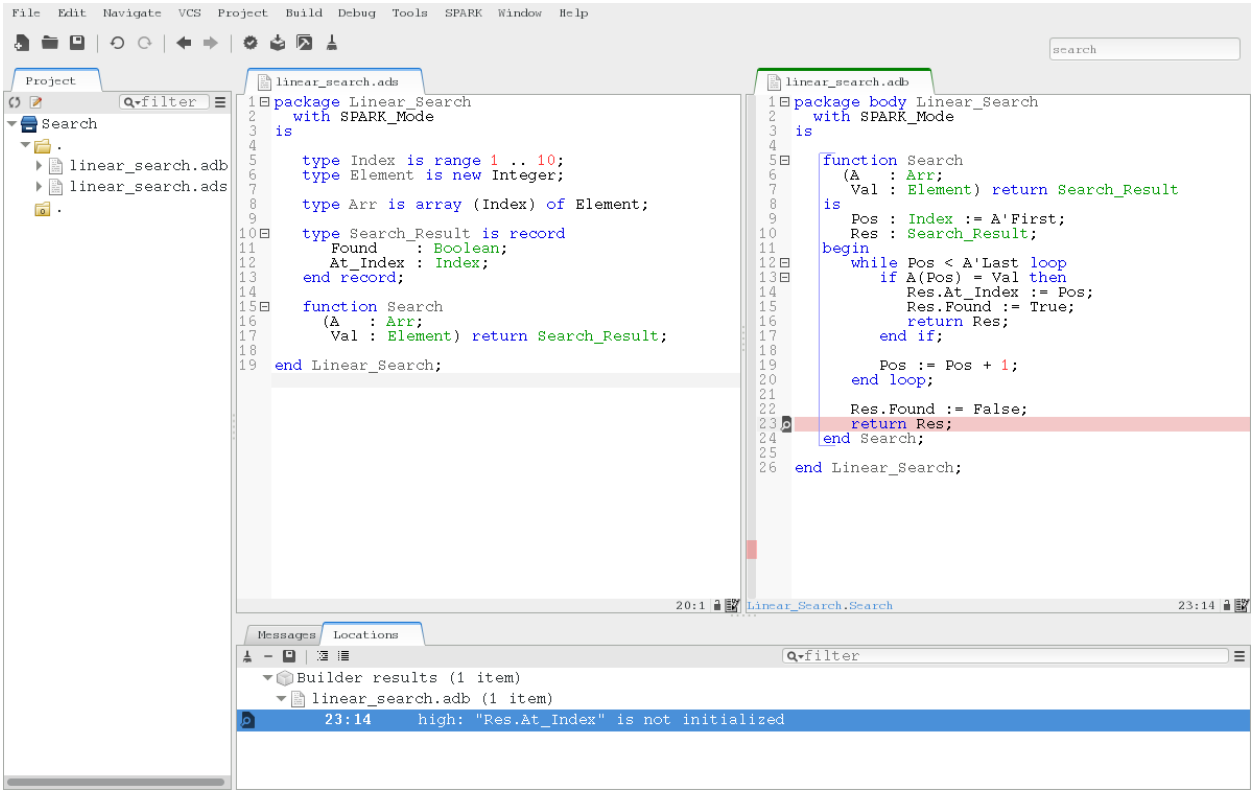
(continues on next page)

(continued from previous page)

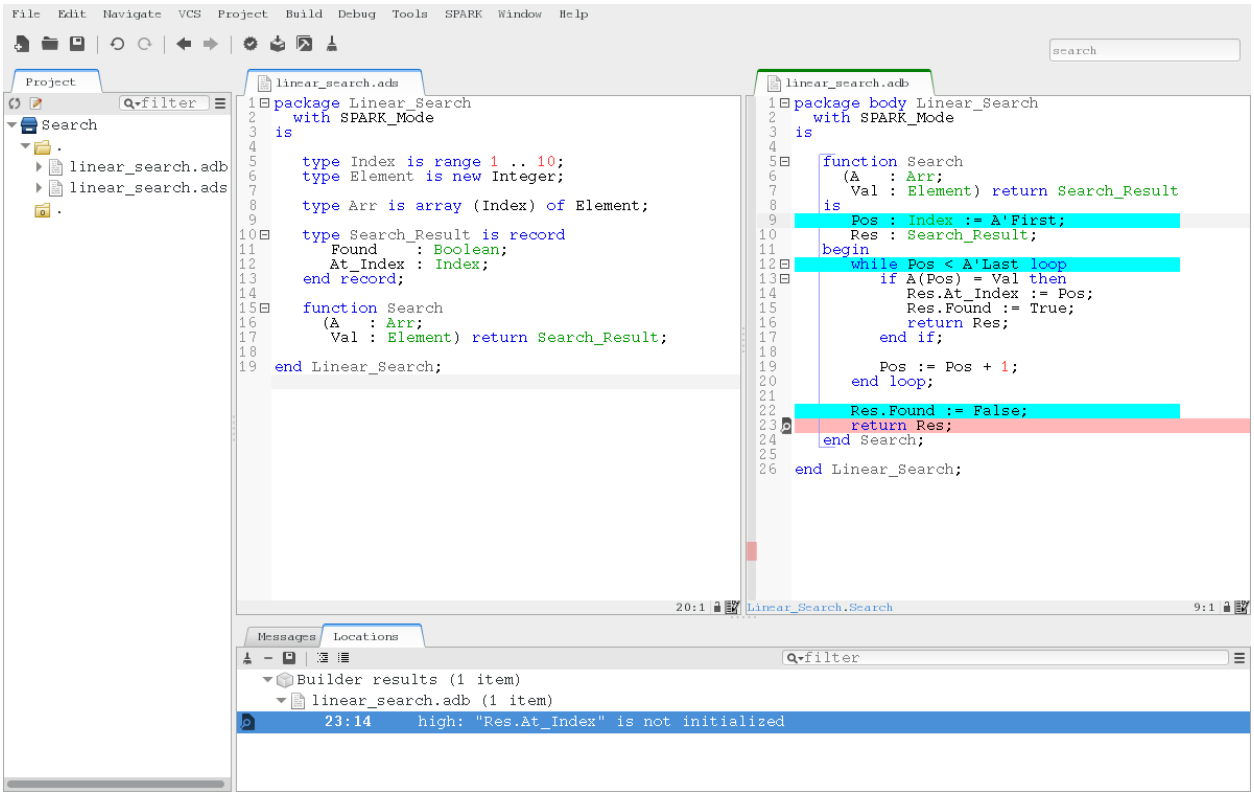
```
28 end Linear_Search;
```

6.1.2 Checking SPARK Initialization Policy

Re-running GNATprove on this unit, still using the *SPARK → Examine File* menu, now reports a different kind of error. This time it is the static analysis pass of GNATprove called *flow analysis* that detects an attempt of the program to return variable `Res` while it is not fully initialized, thus violating the initialization policy of SPARK:



Inside the GNAT Studio editor, we can click on the icon, either on the left of the message, or on line 23 in file `linear_search.adb`, to show the path on which `Res.At_Index` is not initialized:



Another click on the icon makes the path disappear.

This shows that, when the value is not found, the component `At_Index` of the value returned is indeed not initialized. Although that is allowed in Ada, SPARK requires that all inputs and outputs of subprograms are completely initialized (and the value returned by a function is such an output). As a solution, we could give a dummy value to component `At_Index` when the search fails, but we choose here to turn the type `Search_Result` in `linear_search.ads` into a discriminant record, so that the component `At_Index` is only usable when the search succeeds:

```
type Search_Result (Found : Boolean := False) is record
   case Found is
      when True =>
         At_Index : Index;
      when False =>
         null;
   end case;
end record;
```

Then, in the implementation of `Search` in `linear_search.adb`, we change the value of the discriminant depending on the success of the search:

```
function Search
  (A   : Arr;
   Val : Element) return Search_Result
is
   Pos : Index := A'First;
   Res : Search_Result;
begin
   while Pos < A'Last loop
      pragma Loop_Variant (Increases => Pos);
```

(continues on next page)

(continued from previous page)

```
10
11          if A(Pos) = Val then
12             Res := (Found    => True,
13                     At_Index => Pos);
14             return Res;
15          end if;
16
17          Pos := Pos + 1;
18       end loop;
19
20       Res := (Found => False);
```

Now re-running GNATprove on this unit, using the *SPARK → Examine File* menu, shows that there are no reads of uninitialized data.

6.1.3 Writing Functional Contracts

We now have a valid SPARK program. It is not yet very interesting SPARK code though, as it does not contain any contracts, which are necessary to be able to apply formal verification modularly on each subprogram, independently of the implementation of other subprograms. The precondition constrains the value of input parameters, while the postcondition states desired properties of the result of the function. See *Preconditions* and *Postconditions* for more details. Here, we can require in the precondition of `Search` in `linear_search.ads` that callers of `Search` always pass a non-negative value for parameter `Val`, and we can state that, when the search succeeds, the index returned points to the desired value in the array:

```
1    function Search
2      (A   : Arr;
3       Val : Element) return Search_Result
4    with
5      Pre  => Val >= 0,
6      Post => (if Search'Result.Found then
7                 A (Search'Result.At_Index) = Val),
```

Notice the use of an if-expression in the postcondition to express an implication: if the search succeeds it implies that the value at the returned index is the value that was being searched for. Note also the use of `Search'Result` to denote the value returned by the function.

This contract is still not very strong. Many faulty implementations of the search would pass this contract, for example one that always fails (thus returning with `Search'Result.Found = False`). We could reinforce the postcondition, but we choose here to do it through a contract by cases, which adds further constraints to the usual contract by precondition and postcondition. We want to consider here three cases:

- the desired value is found at the first index (1)
- the desired value is found at other indexes (2 to 10)
- the desired value is not found in the range 1 to 10

In the first case, we want to state that the index returned is 1. In the second case, we want to state that the search succeeds. In the third case, we want to state that the search fails. We use a helper function `Value_Found_In_Range` in `linear_search.ads` to express that a value `Val` is found in an array `A` within given bounds `Low` and `Up`:

```
1    function Value_Found_In_Range
2      (A       : Arr;
```

(continues on next page)

(continued from previous page)

```
      Val     : Element;
      Low, Up : Index) return Boolean
   is (for some J in Low .. Up => A(J) = Val);

   function Search
     (A   : Arr;
      Val : Element) return Search_Result
   with
     Pre  => Val >= 0,
     Post => (if Search'Result.Found then
                A (Search'Result.At_Index) = Val),
     Contract_Cases =>
       (A(1) = Val =>
          Search'Result.At_Index = 1,
        Value_Found_In_Range (A, Val, 2, 10) =>
          Search'Result.Found,
        (for all J in Arr'Range => A(J) /= Val) =>
          not Search'Result.Found);
```

Note that we express `Value_Found_In_Range` as an expression function, a function whose body consists of a single expression, which can be given in a specification file.

Note also the use of quantified expressions to express properties over collections: `for some` in `Value_Found_In_Range` expresses an existential property (there exists an index in this range such that ...), `for all` in the third contract case expresses a universal property (all indexes in this range are such that ...).

Each contract case consists of a guard (on the left of the arrow symbol) evaluated on subprogram entry, and a consequence (on the right of the arrow symbol) evaluated on subprogram exit. The special expression `Search'Result` may be used in consequence expressions. The three guards here should cover all possible cases, and be disjoint. When a contract case is activated (meaning its guard holds on entry), its consequence should hold on exit.

The program obtained so far is a valid SPARK program, which GNAT analyzes semantically without errors or warnings.

6.2 Testing SPARK Programs

We can compile the above program, and test it on a set of selected inputs. The following test program in file `test_search.adb` exercises the case where the searched value is present in the array and the case where it is not:

```
with Linear_Search; use Linear_Search;
with Ada.Text_IO;   use Ada.Text_IO;

procedure Test_Search is
   A   : constant Arr := (1, 5, 3, 8, 8, 2, 0, 1, 0, 4);
   Res : Search_Result;

begin
   Res := Search (A, 1);
   if Res.Found then
      if Res.At_Index = 1 then
         Put_Line ("OK: Found existing value at first index");
      else
         Put_Line ("not OK: Found existing value at other index");
```

(continues on next page)

(continued from previous page)

```
      end if;
   else
      Put_Line ("not OK: Did not find existing value");
   end if;

   Res := Search (A, 6);
   if not Res.Found then
      Put_Line ("OK: Did not find non-existing value");
   else
      Put_Line ("not OK: Found non-existing value");
   end if;
end Test_Search;
```

We can check that the implementation of `Linear_Search` passes this test by compiling and running the test program:

```
$ gprbuild test_search.adb
$ test_search
> OK: Found existing value at first index
> OK: Did not find non-existing value
```

Note: We use above the command-line interface to compile and run the test program `test_search.adb`. You can do the same inside GNAT Studio by selecting the menu *Project* → *Properties* and inside the panel *Main* of folder *Sources*, add `test_search.adb` as a main file. Then, click *OK*. To generate the `test_search` executable, you can now select the menu *Build* → *Project* → *test_search.adb* and to run the `test_search` executable, you can select the menu *Build* → *Run* → *test_search*.

But only part of the program was really tested, as the contract was not checked during execution. To check the contract at run time, we recompile with the switch `-gnata` (`a` for assertions, plus switch `-f` to force recompilation of sources that have not changed):

- a check is inserted that the precondition holds on subprogram entry
- a check is inserted that the postcondition holds on subprogram exit
- a check is inserted that the guards of contract cases are disjoint on subprogram entry (no two cases are activated at the same time)
- a check is inserted that the guards of contract cases are complete on subprogram entry (one case must be activated)
- a check is inserted that the consequence of the activated contract case holds on subprogram exit

Note that the evaluation of the above assertions may also trigger other run-time check failures, like an index out of bounds. With these additional run-time checks, an error is reported when running the test program:

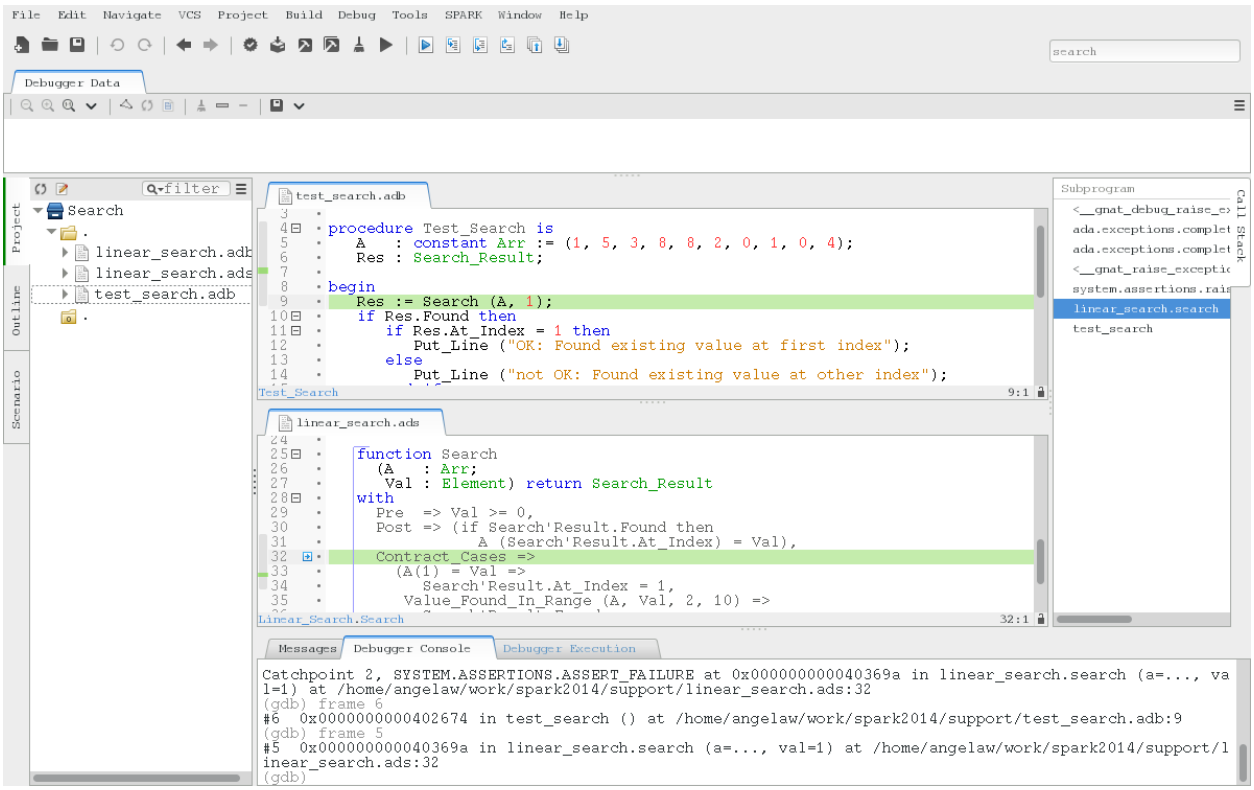
```
$ gprbuild -gnata -f test_search.adb
$ test_search
> raised ADA.ASSERTIONS.ASSERTION_ERROR : contract cases overlap for subprogram search
```

Note: We use above the command-line interface to add compilation switch `-gnata` and force recompilation with switch `-f`. You can do the same inside GNAT Studio by selecting the menu *Project* → *Properties* and inside the panel *Ada* of the subfolder *Switches* of folder *Build*, select the checkbox *Enable assertions*. Then, click *OK*. To force recompilation with the new switch, you can now select the menu *Build* → *Clean* → *Clean All* followed by recompilation with *Build* → *Project* → *test_search.adb*. Then run the `test_search` executable with *Build* → *Run* → *test_search*.

It appears that two contract cases for `Search` are activated at the same time! More information can be generated at run time if the code is compiled with the switch `-gnateE`:

```
$ gprbuild -gnata -gnateE -f test_search.adb
$ test_search
> raised ADA.ASSERTIONS.ASSERTION_ERROR : contract cases overlap for subprogram search
>   case guard at linear_search.ads:33 evaluates to True
>   case guard at linear_search.ads:35 evaluates to True
```

It shows here that the guards of the first and second contract cases hold at the same time. This failure in annotations can be debugged with `gdb` like a failure in the code (provided the program was compiled with appropriate switches, like `-g -O0`). The stack trace inside GNAT Studio shows that the error occurs on the first call to `Search` in the test program:



Indeed, the value 1 is present twice in the array, at indexes 1 and 8, which makes the two guards `A(1) = Val` and `Value_Found_In_Range (A, Val, 2, 10)` evaluate to `True`. We correct the contract of `Search` in `linear_search.ads` by strengthening the guard of the second contract case, so that it only applies when the value is not found at index 1:

```
1     Contract_Cases =>
2       (A(1) = Val =>
3          Search'Result.At_Index = 1,
4        A(1) /= Val and then Value_Found_In_Range (A, Val, 2, 10) =>
5          Search'Result.Found,
6        (for all J in Arr'Range => A(J) /= Val) =>
7          not Search'Result.Found);
```

With this updated contract, the test passes again, but this time with assertions checked at run time:

```
$ gnatmake -gnata test_search.adb
$ test_search
> OK: Found existing value at first index
> OK: Did not find non-existing value
```

The program obtained so far passes successfully a test campaign (of one test!) that achieves 100% coverage for all the common coverage criteria, once impossible paths have been ruled out: statement coverage, condition coverage, the MC/DC coverage used in avionics, and even the full static path coverage.

6.3 Proving SPARK Programs

Formal verification of SPARK programs is a two-step process:

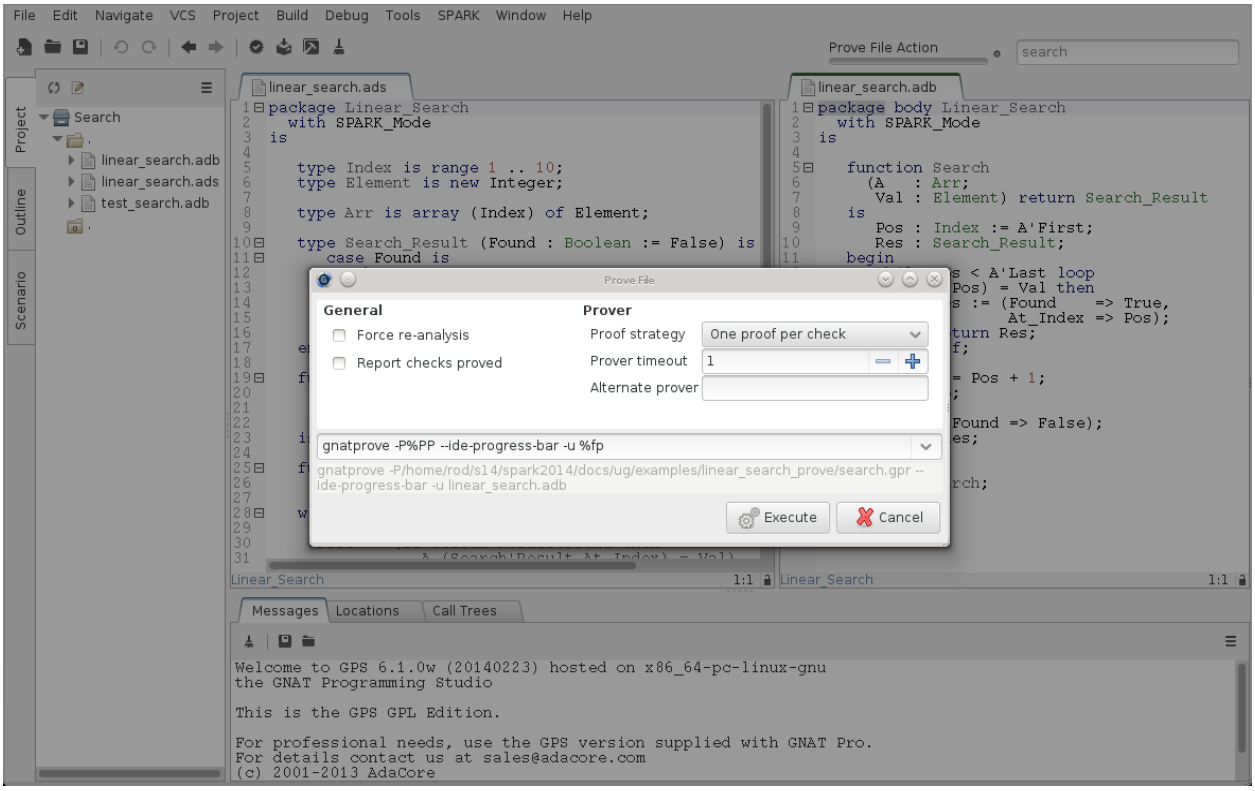
1. the first step checks that flows through the program correctly implement the specified flows (if any), and that all values read are initialized.
2. the second step checks that the program correctly implement its specified contracts (if any), and that no run-time error can be raised.

Step 1 is implemented as a static analysis pass in the tool GNATprove, in `flow` mode. We have seen this flow analysis at work earlier (see *Checking SPARK Initialization Policy*). Step 2 is implemented as a deductive verification (a.k.a. *proof*) pass in the tool GNATprove, in the default `all` mode.

The difference between these two steps should be emphasized. Flow analysis in step 1 is a terminating algorithm, which typically takes 2 to 10 times as long as compilation to complete. Proof in step 2 is based on the generation of logical formulas for each check to prove, which are then passed on to automatic provers to decide whether the logical formula holds or not. The generation of logical formulas is a translation phase, which typically takes 10 times as long as compilation to complete. The automatic proof of logical formulas may take a very long time, or never terminate, hence the use of a timeout (1s at proof level 0) for each call to the automatic provers. It is this last step which takes the most time when calling GNATprove on a program, but it is also a step which can be completely parallelized (using switch `-j` to specify the number of parallel processes): each logical formula can be proved independently, so the more cores are available the faster it completes.

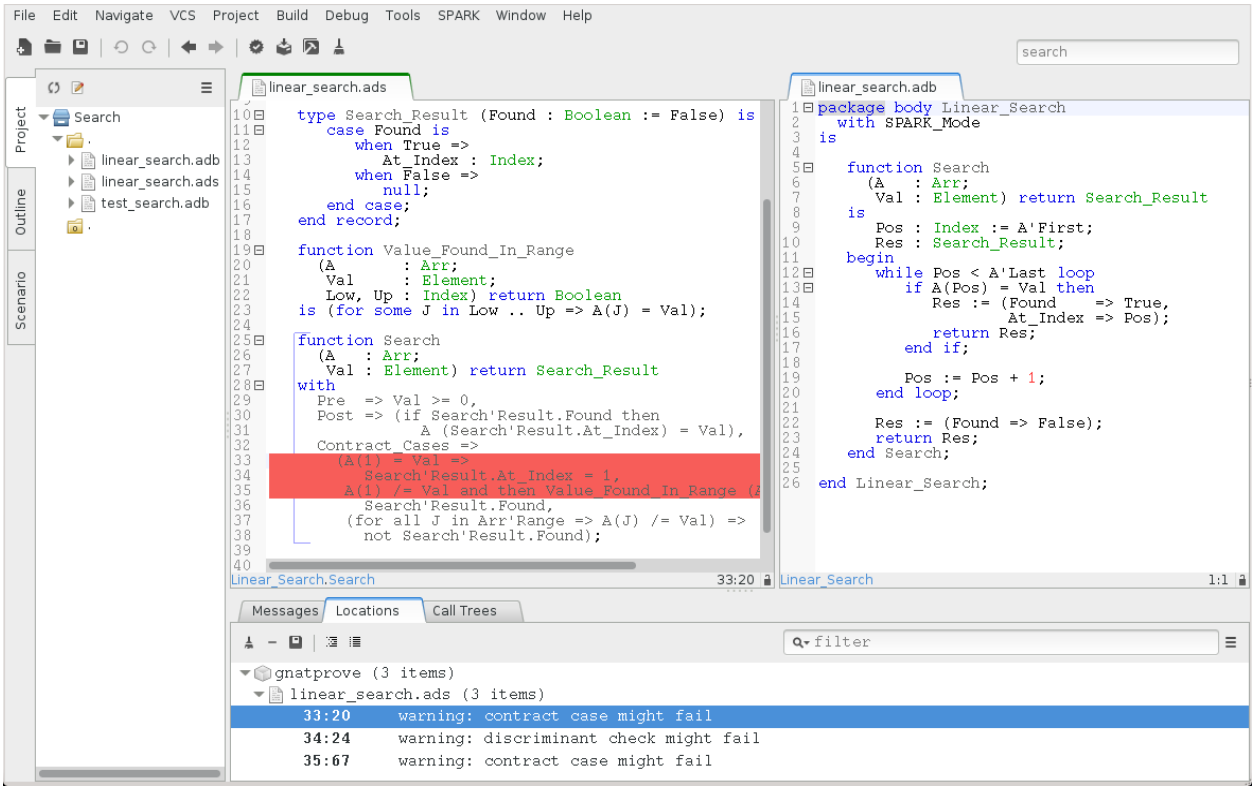
Note: The proof results presented in this tutorial may slightly vary from the results you obtain on your machine, as automatic provers may take more or less time to complete a proof depending on the platform and machine used.

Let us continue with our running example. This time we will see how step 2 works to prove contracts and absence of run-time errors, using the main mode `all` of GNATprove reached through the *SPARK → Prove File* menu.



Note: The proof panels presented in this tutorial correspond to an advanced user profile. A simpler proof panel is displayed when the basic user profile is selected (the default). You can switch to the advanced user profile in menu *Edit* → *Preferences* → *SPARK*, by changing the value of *User profile* from `Basic` to `Advanced`. See *Running GNATprove from GNAT Studio* for details.

We use the default settings and click on *Execute*. It completes in a few seconds, with a message stating that some checks could not be proved:

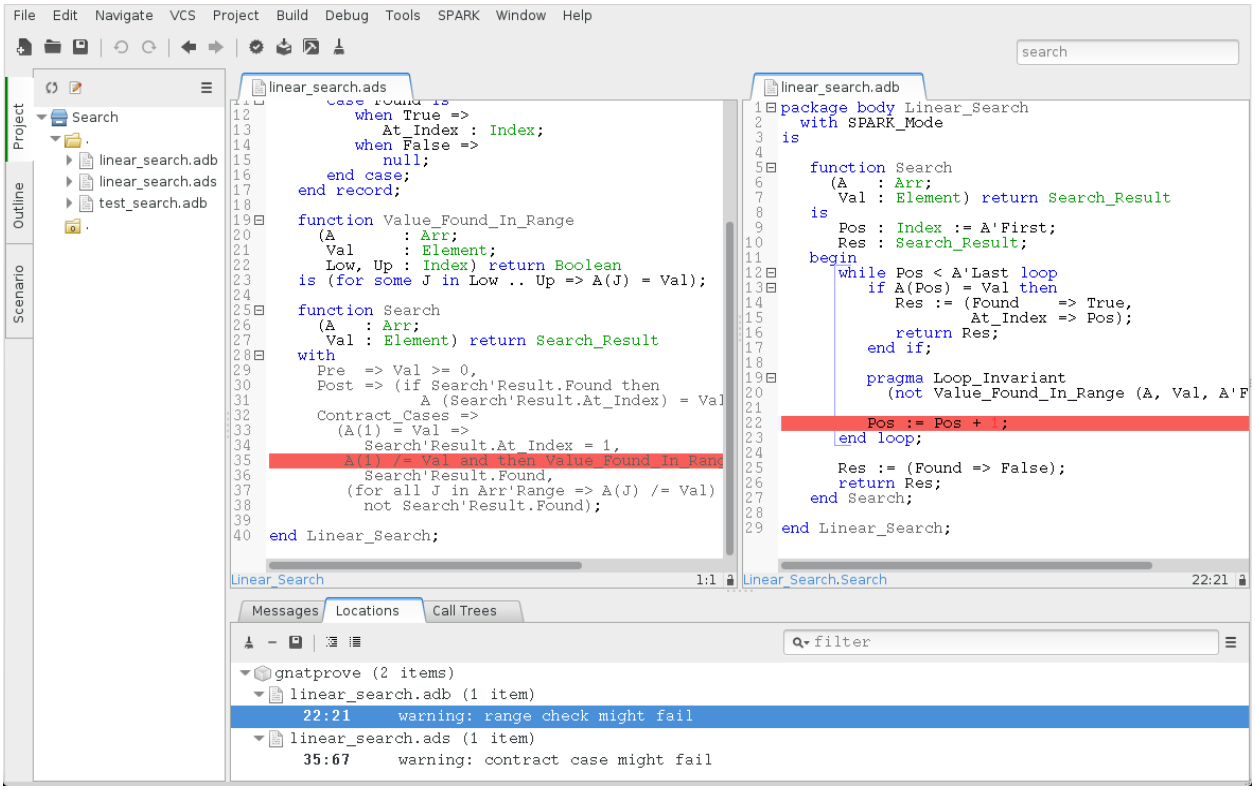


Note that there is no such message on the postcondition of `Search`, which means that it was proved. Likewise, there are no such messages on the body of `Search`, which means that no run-time errors can be raised when executing the function.

These messages correspond to checks done when exiting from `Search`. It is expected that not much can be proved at this point, given that the body of `Search` has a loop but no loop invariant, so the formulas generated for these checks assume the worst about locations modified in the loop. A loop invariant is a special pragma `Loop_Invariant` stating an assertion in a loop, which can be both executed at run-time like a regular pragma `Assert`, and used by GNATprove to summarize the effect of successive iterations of the loop. We need to add a loop invariant in `linear_search.adb` stating enough properties about the cumulative effect of loop iterations, so that the contract cases of `Search` become provable. Here, it should state that the value searched was not previously found:

```
pragma Loop_Invariant
  (not Value_Found_In_Range (A, Val, A'First, Pos));
```

As stated above, this invariant holds exactly between the two statements in the loop in `linear_search.adb` (after the if-statement, before the increment of the index). Thus, it should be inserted at this place. With this loop invariant, two checks previously not proved are now proved, and a check previously proved becomes unproved:



The new unproved checks may seem odd, since all we did was add information in the form of a loop invariant. The reason is that we also removed information at the same time. By adding a loop invariant, we require GNATprove to prove iterations around the (virtual) loop formed by the following steps:

1. Take any context satisfying the loop invariant, which summarizes all previous iterations of the loop.
2. Execute the end of a source loop iteration (just the increment here).
3. Test whether the loop exits, and continue with values which do not exit.
4. Execute the start of a source loop iteration (just the if-statement here).
5. Check that the loop invariant still holds.

Around this virtual loop, nothing guarantees that the index `Pos` is below the maximal index at step 2 (the increment), so the range check cannot be proved. It was previously proved because, in the absence of a loop invariant, GNATprove proves iterations around the source loop, and then we get the information that, since the loop did not exit, its test `Pos < A'Last` is false, so the range check can be proved.

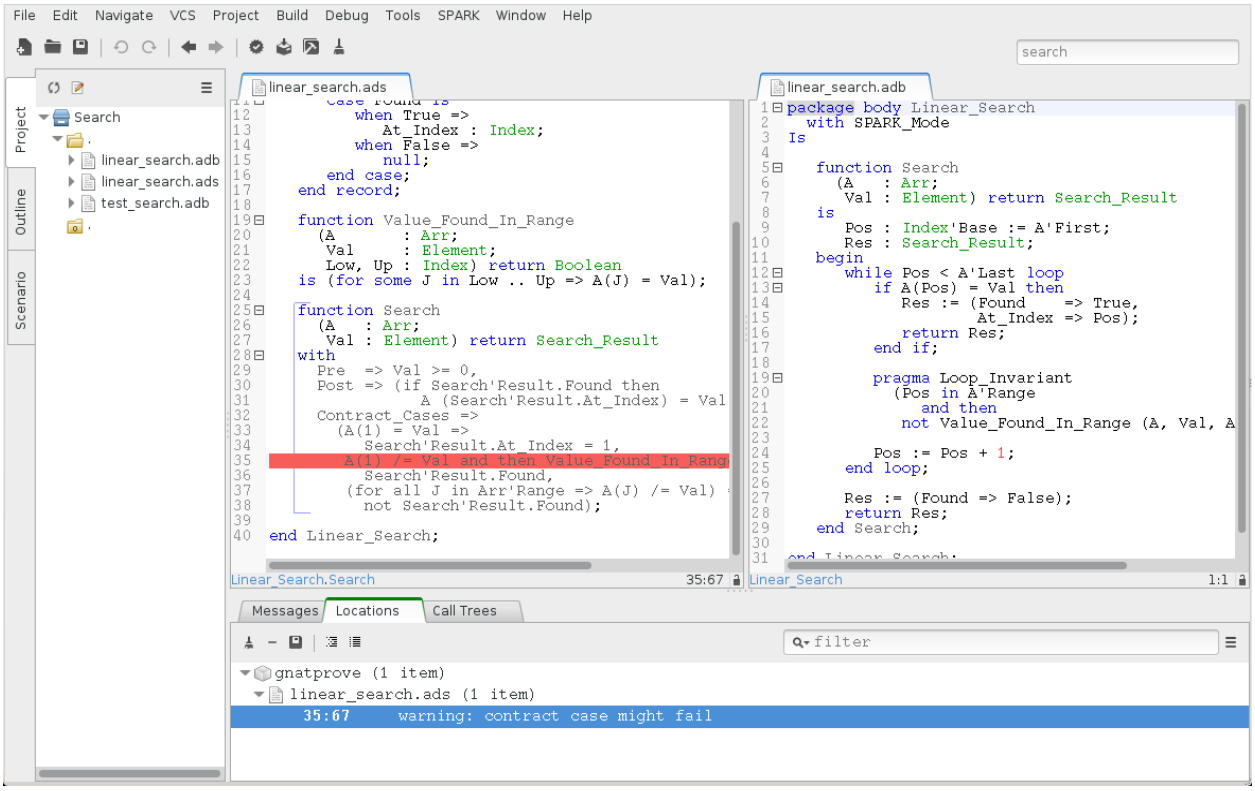
We solve this issue by setting the type of `Pos` in `linear_search.adb` to the base type of `Index`, which ranges past the last value of `Index`. (This may not be the simplest solution, but we use it here for the dynamics of this tutorial.)

```
      Pos : Index'Base := A'First;
```

And we add the range information for `Pos` to the loop invariant in `linear_search.adb`:

```
         pragma Loop_Invariant
           (Pos in A'Range
              and then
            not Value_Found_In_Range (A, Val, A'First, Pos));
```

This allows GNATprove to prove the range check, but not the contract:



This is actually progress! Indeed, the loop invariant should be strong enough to:

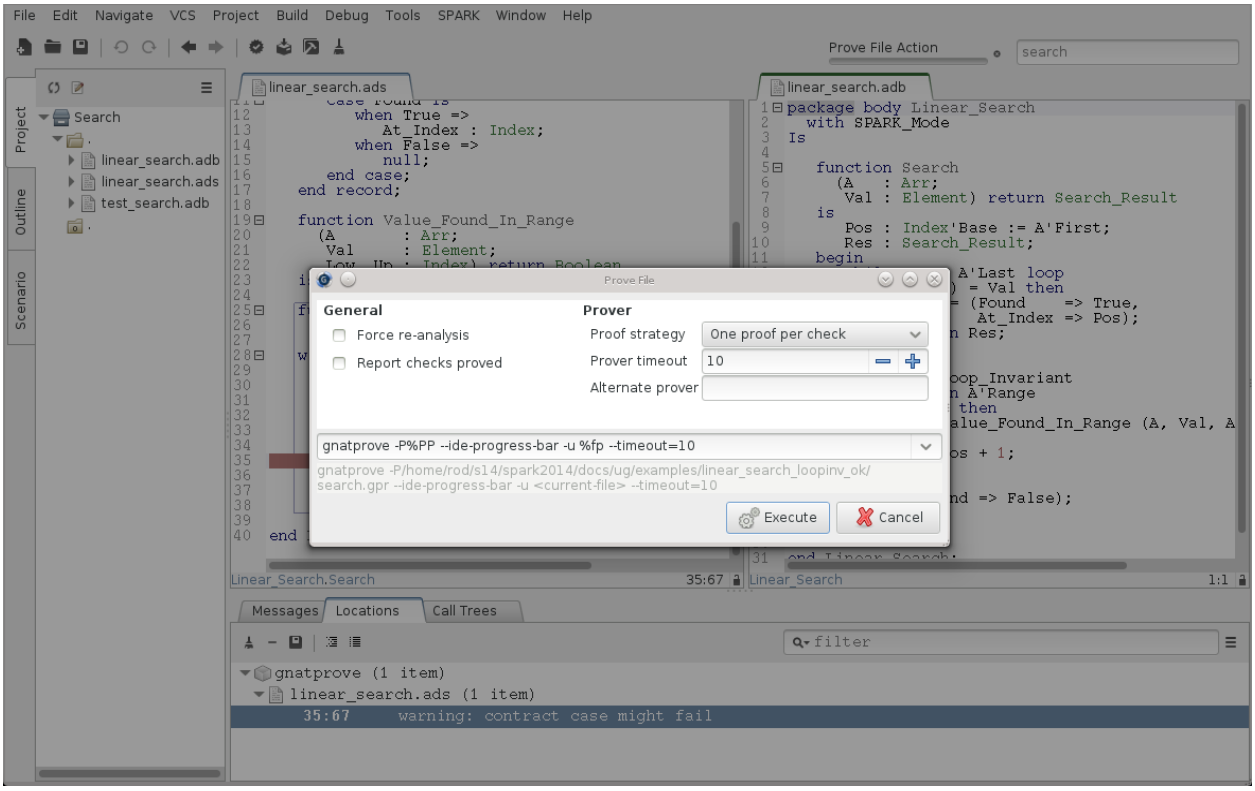
1. prove the absence of run-time errors in the loop and after the loop
2. prove that it is preserved from iteration to iteration
3. prove the postcondition and contract cases of the subprogram

So we have just achieved goal 1 above!

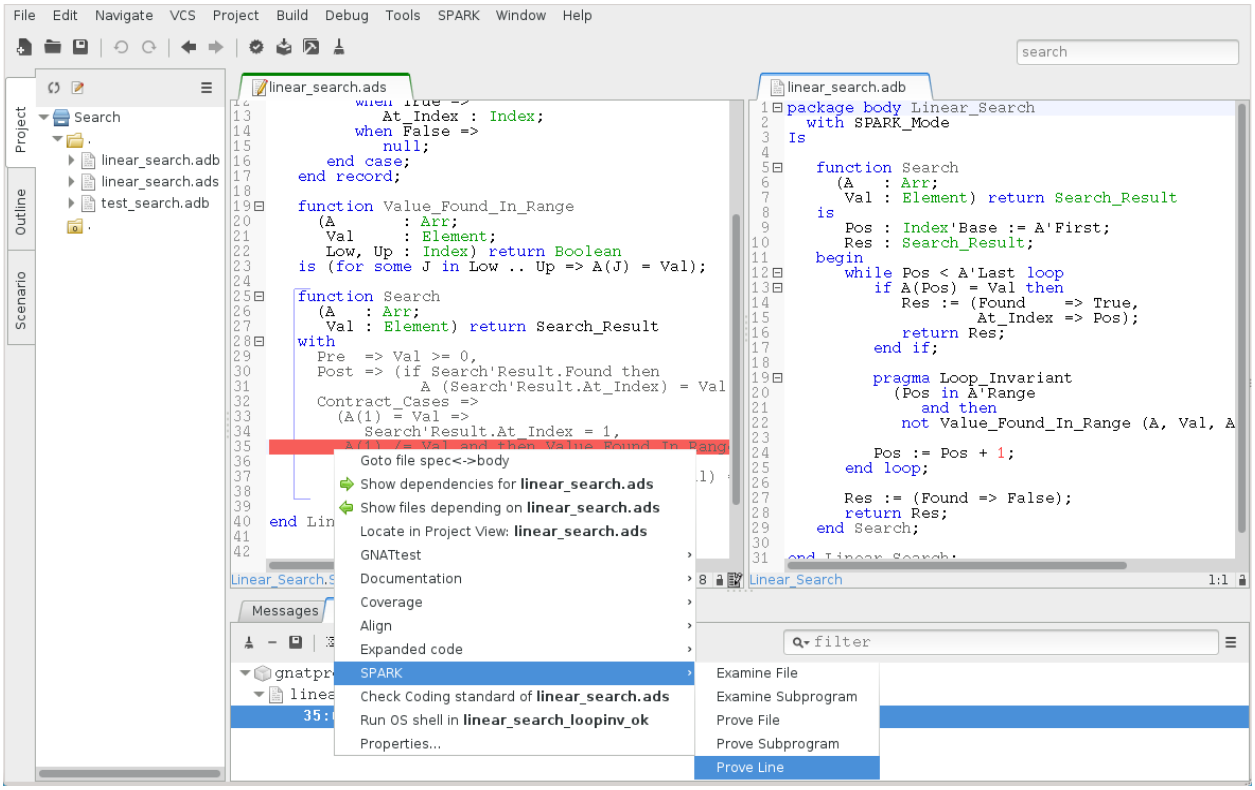
As we have modified the code and annotations, it is a good time to compile and run our test program, before doing any more formal verification work. This helps catch bugs early, and it's easy to do! In particular, the loop invariant will be dynamically checked at each iteration through the loop. Here, testing does not show any problems:

```
$ gnatmake -gnata test_search.adb
$ test_search
> OK: Found existing value at first index
> OK: Did not find non-existing value
```

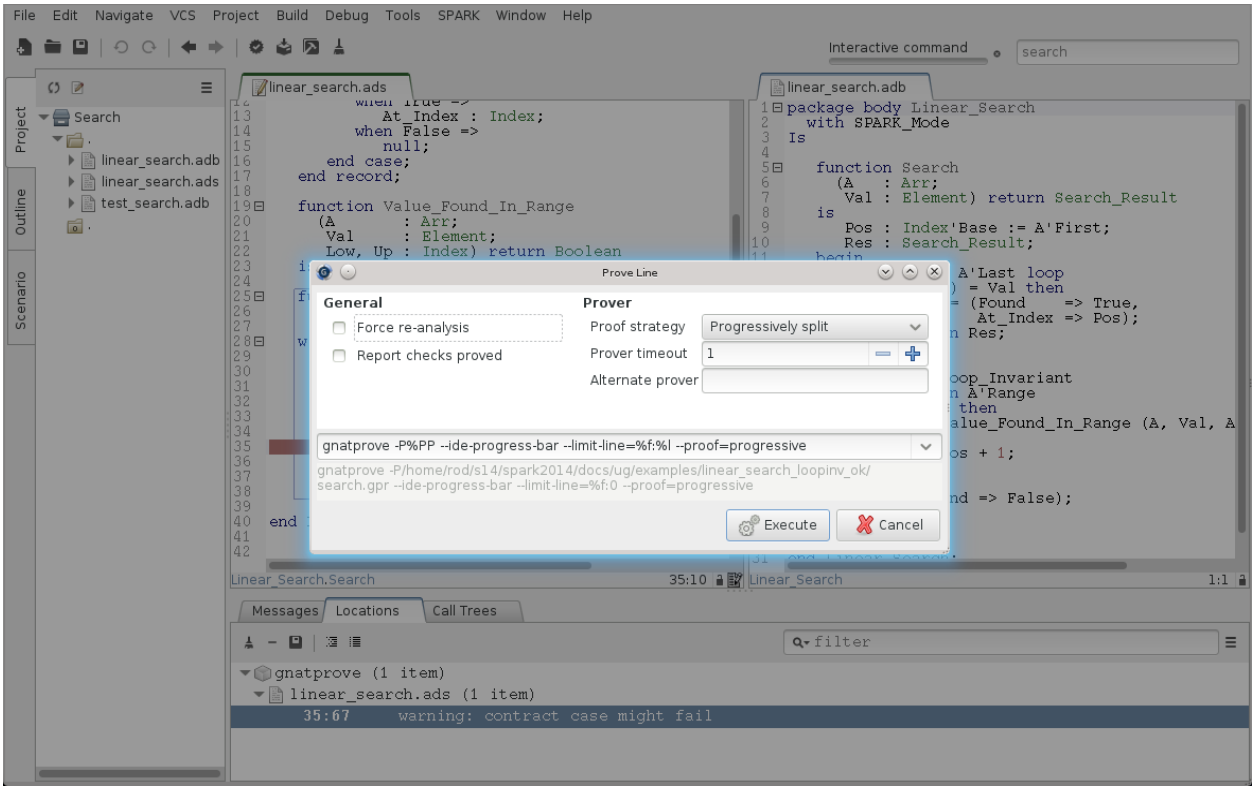
The next easy thing to do is to increase the proof level. Its default of 0 is deliberately low, to facilitate interaction with GNATprove during the development of annotations, but it is not sufficient to prove the more complex checks. Let's increase it to level 2 (passing switch `--level=2` or equivalently setting the `Proof level` to 2 in the proof panel), and rerun GNATprove:



The unproved check remains in the contract cases of `Linear_Search`. The next step is to use the *SPARK → Prove Line* contextual menu available on line 35:



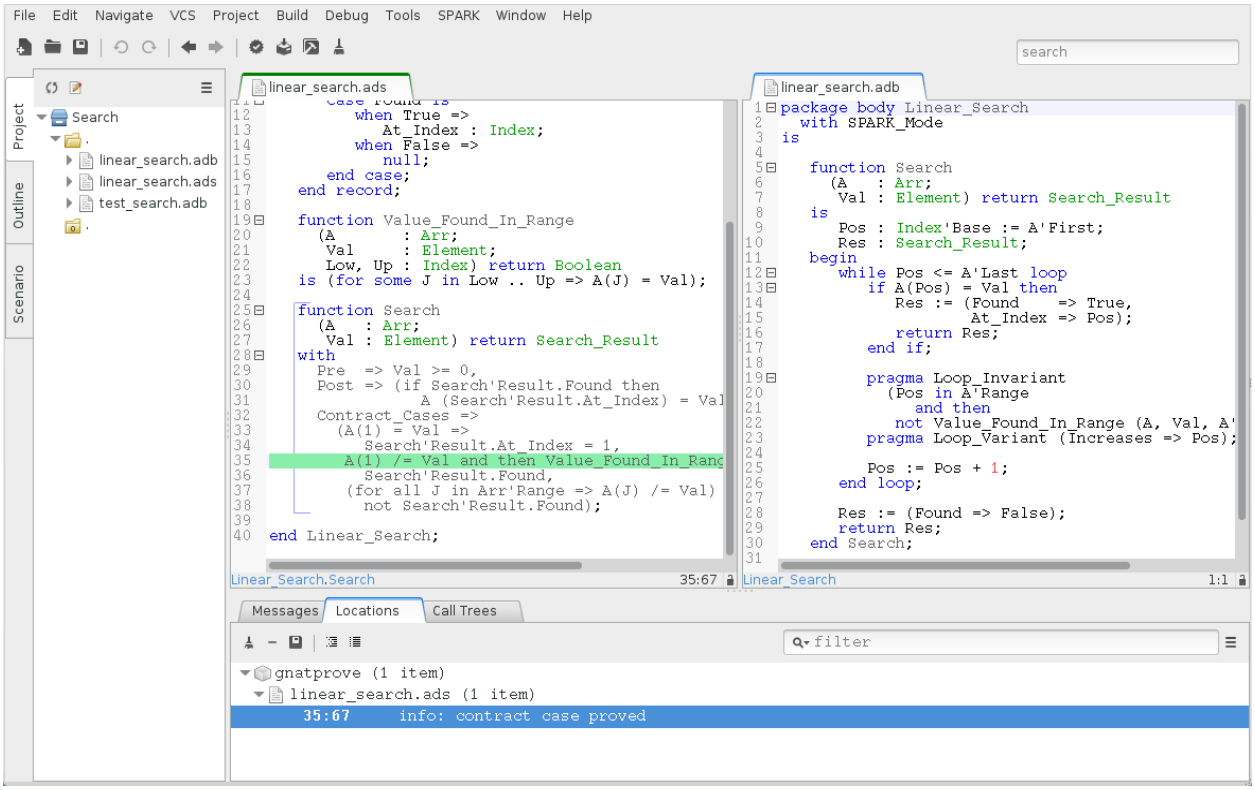
We further increase the `Proof level` to 3 to maximize proof precision, and click on *Execute*:



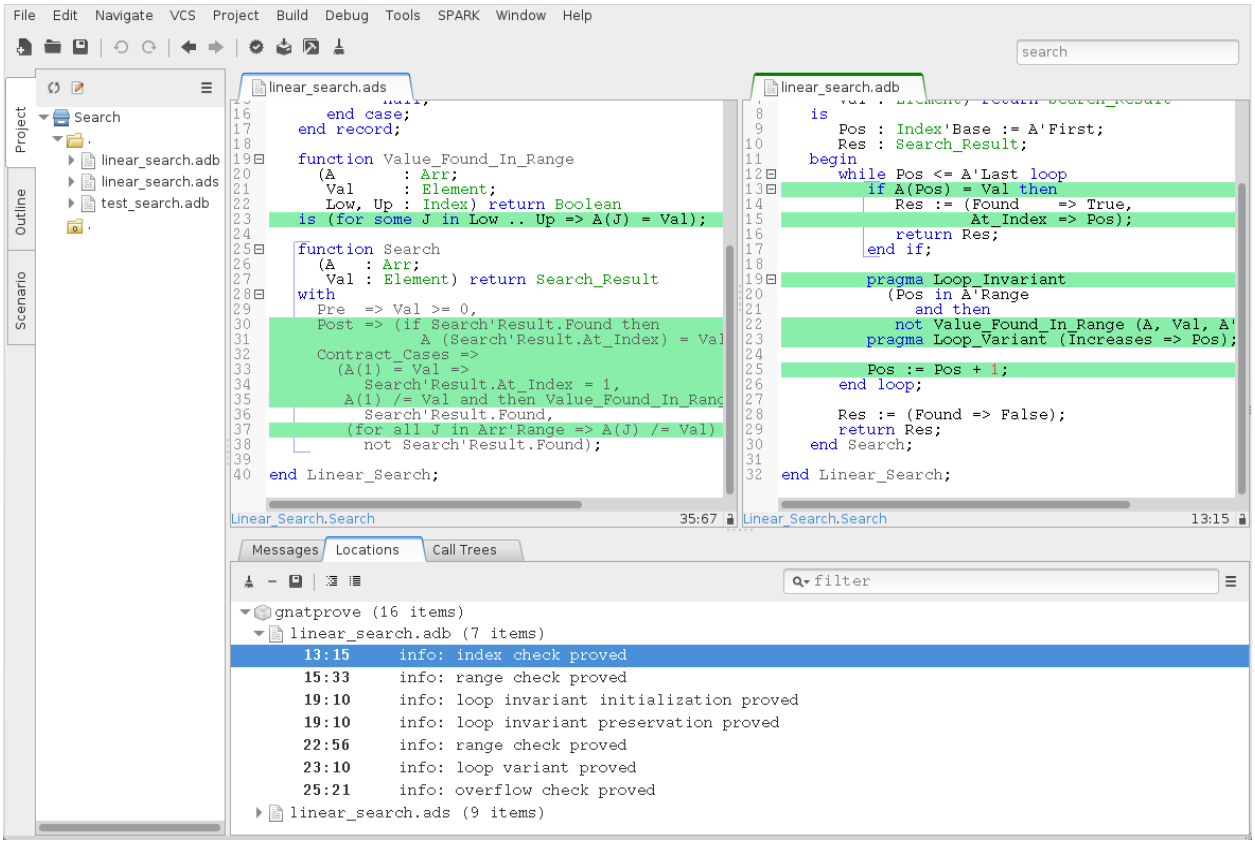
This runs GNATprove only on the checks that originate from line 35. The check is still not proved. This corresponds to a case where the implementation of `Search` does not find the searched value, but the guard of the second contract case holds, meaning that the value is present in the range 2 to 10. Looking more closely at the code, we can see that the loop exits when `Pos = A'Last`, so the value 10 is never considered! We correct this bug by changing the loop test in `linear_search.adb` from a strict to a non-strict comparison operation:

```
      while Pos <= A'Last loop
```

On this modified code, we rerun GNATprove on line 35, selecting value `all checks with provers used` for the value of `Report` in the proof panel to get information even when a check is proved. The reassuring green or blue color (and the accompanying info message) show that the check was proved this time:



As usual after code changes, we rerun the test program, which shows no errors. Rerunning GNATprove on the complete file shows no more unproved checks. The `Linear_Search` unit has been fully proved. To see all the checks that were proved, we can rerun the tool with value `all checks with provers used` for the value of `Report` in the proof panel, which displays the results previously computed:



Note that one thing that was not proved is that `Search` terminates. As it contains a while-loop, it could loop forever. To prove that it is not the case, we add a loop variant, which specifies a quantity varying monotonically with each iteration. Since this quantity is bounded by its type, and we have proved absence of run-time errors in `Search`, proving this monotonicity property also shows that there cannot be an infinite number of iterations of the loop. The natural loop variant for `Search` is the index `Pos`, which increases at each loop iteration:

```
         pragma Loop_Variant (Increases => Pos);
```

With this line inserted after the loop invariant in `linear_search.adb`, the test program still runs without errors (it checks dynamically that the loop variant is respected), and the program is still fully proved. Here is the final version of `Linear_Search`, with the complete annotations:

```
1  package Linear_Search
2    with SPARK_Mode
3  is
4
5     type Index is range 1 .. 10;
6     type Element is new Integer;
7
8     type Arr is array (Index) of Element;
9
10    type Search_Result (Found : Boolean := False) is record
11       case Found is
12          when True =>
13             At_Index : Index;
14          when False =>
15             null;
```

(continues on next page)

(continued from previous page)

```
      end case;
   end record;

   function Value_Found_In_Range
     (A       : Arr;
      Val     : Element;
      Low, Up : Index) return Boolean
   is (for some J in Low .. Up => A(J) = Val);

   function Search
     (A   : Arr;
      Val : Element) return Search_Result
   with
     Pre  => Val >= 0,
     Post => (if Search'Result.Found then
                A (Search'Result.At_Index) = Val),
     Contract_Cases =>
       (A(1) = Val =>
          Search'Result.At_Index = 1,
        A(1) /= Val and then Value_Found_In_Range (A, Val, 2, 10) =>
          Search'Result.Found,
        (for all J in Arr'Range => A(J) /= Val) =>
          not Search'Result.Found);

end Linear_Search;
```

```
package body Linear_Search
  with SPARK_Mode
is

   function Search
     (A   : Arr;
      Val : Element) return Search_Result
   is
      Pos : Index'Base := A'First;
      Res : Search_Result;
   begin
      while Pos <= A'Last loop
         if A(Pos) = Val then
            Res := (Found    => True,
                    At_Index => Pos);
            return Res;
         end if;

         pragma Loop_Invariant
           (Pos in A'Range
              and then
            not Value_Found_In_Range (A, Val, A'First, Pos));
         pragma Loop_Variant (Increases => Pos);

         Pos := Pos + 1;
      end loop;
```

(continues on next page)

(continued from previous page)

```
27
28          Res := (Found => False);
29          return Res;
30       end Search;
31
32    end Linear_Search;
```

This concludes our tutorial on the SPARK toolset.

CHAPTER

SEVEN

FORMAL VERIFICATION WITH GNATPROVE

The GNATprove tool is packaged as an executable called `gnatprove`. Like other tools in GNAT toolsuite, GNATprove is based on the structure of GNAT projects, defined in `.gpr` files.

A crucial feature of GNATprove is that it interprets annotations exactly like they are interpreted at run time during tests. In particular, their executable semantics includes the verification of run-time checks, which can be verified statically with GNATprove. GNATprove also performs additional verifications on the specification of the expected behavior itself, and its correspondence to the code.

7.1 How to Run GNATprove

7.1.1 Setting Up a Project File

Basic Project Set Up

If not already done, create a GNAT project file (*.gpr*), as documented in the GNAT User's Guide, section *GNAT Project Manager*. See also *Project Attributes* for optional project attributes to specify the proof directory and other GNATprove switches in the project file directly.

Note that you can use the project wizard from GNAT Studio to create a project file interactively, via the menu *File* → *New Project...*. In the dialog, see in particular the default option (*Single Project*).

If you want to get started quickly, and assuming a standard naming scheme using `.ads/.adb` lower case files and a single source directory, then your project file will look like:

```
project My_Project is
   for Source_Dirs use (".");
end My_Project;
```

saved in a file called `my_project.gpr`.

Having Different Switches for Compilation and Verification

In some cases, you may want to pass different compilation-level switches to GNAT and GNATprove, for example use warning switches only for compilation, in the same project file. In that case, you can use a scenario variable to specify different switches for compilation and verification. We recommend to use the predefined `GPR_TOOL` variable for this purpose:

```
project My_Project is

  Mode := External ("GPR_TOOL", "");

  package Compiler is
     case Mode is
        when "gnatprove" =>
           for Switches ("Ada") use ...
        when others =>
           for Switches ("Ada") use ...
     end case;
  end Compiler;

end My_Project;
```

With the above project, compilation will be automatically done in the "normal" mode (the "others" branch above):

```
gprbuild -P my_project.gpr
```

while GNATprove automatically sets the `GPR_TOOL` variable to the `gnatprove` value:

```
gnatprove -P my_project.gpr
```

Other tools set the value of this variable to other values. See the documentation of other AdaCore tools to know more about this.

Note that before SPARK Pro 20, the `GPR_TOOL` was not set automatically by the tool. You can set it manually in this case:

```
gnatprove -P my_project.gpr -XGPR_TOOL=gnatprove
```

7.1.2 Running GNATprove from the Command Line

We recommend running GNATprove from the command line using a project file, as follows:

```
gnatprove -P <project-file.gpr>
```

In the appendix, section *Command Line Invocation*, you can find an exhaustive list of switches; here we only give an overview over the most common uses.

There are essentially three common ways you can select the files which will be analyzed by GNATprove:

- Analyze everything:

  ```
  gnatprove -P <project-file.gpr> -U
  ```

  With switch `-U`, all units of all projects in the project tree are analyzed. This includes units that are not used yet.

  This is usually what you want to use for an overnight analysis of a complex project.

- Analyze this project:

  ```
  gnatprove -P <project-file.gpr>
  ```

All main units in the specified project and all units they (recursively) depend on are analyzed. If there are no main units specified, analyze all files in the project. Note that main units of projects that the specified project depends on are not taken into account.

This is what you want to use for the analysis of a particular executable only, or if you want to analyze different executables within a complex project with different options.

- Analyze files:

```
gnatprove -P <project-file.gpr> [-u] FILES...
```

If `-u` is specified, we only analyze the units that contain the given files. If `-u` is not specified, we also analyze all units these units (recursively) depend on.

This usage is intended for the day-to-day command-line or IDE use of GNATprove when implementing a project.

Note that GNATprove always analyzes units as a whole, and cannot analyze a specification (`.ads`) file independently from a body (`.adb`) file. So if you specify a specification file that has a corresponding body, both are analyzed. The same is true for subunits such as separate subprograms: if you specify such a file name, the entire unit is analyzed.

GNATprove consists of two distinct analyses: flow analysis and proof. Flow analysis checks the correctness of aspects related to data flow (`Global`, `Depends`, `Abstract_State`, `Initializes`, and refinement versions of these), and verifies the initialization of variables. Proof verifies the absence of run-time errors and the correctness of assertions such as `Pre` and `Post` aspects. Using the switch `--mode=<mode>`, whose possible values are `check`, `check_all`, `flow`, `prove` `all`, `stone`, `bronze`, `silver` and `gold`, you can choose which analysis is performed:

- In mode `check`, GNATprove partially checks that the program does not violate SPARK restrictions. The benefit of using this mode prior to mode `check_all` is that it is much faster, as it does not require the results of flow analysis.
- In mode `check_all` (`stone` is a synonym for this mode), GNATprove fully checks that the program does not violate SPARK restrictions, including checks not performed in mode `check` like the absence of side effects in regular functions. Mode `check_all` includes mode `check`.
- In mode `flow` (`bronze` is a synonym for this mode), GNATprove checks that no uninitialized data are read in the program, and that the specified data dependencies and flow dependencies are respected in the implementation. Mode `flow` includes mode `check_all`. This phase is called *flow analysis*.
- In mode `prove` , GNATprove checks that the program is free from run-time errors, and that the specified functional contracts are respected in the implementation. Mode `prove` includes mode `check_all`, as well as the part of mode `flow` that checks that no uninitialized data are read, to guarantee soundness of the proof results. This phase is called *proof*.
- In the default mode `all`, GNATprove does both flow analysis and proof. The `silver` and `gold` modes are synonyms for this mode.

Using the option `--limit-line` one can limit proofs to a particular file and line of an Ada file. For example, if you want to prove only line 12 of file `example.adb`, you can add the option `--limit-line=example.adb:12` to the call to GNATprove. Using `--limit-region` one can limit proofs to a range of lines in a particular file. For example, `--limit-region=example.adb:12:14` will limit analysis to lines 12 to 14 in `example.adb`.

Using the `--limit-name` option, users can limit the analysis to subprograms that have a specific name. Note that this option doesn't change the set of units on which the analysis is run. For example, if a subprogram is outside of the closure of the main program, it will not be analyzed even if the `--limit-name` option with the corresponding name is provided.

The `--limit-name` option cannot distinguish between multiple subprograms that have the same name. Users can use the `--limit-subp` option, which expects a location. As an example, the option `--limit-subp=example.ads:12` limits the analysis to the subprogram declared at line 12 in `example.ads`. Note that `--limit-subp` implies analysis of the unit (`example.ads` in the example). If `example.ads` is a generic unit, SPARK skips analysis of such units by

default, as only instances of generics are analyzed. You can specify the switch `-U` in this case to analyze all instances of the generic subprogram.

One can specify a specific instance to analyze by specifying the place of instantiation: the option `--limit-subp=inst.adb:10:example.ads:12` analyzes the same subprogram, but only the instance that was created via the instantiation at line 10 of `inst.adb`. One can specify a longer chain if `inst.adb` is also part of a generic subprogram or package. In all cases, the `-U` switch is only needed if the first unit is a generic unit.

A number of options exist to influence the behavior for proof. Internally, the prover(s) specified with option `--prover` is/are called repeatedly for each check or assertion. Using the options `--timeout` and `--memlimit`, one can change the maximal time and memory that is allocated to each prover to prove each check or assertion. Using the option `--steps`, one can set the maximum number of reasoning steps that the prover is allowed to perform before giving up. The `--steps` option should be used when repeatable results are required, because the results with a timeout and memory limit may differ depending on the computing power, current load or platform of the machine. A default value of 100 for `--steps` is used if none of `--steps` or `--timeout` is used, either directly or indirectly through `--level`. The option `-j` activates parallel compilation and parallel proofs. With `-jnnn`, at most nnn cores can be used in parallel. With the special value `-j0`, at most N cores can be used in parallel, when N is the number of cores on the machine.

Note: When the project has a main file, or a file is passed as starting point to gnatprove, and the dependencies in the project are very linear (unit A depends only on unit B, which depends only on unit C, etc), then even when the `-j` switch is used, gnatprove may only consider one file at a time. This problem can be avoided by additionally using the `-U` switch.

Note: The –memlimit switch is currently ineffective on the Mac OS X operating system, due to limitations of the underlying system call on that system.

The way checks are passed to the prover can also be influenced using the option `--proof`. By default, the prover is invoked a single time for each check or assertion (mode `per_check`). This can be changed using mode `per_path` to invoke the prover for each *path* that leads to the check. This option usually takes much longer, because the prover is invoked much more often, but may give better proof results. Finally, in mode `progressive`, invoking the prover a single time on the entire check is tried, and only if the check is not proved, then other techniques that progressively consider each path in isolation are tried.

The proof mode set with `--proof` can be extended with a qualifier `all` or `lazy`, so that the entire switch may for example look like this: `--proof=progressive:all`. With this qualifier, one can select if proof should stop at the first unproved formula (to save time) for a check or should continue attempting to prove the other formulas related to the same check (typically to identify more precisely which formulas are left unproved, which can be then be handled with manual proof). The former is most suited for fully automatic proof, it is the default value, and can be explicitly selected with `lazy`. The latter is most suited for combination of automatic and manual proof and can be selected with `all`.

Instead of setting individually switches that influence the speed and power of proof, one may use the switch `--level`, which corresponds to predefined proof levels, from the faster level 0 to the more powerful level 4. More precisely, each value of `--level` is equivalent to directly setting a collection of other switches discussed above:

- `--level=0` is equivalent to `--prover=cvc5 --timeout=1 --memlimit=1000 --steps=0 --counterexamples=off`
- `--level=1` is equivalent to `--prover=cvc5,z3,altergo --timeout=1 --memlimit=1000 --steps=0 --counterexamples=off`
- `--level=2` is equivalent to `--prover=cvc5,z3,altergo --timeout=5 --memlimit=1000 --steps=0 --counterexamples=on`

- `--level=3` is equivalent to `--prover=cvc5,z3,altergo --timeout=20 --memlimit=2000 --steps=0 --counterexamples=on`
- `--level=4` is equivalent to `--prover=cvc5,z3,altergo --timeout=60 --memlimit=2000 --steps=0 --counterexamples=on`

If both `--level` is set and an underlying switch is set (`--prover`, `--timeout`, `--proof`, or `--counterexamples`), the value of the latter takes precedence over the value set through `--level`.

Note that using `--level` does not provide results that are reproducible accross different machines. For nightly builds or shared repositories, consider using the `--steps` or `--replay` switches instead. The number of steps required to proved an example can be accessed by running GNATprove with the option `--report=statistics`.

By default, GNATprove avoids reanalyzing unchanged files, on a per-unit basis. This mechanism can be disabled with the option `-f`.

When GNATprove proves a check, it stores this result in a session file, along with the required time and steps for this check to be proved. This information can be used to replay the proofs, to check that they are indeed correct. If such session files are present, and when GNATprove is invoked using the `--replay` option, it will attempt such a replay, using the same prover that was able to prove the check last time, with some slightly higher time and step limit. In this mode, the user-provided steps and time limits are ignored. If the `--prover` option is not provided, GNATprove will attempt to replay all checks, otherwise it will replay only the proofs proved by one of the specified provers. If all replays succeeded, GNATprove output will be exactly the same as a normal run of GNATprove. If a replay failed, the corresponding check will be reported as not proved. If a replay has not been attempted because the corresponding prover is not available (a third-party prover that is not configured, or the user has selected other provers using the `--prover` option), a warning will be issued that the proof could not be replayed, but the check will still be marked as proved.

By default, GNATprove stops at the first unit where it detect errors (violations of Ada or SPARK legality rules). The option `-k` can be used to get GNATprove to issue errors of the same kind for multiple units. If there are any violations of Ada legality rules, GNATprove does not attempt any analysis. If there are violations of SPARK legality rules, GNATprove stops after the checking phase and does not attempt flow analysis or proof.

GNATprove returns with a non-zero exit status when an error is detected. This includes cases where GNATprove issues unproved check messages when switch `--checks-as-errors=on` is used, as well as cases where GNATprove issues warnings when switch `--warnings=error` is used. In such cases, GNATprove also issues a message about termination in error. Otherwise, GNATprove returns with an exit status of zero, even when unproved check messages and warnings are issued.

7.1.3 Using the GNAT Target Runtime Directory

If you are using GNAT as your target compiler and explicitly specify a runtime and target to use in your project, for instance:

```
for Target use "arm-eabi";
for Runtime ("Ada") use "ravenscar-sfp-stm32f4";
```

GNATprove will take such setting into account and will use the GNAT runtime directory, as long as your target compiler is found in your PATH environment variable. Note that you will need to use a matching version of GNAT and SPARK (e.g. GNAT 18.2 and SPARK 18.2).

The handling of runtimes of GNATprove is in fact unified with that of the GNAT compiler. For details, see "GNAT User's Guide Supplement for Cross Platforms", Section 3. If you specify a target, note that GNATprove requires additional configuration, see the section *Specifying the Target Architecture and Implementation-Defined Behavior*.

If you're using GNAT Common Code Generator to generate C code from SPARK, you can specify the target and runtime as follows:

```
for Target use "c";
for Runtime ("Ada") use "ccg";
```

7.1.4 Specifying the Target Architecture and Implementation-Defined Behavior

A SPARK program is guaranteed to be unambiguous, so that formal verification of properties is possible. However, some behaviors (for example some representation attribute values like the `Size` attribute) may depend on the compiler used. By default, GNATprove adopts the same choices as the GNAT compiler. GNATprove also supports other compilers by providing special switches:

- `-gnateT` for specifying the target configuration
- `--pedantic` for warnings about possible implementation-defined behavior

Note that, even with switch `--pedantic`, GNATprove only detects some implementation-defined behaviors. For more details, see the dedicated section on how to *Ensure Portability of Programs*.

Note that GNATprove will always choose the smallest multiple of 8 bits for the base type, which is a safe and conservative choice for any Ada compiler.

Target Parameterization

If you specify the `Target` and `Runtime` attributes in your project file or via the `--target` and `--RTS` switches, GNATprove attempts to configure automatically the target dependent values such as endianness or sizes and alignments of standard types. If this automatic configuration fails, GNATprove outputs a warning and assumes that the compilation target is the same as the host on which it is run.

You can however configure the target dependent values manually. In addition to specifying the target and runtime via the project file or the commandline, you need to add the following to your project file, under a scenario variable as seen in *Having Different Switches for Compilation and Verification*:

```
project My_Project is
   [...]
   package Builder is
      case Mode is
         when "Compile" =>
            ...
         when "Analyze" =>
            for Global_Compilation_Switches ("Ada") use ("-gnateT=" & My_Project'Project_
↪Dir & "/target.atp");
      end case;
   end Builder;
end My_Project;
```

where `target.atp` is a file stored here in the same directory as the project file `my_project.gpr`, which contains the target parametrization. The format of this file is described in the GNAT User's Guide as part of the `-gnateT` switch description.

Target parameterization can be used:

- to specify a target different than the host on which GNATprove is run, when cross-compilation is used. If GNAT is the cross compiler and the automatic configuration fails, the configuration file can be generated by calling the compiler for your target with the switch `-gnatet=target.atp`. Otherwise, the target file should be generated manually.

- to specify the parameters for a different compiler than GNAT, even when the host and target are the same. In that case, the target file should be generated manually.

Here is an example of a configuration file for a bare board PowerPC 750 processor configured as big-endian:

```
Bits_BE                       1
Bits_Per_Unit                 8
Bits_Per_Word                32
Bytes_BE                      1
Char_Size                     8
Double_Float_Alignment        0
Double_Scalar_Alignment       0
Double_Size                  64
Float_Size                   32
Float_Words_BE                1
Int_Size                     32
Long_Double_Size             64
Long_Long_Size               64
Long_Size                    32
Maximum_Alignment            16
Max_Unaligned_Field          64
Pointer_Size                 32
Short_Enums                   0
Short_Size                   16
Strict_Alignment              1
System_Allocator_Alignment    8
Wchar_T_Size                 32
Words_BE                      1

float           6  I  32  32
double         15  I  64  64
long double    15  I  64  64
```

7.1.5 Running GNATprove from GNAT Studio

GNATprove can be run from GNAT Studio. When GNATprove is installed and found on your PATH, a *SPARK* menu is available with the following entries:

Submenu	Action
Examine All	This runs GNATprove in flow analysis mode on all mains and the units they depend on in the project.
Examine All Sources	This runs GNATprove in flow analysis mode on all files in the project.
Examine File	This runs GNATprove in flow analysis mode on the current unit, its body and any subunits.
Prove All	This runs GNATprove on all mains and the units they depend on in the project.
Prove All Sources	This runs GNATprove on all files in the project.
Prove File	This runs GNATprove on the current unit, its body and any subunits.
Show Report	This displays the report file generated by GNATprove.
Clean Proofs	This removes all files generated by GNATprove.
Show Previous Runs	This displays previous runs of GNATprove.

The three "Prove…" entries run GNATprove in the mode given by the project file, or in the default mode "all" if no

mode is specified.

The menus *SPARK → Examine/Prove All* run GNATprove on all main files in the project, and all files they depend on (recursively). Both main files in the root project and in projects that are included in the root project are considered. The menus *SPARK → Examine/Prove All Sources* run GNATprove on all files in all projects. On a project that has neither main files nor includes other projects, menus *SPARK → Examine/Prove All* and *SPARK → Examine/Prove All Sources* are equivalent.

The menu *SPARK → Show Previous Runs* gives access to the results of previous runs of GNATprove on the project, up to a bound which can be set using the *Edit → Preferences* dialog in GNAT Studio, and opening the *Plugins → Gnatprove Runs* section. Note that the higher this bound, the more disk space will be used to store the results of previous runs of GNATprove.

Keyboard shortcuts for these menu items can be set using the *Edit → Preferences* dialog in GNAT Studio, and opening the *General → Key Shortcuts* section.

Note: The changes made by users in the panels raised by these submenus are persistent from one session to the other. Be sure to check that the selected checkboxes and additional switches that were previously added are still appropriate.

When editing an Ada file, GNATprove can also be run from a *SPARK* contextual menu, which can be obtained by a right click:

Submenu	Action
Examine File	This runs GNATprove in flow analysis mode on the current unit, its body and any subunits.
Examine Subprogram	This runs GNATprove in flow analysis mode on the current subprogram.
Prove File	This runs GNATprove on the current unit, its body and any subunits.
Prove Subprogram	This runs GNATprove on the current subprogram.
Prove Line	This runs GNATprove on the current line.
Prove Selected Region	This runs GNATprove on the currently selected region.
Prove Check	This runs GNATprove on the current failing condition. GNATprove must have been run at least once for this option to be available in order to know which conditions are failing.
Globals	This generates Global contracts for the current file.

Except from *Examine File*, *Prove File*, and *Globals*, all other submenus are also applicable to code inside generic units, in which case the corresponding action is applied to all instances of the generic unit in the project. For example, if a generic unit is instantiated twice, selecting *Prove Subprogram* on a subprogram inside the generic unit will apply proof to the two corresponding subprograms in instances of the generic unit.

The menus *SPARK → Examine ...* open a panel which allows setting various switches for GNATprove's analysis. The main choice offered in this panel is to select the mode of analysis, among modes `check`, `check_all` (which corresponds to the `stone` analysis mode) and `flow` (the default, which corresponds to the `bronze` analysis mode).

The menus *SPARK → Prove ...* open a panel which allows setting various switches for GNATprove's analysis, corresponding to the `silver` and `gold` analysis modes. By default, this panel offers a few simple choices, like the proof level (see description of switch `--level` in *Running GNATprove from the Command Line*). If the user changes its `User profile` for SPARK (in the SPARK section of the Preferences dialog - menu *Edit → Preferences*) from `Basic` to `Advanced`, then a more complex panel is displayed for proof, with more detailed switches.

GNATprove project switches can be edited from the panel `GNATprove` (menu *Edit → Project Properties*, in the *Build → Switches* section of the dialog).

When proving a check fails on a specific path through a subprogram (for both checks verified in flow analysis and in proof), GNATprove may generate path information for the user to see. The user can display this path in GNAT Studio

by clicking on the icon to the left of the failed proof message, or to the left of the corresponding line in the editor. The path is hidden again when re-clicking on the same icon.

The contextual menu *SPARK → Globals → ...* allows the user to show and hide the Global contracts that are internally generated by GNATprove for the current unit. This can be useful when learning how to write *Data Dependencies*, because the tool provides the contracts where they are missing. Note that this does not modify the unit source code - the Global contracts are inserted into a special buffer; the buffer contents can be copy-pasted into the editor if desired.

For checks verified in proof, GNATprove may also generate counterexample information for the user to see (see *Understanding Counterexamples*). The user can display this counterexample in GNAT Studio by clicking on the icon to the left of the failed proof message, or to the left of the corresponding line in the editor. The counterexample is hidden again when re-clicking on the same icon.

A monospace font with ligature like Fira Code (https://github.com/tonsky/FiraCode) or Hasklig (https://github.com/i-tu/Hasklig) can be separately installed and selected to make contracts more readable inside GNAT Studio or GNATbench. See the following screenshot which shows how symbols like => (arrow) or >= (greater than or equal) are displayed in such a font:

```
10  function LCP (A : Text; X, Y : Integer) return Natural with
11    SPARK_Mode,
12    Pre  ⇒ X in A'Range and then Y in A'Range,
13    Post ⇒
14      (for all K in 0 .. LCP'Result - 1 ⇒ A (X + K) = A (Y + K))
15        and then (X + LCP'Result = A'Last + 1
16                    or else Y + LCP'Result = A'Last + 1
17                    or else A (X + LCP'Result) /= A (Y + LCP'Result)),
18    Contract_Cases ⇒
19       (A (X) /= A (Y) ⇒ LCP'Result = 0,
20        X = Y          ⇒ LCP'Result = A'Last - X + 1,
21        others         ⇒ LCP'Result ≥ 0);
```

7.1.6 Running GNATprove from Visual Studio Code

GNATprove can be run from Visual Studio Code, using the Ada/SPARK extension for Visual Studio Code. It provides the following "auto-detected" tasks (under menu *Terminal → Run Task...*):

Submenu	Action
Examine project	This runs GNATprove in flow analysis mode on all mains and the units they depend on in the project.
Examine file	This runs GNATprove in flow analysis mode on the current unit, its body and any subunits.
Examine subprogram	This runs GNATprove in flow analysis mode on the current subprogram.
Prove project	This runs GNATprove on all mains and the units they depend on in the project.
Prove file	This runs GNATprove on the current unit, its body and any subunits.
Prove subprogram	This runs GNATprove on the current subprogram.
Prove selected region	This runs GNATprove on the currently selected region.
Prove line	This runs GNATprove on the current line.

7.1.7 Running GNATprove from GNATbench

GNATprove can be run from GNATbench. When GNATprove is installed and found on your PATH, a *SPARK* menu is available with the following entries:

Submenu	Action
Examine All	This runs GNATprove in flow analysis mode on all mains and the units they depend on in the project.
Examine All Sources	This runs GNATprove in flow analysis mode on all files in the project.
Examine File	This runs GNATprove in flow analysis mode on the current unit, its body and any subunits.
Prove All	This runs GNATprove on all mains and the units they depend on in the project.
Prove All Sources	This runs GNATprove on all files in the project.
Prove File	This runs GNATprove on the current unit, its body and any subunits.
Show Report	This displays the report file generated by GNATprove.
Clean Proofs	This removes all files generated by GNATprove.

The three "Prove…" entries run GNATprove in the mode given by the project file, or in the default mode "all" if no mode is specified.

The menus *SPARK* → *Examine/Prove All* run GNATprove on all main files in the project, and all files they depend on (recursively). Both main files in the root project and in projects that are included in the root project are considered. The menus *SPARK* → *Examine/Prove All Sources* run GNATprove on all files in all projects. On a project that has neither main files nor includes other projects, menus *SPARK* → *Examine/Prove All* and *SPARK* → *Examine/Prove All Sources* are equivalent.

Note: The changes made by users in the panels raised by these submenus are persistent from one session to the other. Be sure to check that the selected checkboxes and additional switches that were previously added are still appropriate.

When editing an Ada file, GNATprove can also be run from a *SPARK* contextual menu, which can be obtained by a right click:

Submenu	Action
Examine File	This runs GNATprove in flow analysis mode on the current unit, its body and any subunits.
Examine Subprogram	This runs GNATprove in flow analysis mode on the current subprogram.
Prove File	This runs GNATprove on the current unit, its body and any subunits.
Prove Subprogram	This runs GNATprove on the current subprogram.
Prove Line	This runs GNATprove on the current line.
Globals	This generates Global contracts for the current file.

7.1.8 Running GNATprove Without a Project File

GNATprove accepts the invocation without a project file (`-P` switch on the command line). In that case, if the current directory contains exactly one project file, GNATprove uses this project file. If no project file exists, GNATprove creates a trivial project file with the name `default.gpr` and uses that.

7.1.9 GNATprove and Manual Proof

When automated provers fail to prove some condition that is valid, the validity may be proved using manual proof inside GNAT Studio or an external interactive prover.

In the appendix, section *Alternative Provers*, is explained how to use different provers than the one GNATprove uses as default.

Calling an Interactive Prover From the Command Line

When the prover used by GNATprove is configured as interactive, for each analysed condition, either:

- It is the first time the prover is used on the condition then a file (containing the condition as input to the specified prover) is created in the project's proof directory (see *Project Attributes*). GNATprove outputs a message concerning this condition indicating the file that was created. The created file should be edited by the user in order to prove the condition.
- The prover has already been used on this condition and the editable file exists. The prover is run on the file and the success or failure of the proof is reported in the same way it is done with the default prover.

Note: Once a manual proof file is created and has been edited by the user, in order to run the prover on the file, the same prover must be once again specified to GNATprove. Once the condition is proved, the result will be saved in the why3 session so GNATprove won't need to be specified the prover again to know that the condition is valid.

Analysis with GNATprove can be limited to a single condition with the `--limit-line` option:

```
gnatprove -P <project-file.gpr> --prover=<prover> --limit-line=<file>:<line>:<column>:
↪<check-kind>
```

Please use the output of `gnatprove --list-categories` to determine the `check-kind` to be provided in this command.

Calling an Interactive Prover From GNAT Studio

After running GNATprove with proof mode, the menu *SPARK → Prove Check* is available by right-clicking on a check message in the location tab or by right-clicking on a line that fails because of a single condition (i.e. there is only one check in the output of GNATprove concerning this line).

In the dialog box, the field "Alternate prover" can be filled to use another prover than Alt-Ergo. If the alternative prover is configured as "interactive", after the execution of *SPARK → Prove Check*, GNAT Studio opens the manual proof file with the editor corresponding to the prover under the condition that an editor is specified in the configuration of the alternative prover.

Once the editor is closed, GNAT Studio re-executes *SPARK → Prove Check*. The user should verify the same alternative prover as before is still specified. After execution, GNAT Studio will offer to re-edit the file if the proof fails.

7.1.10 How to Speed Up a Run of GNATprove

GNATprove can take some time on large programs with difficult checks to prove. This section describes how one can improve the running time of the GNATprove tool. Note that some of the suggested settings will decrease the number of proved checks or decrease usability of the tool, because spending more time often results in more successful proofs. You may still want to try some of the suggestions here to see if the time spent by GNATprove is really useful in your context.

These settings will speed up GNATprove:

- Use the `-j` switch to use more than one core on your machine. GNATprove can make efficient usage of multi-processing. If your machine has more than one processor or core, we strongly suggest to enable multi-processing, using the `-j` switch. This switch should not have an impact on proof results, only on running time.
- Use `--no-loop-unrolling` to deactivate loop unrolling. Loop unrolling can often avoid the use of a loop invariant, but it almost always will be more costly to analyze than a loop with a loop invariant. See also *Automatic Unrolling of Simple For-Loops*.
- Use `--no-inlining` to deactivate contextual analysis of local subprograms without contracts. This feature can often avoid the use of subprogram contracts, but it will be more costly to analyze such subprograms in their calling context than analyzing them separately. See also *Contextual Analysis of Subprograms Without Contracts*.
- Use `--counterexamples=off` to deactive counterexamples. Counter-examples are very useful to understand the reason for a failed proof attempt. You can disable this feature if you are not working on a failed proof attempt.
- Use the `--level` switch to use a lower level and faster preset. Generally, a lower level is faster than higher levels. See also *Running GNATprove from the Command Line*.
- More fine-grained than the `--level` switch, you can directly set the `--prover`, `--timeout` and `--steps` options. Using only one prover with a small timeout or a small steps limit will result in much faster execution.
- If you have access to up-to-date session files, (see *Running GNATprove from the Command Line*) and you only want to check the proof results of the stored session, you can use `--replay`. Replay only runs previously successful provers and is therefore much faster than a run of GNATprove without this option.
- Use `--function-sandboxing=off`. By default, GNATprove sandboxes functions to limit the impact of *Infeasible Subprogram Contracts*. These guards have a non-negligible impact on prover performance. If in your project, all subprograms are in the SPARK subset, or you have confidence in the contracts you wrote for the subprograms which are not in SPARK, you can disable these guards using the `--function-sandboxing=off` option.
- Use `--memcached-server` switch for *Sharing Proof Results Via a Cache*.

7.1.11 GNATprove and Network File Systems or Shared Folders

On Linux and Mac-OS, GNATprove needs to create a Unix domain socket file. This might be a problem if GNATprove attempts to create such a file in a directory that is a shared folder or on a network file system like NFS, which does not support such folders. To minimize chances for this to occur, GNATprove determines the folder to create that special file as follows:

- if the environment variable `TMPDIR` is set, and the corresponding directory exists and is writeable, use that; otherwise,
- if `/tmp` exists and is writable, use that; otherwise,
- use the `gnatprove` subfolder of the object directory of the root project.

On Linux, GNATprove uses POSIX semaphores to coordinate parallel processes. If your system does not provide POSIX semaphores (this may be the case in some virtualized environments), GNATprove fails with a message similar to the following:

```
failed to create semaphore: Permission denied
```

In this case, you can use the switch *–debug-no-semaphore* to avoid the use of semaphores. This switch might reduce the performance of the tool in some cases, but otherwise should not affect its behavior.

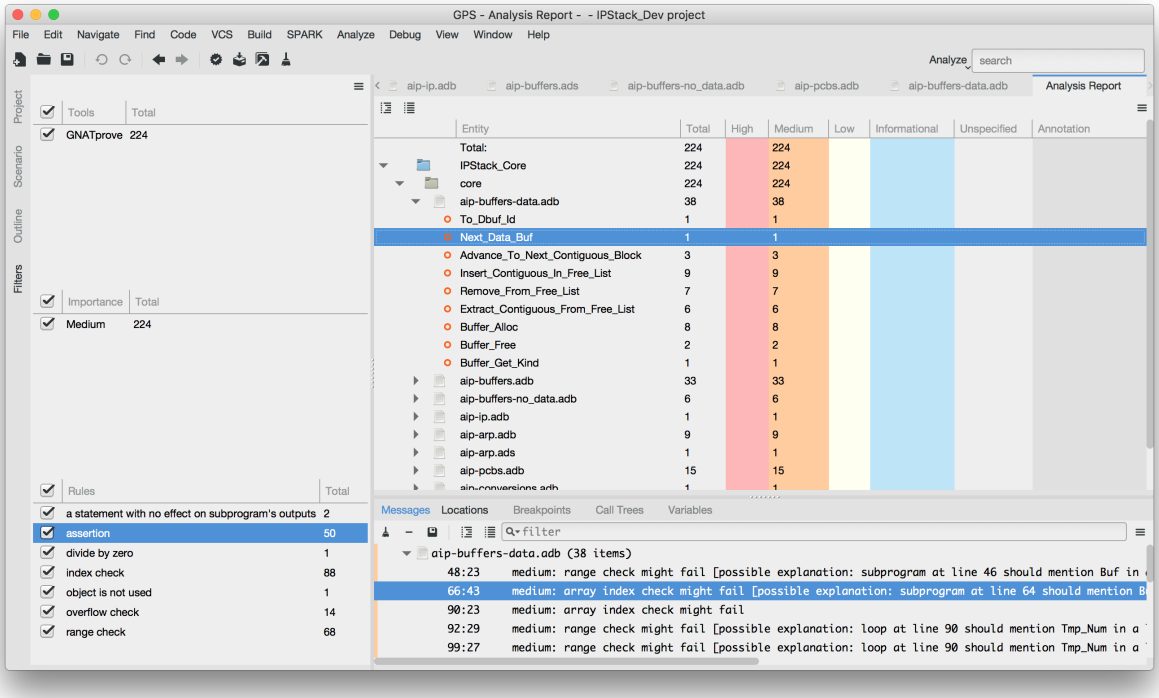
7.2 How to View GNATprove Output

GNATprove produces two kinds of outputs: the one which is echoed to standard output or displayed in your IDE (GNAT Studio or GNATbench), and a textual summary of the analysis results.

7.2.1 The Analysis Report Panel

GNAT Studio can display an interactive view reporting the results of the analysis, with a count of issues per file, subprogram and severity, as well as filters to selectively view a subset of the issues only. This interactive view is displayed using the menu *SPARK → Show Report*. This menu becomes available after the checkbox `Display analysis report` is checked in the SPARK section of the Preferences dialog - menu *Edit → Preferences*, and only if GNATprove was run so that there are results to display.

Here is an example of this view:



7.2.2 The Analysis Results Summary File

GNATprove generates global project statistics in file `gnatprove.out`, which can be displayed in GNAT Studio using the menu *SPARK → Show Log*. The file `gnatprove.out` is generated in the `gnatprove` subdirectory of the object directory of the project.

When switch `--output-header` is used, this file starts with a header containing extra information about the run including:

- The date and time of GNATprove run
- The GNATprove version that has generated this report
- The host for which GNATprove is configured (e.g. Windows 32 bits)
- The full command-line of the GNATprove invocation, including project file
- The GNATprove switches specified in the project file

A summary table at the start of file `gnatprove.out` provides an overview of the verification results for all checks in the project. The table may look like this:

```
-------------------------------------------------------------------------------------------------
↪-------------------------
SPARK Analysis results        Total          Flow   Interval                       ␣
↪Provers   Justified   Unproved
-------------------------------------------------------------------------------------------------
↪-------------------------
Data Dependencies               281           281          .                           ␣
↪  .           .          .
Flow Dependencies               228           228          .                           ␣
↪  .           .          .
Initialization                  693           692          .                           ␣
↪  .           1          .
Non-Aliasing                      .             .          .                           ␣
↪  .           .          .
Run-time Checks                 474             .          .      458 (CVC5 95%, Trivial␣
↪5%)          16          .
Assertions                       45             .          .       45 (CVC5 82%, Trivial␣
↪18%)           .          .
Functional Contracts            304             .          .     302 (CVC5 82%, Trivial␣
↪18%)           2          .
LSP Verification                  .             .          .                           ␣
↪  .           .          .
Termination                       .             .          .                           ␣
↪  .           .          .
Concurrency                       .             .          .                           ␣
↪  .           .          .
-------------------------------------------------------------------------------------------------
↪-------------------------
Total                          2025    1201 (59%)          .                      805␣
↪(40%)     19 (1%)          .
```

The following table explains the lines of the summary table:

Line Description	Explanation
Data Dependencies	Verification of *Data Dependencies* and parameter modes
Flow Dependencies	Verification of *Flow Dependencies*
Initialization	Verification of *Data Initialization Policy*
Non-Aliasing	Verification of *Absence of Interferences*
Run-time Checks	Verification of absence of run-time errors (AoRTE) (except those raising `Storage_Error`)
Assertions	Verification of *Assertion Pragmas*
Functional Contracts	Verification of functional contracts (includes *Subprogram Contracts*, *Package Contracts* and *Type Contracts*)
LSP Verification	Verification related to *Object Oriented Programming and Liskov Substitution Principle*
Termination	Verification related to *Loop Variants* and *Subprogram Termination*
Concurrency	Verification related to *Concurrency and Ravenscar Profile*

We now explain the columns of the table.

- The `Total` column describes the total number of checks in this category.
- The `Flow` column describes the number of checks proved by flow analysis.
- The `Interval` column describes the number of checks (overflow and range checks) proved by a simple static analysis of bounds for floating-point expressions based on type bounds of sub-expressions.
- The `Provers` column describes the number of checks proved by automatic or manual provers. The column also gives information on the provers used, and the percentage of checks proved by each prover. Note that sometimes a check is proved by a combination of provers, hence the use of percentage instead of an absolute count. Also note that generally the prover which is run first (as determined by the `--prover` command line switch) proves the most checks, because each prover is called only on those checks that were not previously proved. The prover percentages are provided in alphabetical order. The special name `Trivial` is used to refer to an internal simplication that discards checks that are trivially true.
- The `Justified` column contains the number of checks for which the user has provided a *Direct Justification with Pragma Annotate*.
- Finally, the column `Unproved` counts the checks which have neither been proved nor justified.

After the summary table, a line states the maximal steps that were consumed by automated provers. The line may look like this:

```
max steps used for successful proof: 1234
```

The use of this line is to help with reproducability of a run of GNATprove that proved all checks and properties. If the user provides the given number via the `--steps` option to GNATprove, and disables the time and memory limits, (if enabled directly or indirectly such as via the `--level` switch), then GNATprove will again prove all checks and properties. For example, if a user has proved all checks in a project using an invocation of GNATprove as follows:

```
gnatprove -P <projectfile> --level=2
```

then the following command will also prove all checks:

```
gnatprove -P <projectfile> --level=2 --timeout=0 --memlimit=0 --steps=1234
```

The next contents in the file are statistics describing:

- which units were analyzed (with flow analysis, proof, or both)
- if the analysis for a given unit was incomplete because of errors
- which subprograms in these units were analyzed (with flow analysis, proof, or both)

- the results of this analysis

7.2.3 Categories of Messages

GNATprove issues four different kinds of messages: errors, warnings, check messages and information messages.

- Errors are issued for SPARK violations or other language legality problems, or any other problem which does not allow to proceed to analysis. Errors cannot be suppressed and must be fixed to proceed with analysis.
- Warnings are issued for any suspicious situation like unused values of variables, useless assignments, etc. Warnings are prefixed with the text `"warning: "` and can be suppressed with `pragma Warnings`, see section *Suppressing Warnings*.
- Check messages are issued for any potential problem in the code which could affect the correctness of the program, such as missing initialization, possible failing run-time checks or unproved assertions. Checks come with a severity, and depending on the severity the message text is prefixed with `"low: "`, `"medium: "` or `"high: "`. Check messages cannot be suppressed like warnings, but they can be individually justified with pragma `Annotate`, see section *Justifying Check Messages*.
- Information messages are issued to notify the user of limitations of GNATprove on some constructs, or to prevent possible confusion in understanding the output of GNATprove. They are also issued to report proved checks in some modes of GNATprove.

7.2.4 Errors and Completeness of Analysis

As mentioned in the previous section, if errors are encountered (manifested by an error message), the analysis may not be complete.

At the project level, GNATprove analyzes the units of a project independently, and stops when a unit contains an error. Other units may not be analyzed in this case. You can use the command line switch *-k* to analyze all units even in the presence of errors. Of course this may take more time.

At the unit level, errors in earlier phases stop the analysis and block the display of other errors. If this happens, the `gnatprove.out` file contains information about this, for example:

```
flow analysis and proof skipped for this unit (error during ownership checking)
```

It can be confusing to try to fix errors that come from later stages (e.g. proof) while errors in earlier stages are still present (e.g. in other units). Therefore we recommend a gradual approach, eliminating simpler errors before going to more advanced errors. This can be achieved by usind the `--mode` switch, which is explained in detail in the next section.

7.2.5 Effect of Mode on Output

GNATprove can be run in four different modes, as selected with the switch `--mode=<mode>`, whose possible values are `check`, `check_all`, `flow`, `prove` and `all` (see *Running GNATprove from the Command Line*). The output depends on the selected mode.

In modes `check` and `check_all`, GNATprove prints on the standard output a list of error messages for violations of SPARK restrictions on all the code for which `SPARK_Mode` is `On`.

In modes `flow` and `prove`, this checking is done as a first phase.

In mode `flow`, GNATprove prints on the standard output messages for possible reads of uninitialized data, mismatches betwen the specified data dependencies and flow dependencies and the implementation, and suspicious situations such as unused assignments and missing return statements. These messages are all based on flow analysis.

In mode `prove`, GNATprove prints on the standard output messages for possible reads of uninitialized data (using flow analysis), possible run-time errors and mismatches between the specified functional contracts and the implementation (using proof).

In mode `all`, GNATprove prints on the standard output both messages for mode `flow` and for mode `prove`.

If switch `--report=all`, `--report=provers` or `--report=statistics` is specified, GNATprove additionally prints on the standard output information messages for proved checks.

7.2.6 Description of Messages

This section lists the different messages which GNATprove may output. Each message points to a very specific place in the source code. For example, if a source file `file.adb` contains a division as follows:

```
if X / Y > Z then ...
```

GNATprove may output a message such as:

```
file.adb:12:37: medium: divide by zero might fail
```

where the division sign `/` is precisely on line 12, column 37. Looking at the explanation in the first table below, which states that a division check verifies that the divisor is different from zero, it is clear that the message is about `Y`, and that GNATprove was unable to prove that `Y` cannot be zero. The explanations in the table below should be read with the context that is given by the source location.

When switch `--cwe` is used, a corresponding CWE id is included in the message when relevant. For example, on the example above, GNATprove will output a message such as:

```
file.adb:12:37: medium: divide by zero might fail [CWE 369]
```

Note that CWE ids are only included in check messages and warnings, never in information messages about proved checks. For more information on CWE, see the MITRE Corporation's Common Weakness Enumeration (CWE) Compatibility and Effectiveness Program (http://cwe.mitre.org/). The current version of GNATprove is based on CWE version 3.2 released on January 3, 2019.

Messages reported by Proof

The following table shows all check messages reported by proof.

Message Kind	CWE	Explanation
run-time checks		
divide by zero	CWE 369	Check that the second operand of the division, mod or rem operation is different from zero.
index check	CWE 119	Check that the given index is within the bounds of the array.
overflow check	CWE 190	Check that the result of the given integer arithmetic operation is within the bounds of the base type.
fp_overflow check	CWE 739	Check that the result of the given floating point operation is within the bounds of the base type.
range check	CWE 682	Check that the given value is within the bounds of the expected scalar subtype.
predicate check	CWE 682	Check that the given value respects the applicable type predicate.

continues on next page

Table 1 – continued from previous page

Message Kind	CWE	Explanation
predicate check on default value	CWE 682	Check that the default value for the type respects the applicable type predicate.
null pointer dereference	CWE 476	Check that the given pointer is not null so that it can be dereferenced.
null exclusion		Check that the subtype_indication of the allocator does not specify a null_exclusion
dynamic accessibility check		Check that the accessibility level of the result of a traversal function call is not deeper than the accessibility level of its traversed parameter.
resource or memory leak	CWE 772	Check that the assignment does not lead to a resource or memory leak
resource or memory leak at end of scope	CWE 772	Check that the declaration does not lead to a resource or memory leak
unchecked union restriction		Check restrictions imposed on expressions of an unchecked union type
length check		Check that the given array is of the length of the expected array subtype.
discriminant check	CWE 136	Check that the discriminant of the given discriminated record has the expected value. For variant records, this can happen for a simple access to a record field. But there are other cases where a fixed value of the discriminant is required.
tag check	CWE 136	Check that the tag of the given tagged object has the expected value.
ceiling priority in Interrupt_Priority		Check that the ceiling priority specified for a protected object containing a procedure with an aspect Attach_Handler is in Interrupt_Priority.
use of an uninitialized variable	CWE 457	Check that a variable is initialized
interrupt is reserved		Check that the interrupt specified by Attach_Handler is not reserved.
invariant check		Check that the given value respects the applicable type invariant.
invariant check on default value		Check that the default value for the type respects the applicable type invariant.
ceiling priority protocol		Check that the ceiling priority protocol is respected, i.e., when a task calls a protected operation, the active priority of the task is not higher than the priority of the protected object (Ada RM Annex D.3).
task termination		Check that the task does not terminate, as required by Ravenscar.
assertions		
initial condition		Check that the initial condition of a package is true after elaboration.
default initial condition		Check that the default initial condition of a type is true after default initialization of an object of the type.
precondition	CWE 628	Check that the precondition aspect of the given call evaluates to True.
precondition of main	CWE 628	Check that the precondition aspect of the given main procedure evaluates to True after elaboration.

continues on next page

Table 1 – continued from previous page

Message Kind	CWE	Explanation
postcondition	CWE 682	Check that the postcondition aspect of the subprogram evaluates to True.
refined postcondition	CWE 682	Check that the refined postcondition aspect of the subprogram evaluates to True.
contract case	CWE 682	Check that all cases of the contract case evaluate to true at the end of the subprogram.
disjoint contract cases		Check that the cases of the contract cases aspect are all mutually disjoint.
complete contract cases		Check that the cases of the contract cases aspect cover the state space that is allowed by the precondition aspect.
exceptional case	CWE 682	Check that all cases of the exceptional cases evaluate to true on exceptional exits.
loop invariant		Check that the loop invariant evaluates to True on all iterations of the loop.
loop invariant in first iteration		Check that the loop invariant evaluates to True on the first iteration of the loop.
loop invariant after first iteration		Check that the loop invariant evaluates to True at each further iteration of the loop.
loop variant	CWE 835	Check that the given loop variant decreases/increases as specified during each iteration of the loop. This implies termination of the loop.
subprogram variant		Check that the given subprogram variant decreases/increases as specified during each recursive call. This implies there will be no infinite recursion.
assertion		Check that the given assertion evaluates to True.
assertion step		Check that the side-condition of a cut operation evaluates to True.
assertion premise		Check that the premise of an assertion with cut operations evaluates to True.
raised exception		Check that raise expressions can never be reached and that all exceptions raised by raise statement and procedure calls are expected.
feasible function		Check that an abstract function or access-to-function type is feasible.
Inline_For_Proof or Logical_Equal annotation		Check that an Annotate pragma with the Inline_For_Proof or Logical_Equal identifier is correct.
Container_Aggregates annotation		Check the invariants used to translate container aggregates using the primitives provided by the Aggregate aspect and the Container_Aggregates annotation.
termination check		Check the termination of subprograms annotated with an Always_Terminates aspect whose value is not known at compile time and of calls to such subprograms.
unchecked conversion source check	CWE 843	Check that a source type in an unchecked conversion can safely be used for such conversions. This means that the memory occupied by objects of this type is fully used by the object.

continues on next page

Table 1 – continued from previous page

Message Kind	CWE	Explanation
unchecked conversion target check	CWE 843	Check that a target type in an unchecked conversion can safely be used for such conversions. This means that the memory occupied by objects of this type is fully used by the object, and no invalid bitpatterns occur.
unchecked conversion size check	CWE 843	Check that the two types in an unchecked conversion instance are of the same size.
alignment check		Check that the first object's alignment is an integral multiple of the second object's alignment.
volatile overlay check		Check that, if an object has an address clause that is not simply the address of another object, it is volatile
Liskov Substitution Principle		
precondition weaker than class-wide precondition		Check that the precondition aspect of the subprogram is weaker than its class-wide precondition.
precondition not True while class-wide precondition is True		Check that the precondition aspect of the subprogram is True if its class-wide precondition is True.
postcondition stronger than class-wide postcondition		Check that the postcondition aspect of the subprogram is stronger than its class-wide postcondition.
class-wide precondition weaker than overridden one		Check that the class-wide precondition aspect of the subprogram is weaker than its overridden class-wide precondition.
class-wide postcondition stronger than overridden one		Check that the class-wide postcondition aspect of the subprogram is stronger than its overridden class-wide postcondition.
precondition of the source weaker than precondition of the target		Check that the precondition aspect of the access-to-subprogram type used as the target of a conversion implies the precondition of the source.
postcondition of the source stronger than postcondition of the target		Check that the postcondition aspect of the access-to-subprogram type used as the target of a conversion is implied by the postcondition of the source.

The following table shows all warning messages reported by proof when using switch `--proof-warnings`.

Message Kind	CWE	Explanation
precondition always False	CWE 570	Warn if precondition is found to be always False
postcondition always False	CWE 570	Warn if postcondition is found to be always False
pragma Assume always False	CWE 570	Warn if pragma Assume is found to be always False
unreachable branch	CWE 561	Warn if branch is found to be unreachable
unreachable code	CWE 561	Warn if code is found to be unreachable

Messages reported by Flow Analysis

The following table shows all flow analysis messages, (E)rrors, (W)arnings and (C)hecks.

Message Kind	Class	CWE	Explanation
critically incomplete Global or Initializes contract	E		A Global or Initializes contract fails to mention some objects.
volatile function wrongly declared as non-volatile	E		A volatile function wrongly declared as non-volatile.
function with side effects	E		A function with side effects.
aliasing between subprogram parameters	C		Aliasing between formal parameters or global objects.
invalid call in type invariant	C	CWE 674	A type invariant calls a boundary subprogram for the type.
invalid context for call to Current_Task	C		Current_Task is called from an invalid context.
race condition	C	CWE 362	An unsynchronized global object is accessed concurrently.
wrong Default_Initial_Condition aspect	C	CWE 457	A type is wrongly declared as initialized by default.
input item missing from the dependency clause	C		An input is missing from the dependency clause.
output item missing from the dependency clause	C		An output item is missing from the dependency clause.
input item missing from the null dependency clause	C		An input item is missing from the null dependency clause.
extra input item in the dependency clause	C		Extra input item in the dependency clause.
subprogram output depends on a Proof_In global	C		Subprogram output depends on a Proof_In global.
non-ghost output of ghost subprogram	C		A ghost subprogram has a non-ghost global output.
incomplete Global or Initializes contract	C		A Global or Initializes contract fails to mention some objects.
an extra item in the Global or Initializes contract	C		A Global or Initializes contract wrongly mentions some objects.
constants with variable inputs that is not a state constituent	C		Constants with variable inputs that are not state constituents.
illegal write of a global input	C		Illegal write of a global input.
an extra item in the Initializes contract	C		An object that shall not appear in the Initializes contract.
all execution paths raise exceptions or do not return	C		All execution paths raise exceptions or do not return.
illegal write of an object declared as constant after elaboration	C		Illegal write of an object declared as constant after elaboration.
protected operation blocks	C	CWE 667	A protected operation may block.
illegal reference to a global object in precondition of a protected operation	C		An illegal reference to global in precondition of a protected operation.
constant with no variable inputs as an abstract state's constituent	C		Constant with no variable inputs as an abstract state's constituent.

continues on next page

Table 2 – continued from previous page

Message Kind	Class	CWE	Explanation
subprogram with aspect Always_Terminates may not terminate	C	CWE 674	A subprogram with aspect Always_Terminates may not terminate.
use of an uninitialized variable	C	CWE 457	Flow analysis has detected the use of an uninitialized variable.
global object is not used	C		A global object is never used.
dead code	W	CWE 561	A statement is never executed.
a state abstraction that is impossible to initialize	W		A state abstraction that is impossible to initialize.
a statement with no effect on subprogram's outputs	W	CWE 1164	A statement with no effect on subprogram's outputs.
an IN OUT parameter or an In_Out global that is not written	W		An IN OUT parameter or an In_Out global that is not written.
loop with stable statement	W		A loop with stable statement.
object is not used	W	CWE 563	A parameter or locally declared object is never used.
initial value of an object is not used	W	CWE 563	The initial value of an object is not used.
non-volatile function wrongly declared as volatile	W		A non-volatile function wrongly declared as volatile.

Note: Certain messages emitted by flow analysis are classified as errors and consequently cannot be suppressed or justified.

Miscellaneous warnings reported by GNATprove

The following table shows default warnings reported by GNATprove outside of flow analysis and proof.

Message Kind	Explanation
address to access conversion	call to conversion function is assumed to return a valid access designating a valid value
volatile and atomic status of aliases	aliased objects should both be volatile or non-volatile, and both be atomic or non-atomic
volatile properties of aliases	aliased objects should have the same volatile properties
attribute Valid always True	attribute Valid is assumed to return True
initialization of alias	initialization of object is assumed to have no effects on other non-volatile objects
function Is_Valid always return True	function Is_Valid is assumed to return True
procedure not terminating normally nor abnormally	?procedure which does not return normally nor raises an exception cannot always terminate
no check message justified	no check message justified by this pragma
proved check message justified	only proved check messages justified by this pragma
Terminating deprecated	Terminating, Always_Return, and Might_Not_Return annotations are ignored
External Axiomatizations not supported	External Axiomatizations are not supported anymore, ignored
pragma ignored	pragma is ignored (it is not yet supported)
Overflow_Mode ignored	pragma Overflow_Mode in code is ignored
precondition statically False	precondition is statically False
unreferenced function	analyzing unreferenced function
unreferenced procedure	analyzing unreferenced procedure
useless Relaxed_Initialization aspect on function result	function result annotated with Relaxed_Initialization cannot be partially initialized
useless Relaxed_Initialization aspect on object	object annotated with Relaxed_Initialization cannot be partially initialized
variant not recursive	no recursive call visible on subprogram with Subprogram_Variant

The following table shows warnings guaranteed to be reported by GNATprove.

Message Kind	Explanation
assumed Always_Terminates	no Always_Terminates aspect available for subprogram, subprogram is assumed to always terminate
assumed Global null	no Global contract available for subprogram, null is assumed
imprecisely supported address specification	object with an imprecisely supported address specification: non-atomic objects should not be accessed concurrently, volatile properties should be correct, indirect writes to object to and through potential aliases are ignored, and reads should be valid

The following table shows warnings reported by GNATprove when using switch `--pedantic`.

Message Kind	Explanation
string attribute length	string attribute has an implementation-defined length
operator reassociation	possible operator reassociation due to missing parentheses
representation attribute value	representation attribute has an implementation-defined value

Messages of a specific category or related to a specific CWE can be filtered inside GNAT Studio by typing the desired substring in the search bar of the `Locations` panel. For example, search for "CWE" to get all messages with a corresponding CWE, or "CWE 369" to get all messages related to division by zero vulnerability.

7.2.7 Understanding Counterexamples

When a check cannot be proved, GNATprove may generate a counterexample when switch `--counterexamples=on` is used, either explicitly or implicitly through the use of switch `--level`. A counterexample consists in two parts:

- a path (or set of paths) through the subprogram
- an assignment of values to variables that appear on that path

The best way to look at a counterexample is to display it in GNAT Studio by clicking on the icon to the left of the failed proof message, or to the left of the corresponding line in the editor (see *Running GNATprove from GNAT Studio*). GNATprove then displays the path in one color, and the values of variables on the path by inserting lines in the editor only (not in the file) which display these values. For example, consider procedure `Counterex`:

```
1  procedure Counterex (Cond : Boolean; In1, In2 : Integer; R : out Integer) with
2    SPARK_Mode,
3    Pre => In1 <= 25 and In2 <= 25
4  is
5  begin
6     R := 0;
7     if Cond then
8        R := R + In1;
9        if In1 < In2 then
10          R := R + In2;
11          pragma Assert (R < 42);
12       end if;
13    end if;
14 end Counterex;
```

The assertion on line 11 may fail when input parameter `Cond` is `True` and input parameters `I1` and `I2` are too big. The counterexample generated by GNATprove is displayed as follows in GNAT Studio, where each line highlighted in the path is followed by a line showing the value of variables from the previous line:

counterex.adb

```
1  procedure Counterex (Cond : Boolean; In1, In2 : Integer; R : out Integer) with
   --  Cond = True and In1 = 17 and In2 = 25 and R = 0
2    SPARK_Mode,
3    Pre => In1 <= 25 and In2 <= 25
4  is
5  begin
6     R := 0;
      --  R = 0
7     if Cond then
8        R := R + In1;
         --  R = 17
9        if In1 < In2 then
10          R := R + In2;
            --  R = 42
11          pragma Assert (R < 42);
            --  R = 42
12       end if;
13    end if;
14 end Counterex;
```

GNATprove also completes the message for the failed proof with an explanation giving the values of variables from the checked expression for the counterexample. Here, the message issued by GNATprove on line 11 gives the value of output parameter R:

```
counterex.adb:11:25: high: assertion might fail
   11 |         pragma Assert (R < 42);
      |                        ^~~~~
  e.g. when R = 42
```

To limit the time spent trying to generate counterexamples, GNATprove sets a small timeout to prover cvc5 for generating counterexamples. It can be replaced by a number of reasoning steps in the prover by using the switch `--ce-steps`. Generation of counterexamples is deterministic, hence the use of `--ce-steps` ensures that results are repeatable.

By default, GNATprove internally checks that counterexamples correctly point to a problem in the code or in the contracts (which includes missing preconditions, loop invariants, etc.) so that only interesting counterexamples are displayed to the user. As a result, some counterexamples from cvc5 are not displayed to the user, while in other cases the ranking of messages might be promoted from `medium` to `high` after checking that the counterexample points to an actual problem in the code. In cases where the counterexample generated by cvc5 is dropped, this checking phase also tries to generate a candidate counterexample by fuzzing input values of the subprogram, based on extreme values of input types. If a candidate counterexample passes the checking phase, it is displayed in place of the original counterexample. This checking phase can be skipped with the switch `--check-counterexamples=off`.

The counterexample generated by GNATprove does not always correspond to a feasible execution of the program, in particular when using the switch `--check-counterexamples=off`:

1. When some contracts or loop invariants are missing, thus causing the property to become unprovable (see details in section on *Investigating Unprovable Properties*), the counterexample may help point to the missing contract or loop invariant. For example, the postcondition of procedure `Double_In_Call` is not provable because the postcondition of the function `Double` that it calls is too weak, and the postcondition of procedure `Double_In_Loop` is not provable because its loop does not have a loop invariant:

```
1  package Counterex_Unprovable with
2    SPARK_Mode
3  is
4
5     type Int is new Integer range -100 .. 100;
```

(continues on next page)

(continued from previous page)

```
6
7     function Double (X : Int) return Int with
8       Pre  => abs X <= 10,
9       Post => abs Double'Result <= 20;
10
11    procedure Double_In_Call (X : in out Int) with
12      Pre  => abs X <= 10,
13      Post => X = 2 * X'Old;
14
15    procedure Double_In_Loop (X : in out Int) with
16      Pre  => abs X <= 10,
17      Post => X = 2 * X'Old;
18
19 end Counterex_Unprovable;
```

```
1  package body Counterex_Unprovable with
2    SPARK_Mode
3  is
4
5     function Double (X : Int) return Int is
6     begin
7        return 2 * X;
8     end Double;
9
10    procedure Double_In_Call (X : in out Int) is
11    begin
12       X := Double (X);
13    end Double_In_Call;
14
15    procedure Double_In_Loop (X : in out Int) is
16       Result : Int := 0;
17       J      : Integer := 1;
18    begin
19       while J <= 2 loop
20          Result := Result + X;
21          J      := J + 1;
22       end loop;
23       X := Result;
24    end Double_In_Loop;
25
26 end Counterex_Unprovable;
```

The counterexample generated by GNATprove in the first case shows that the prover could deduce wrongly that `X` on ouput is 0 when its value is 1 on input, due to a missing contract in the called function:

```
counterex_unprovable.ads:13:14: medium: postcondition might fail
   13 |      Post => X = 2 * X'Old;
      |              ^~~~~~~~~~~~~
  e.g. when X = 0
        and X'Old = 1
  possible fix: add or complete related loop invariants or postconditions
```

(continues on next page)

(continued from previous page)

```
counterex_unprovable.ads:17:14: medium: postcondition might fail
   17 |     Post => X = 2 * X'Old;
      |             ^~~~~~~~~~~~~
  e.g. when X = -1
        and X'Old = 0
  possible fix: add or complete related loop invariants or postconditions
```

Similarly, the counterexample generated by GNATprove in the second case shows that the prover could deduce wrongly that `X` on ouput is -1 when its value is 0 on input, due to a missing loop invariant in the executed loop.

2. When some property cannot be proved due to prover shortcomings (see details in section on *Investigating Prover Shortcomings*), the counterexample may explain why the prover cannot prove the property. However, note that since the counterexample is always generated only using cvc5 prover, it can just explain why this prover cannot prove the property. Also note that if cvc5 is not selected and generating of a counterexample is enabled by `--counterexamples=on` switch (explicitly or implicitly through the use of `--level` switch), a counterexample is still attempted to be generated using cvc5, but the proof result of cvc5 is not taken into account in this case.
3. When using a short value of timeout or steps, the prover may hit the resource bound before it has produced a full counterexample. In such a case, the counterexample produced may not correspond to a feasible execution.
4. When the value of `--proof` switch is `per_check` (the default value), then the counterexample may give values to variables on all paths through the subprogram, not only the path which corresponds to the feasible execution. One can rerun GNATprove with value `progressive` or `per_path` to separate possible execution paths in the counterexample.

7.3 How to Use GNATprove in a Team

The most common use of GNATprove is as part of a regular quality control or quality assurance activity inside a team. Usually, GNATprove is run every night on the current codebase, and during the day by developers either on their computer or on servers. For both nightly and daily runs, GNATprove results need to be shared between team members, either for viewing results or to compare new results with the shared results. These various processes are supported by specific ways to run GNATprove and share its results.

In all cases, the source code should not be shared directly (say, on a shared drive) between developers, as this is bound to cause problems with file access rights and concurrent accesses. Rather, the typical usage is for each user to do a check out of the sources/environment, and use therefore her own version/copy of sources and project files, instead of physically sharing sources across all users.

The project file should also always specify a local, non shared, user writable directory as object directory (whether explicitly or implicitly, as the absence of an explicit object directory means the project file directory is used as object directory).

7.3.1 Possible Workflows

Multiple workflows allow to use GNATprove in a team:

1. GNATprove is run on a server or locally, and no warnings or check messages should be issued. Typically this is achieved by suppressing spurious warnings and justifying unproved check messages.
2. GNATprove is run on a server or locally, and textual results are shared in Configuration Management.
3. GNATprove is run on a server, and textual results are sent to a third-party qualimetry tool (like GNATdashboard, SonarQube, SQUORE, etc.)

4. GNATprove is run on a server or locally, and the GNATprove session files are shared in Configuration Management.

In all workflows (but critically for the first workflow), messages can be suppressed or justified. Indeed, like every sound and complete verification tool, GNATprove may issue false alarms. A first step is to identify the type of message:

- warnings can be suppressed, see *Suppressing Warnings*
- check messages can be justified, see *Justifying Check Messages*

Check messages from proof may also correspond to provable checks, which require interacting with GNATprove to find the correct contracts and/or analysis switches, see *How to Investigate Unproved Checks*.

The textual output in workflow 3 corresponds to the compiler-like output generated by GNATprove and controlled with switches `--report` and `--warnings` (see *Running GNATprove from the Command Line*). By default messages are issued only for unproved checks and warnings.

The textual output in workflow 2 comprises this compiler-like output, and possibly additional output generated by GNATprove in file `gnatprove.out` (see *Effect of Mode on Output* and *Managing Assumptions*).

Workflow 4 is explained in more detail in *Sharing Proof Results with Others*.

7.3.2 Suppressing Warnings

GNATprove warnings are controlled with switch `--warnings`:

- `--warnings=off` suppresses all warnings
- `--warnings=error` treats warnings as errors
- `--warnings=continue` issues warnings but does not stop analysis (default)

The default is that GNATprove issues warnings but does not stop.

Warnings can be suppressed selectively by the use of pragma `Warnings` in the source code. For example, GNATprove issues three warnings on procedure `Warn`, which are suppressed by the three pragma `Warnings` in the source code:

```
pragma Warnings (Off, "unused initial value of ""X""",
                 Reason => "Parameter mode is mandated by API");

procedure Warn (X : in out Integer) with
  SPARK_Mode
is
   pragma Warnings (Off, "initialization has no effect",
                    Reason => "Coding standard requires initialization");
   Y : Integer := 0;
   pragma Warnings (On, "initialization has no effect");

begin
   pragma Warnings (Off, "unused assignment",
                    Reason => "Test program requires double assignment");
   X := Y;
   pragma Warnings (On, "unused assignment");
   X := Y;
end Warn;
```

Warnings with the specified message are suppressed in the region starting at pragma `Warnings Off` and ending at the matching pragma `Warnings On` or at the end of the file (pragma `Warnings` is purely textual, so its effect does not stop at the end of the enclosing scope). The `Reason` argument string is optional. A regular expression can be given instead

of a specific message in order to suppress all warnings of a given form. Pragma `Warnings Off` can be added in a configuration file to suppress the corresponding warnings across all units in the project. Pragma `Warnings Off` can be specified for an entity to suppress all warnings related to this entity.

Additionally, aspect `Warnings Off` on a static stand-alone constant object can be used to suppress warnings about unreachable code due to a "statically disabled" condition of an IF statement. This is useful when using configuration constants, which may trigger spurious warnings about unreachable code.

A "statically disabled" condition which evaluates to Value is either:

- a static stand-alone constant when it is of a boolean type, has aspect `Warnings Off` and its initial value evaluates to Value
- a *relational_operator* where one operand is static stand-alone constant with aspect `Warnings Off`, the other operand is a literal of the corresponding type and the operator evaluates to Value
- an `and` or `and then` operators when:
  - Value is True and both operands are statically disabled conditions that evaluate to True
  - Value is False and at least one operand is a statically disabled condition that evaluates to False
- an `or` or `or else` operators when:
  - Value is True and at least one operand is a statically disabled condition that evaluates to True
  - Value is False and both operands are statically disabled conditions that evaluate to False
- a *not* operator when the right operand is a statically disabled condition that evaluates to the negation of Value

Pragma `Warnings` can also take a first argument of `GNATprove` to specify that it applies only to GNATprove. For example, the previous example can be modified to use these refined pragma `Warnings`:

```
pragma Warnings (GNATprove, Off, "unused initial value of ""X""",
                 Reason => "Parameter mode is mandated by API");

procedure Warn2 (X : in out Integer) with
  SPARK_Mode
is
   pragma Warnings (GNATprove, Off, "initialization has no effect",
                    Reason => "Coding standard requires initialization");
   Y : Integer := 0;
   pragma Warnings (GNATprove, On, "initialization has no effect");

begin
   pragma Warnings (GNATprove, Off, "unused assignment",
                    Reason => "Test program requires double assignment");
   X := Y;
   pragma Warnings (GNATprove, On, "unused assignment");
   X := Y;
end Warn2;
```

Besides the documentation benefit of using this refined version of pragma `Warnings`, it makes it possible to exclude such pragma `Warnings` from the detection of useless pragma `Warnings`, that do not suppress any warning at compilation, with compilation switch `-gnatw.w`. Indeed, this switch can then be used during compilation with GNAT, as pragma `Warnings` that apply only to GNATprove can be identified as such.

See the GNAT Reference Manual for more details.

Additionally, GNATprove can issue warnings as part of proof, on preconditions or postconditions or pragma `Assume` that are always false, unreachable branches in complex Boolean expressions (typically in assertions and contracts), dead

code at branching points in the program. These warnings are not enabled by default, as they require calling a prover for each potential warning, which incurs a small cost (1 sec for each property thus checked). They can be enabled with switch `--proof-warnings`, and their effect is controlled by switch `--warnings` and pragma `Warnings` as described previously.

There are two benefits of activating these warnings:

- they may detect unintentional unreachable or useless code and assertions, which may originate from errors in either code or assertions;
- they strengthen confidence in the tool output, acting as a *smoke detector* for cases where the tool would get into an inconsistent context by error, and report some unreachable code or branch where there is none.

Note that GNATprove, just like GNAT, suppresses warnings about unused variables if their name contains any of the substrings DISCARD, DUMMY, IGNORE, JUNK, UNUSED, in any casing.

7.3.3 Suppressing Information Messages

Information messages can be suppressed by the use of pragma `Warnings` in the source code, like for warnings.

7.3.4 Justifying Check Messages

GNATprove's analysis relies on the fact that, at any given point in the program, previous checks on any execution reaching that program point have been successful. Thus, given two successive assertions of the same property:

```
pragma Assert (Prop);  --  possibly not proved
pragma Assert (Prop);  --  proved
```

The second assertion will be reported as proved by GNATprove, even if the first assertion is reported as not proved. This is because any execution that fails the first assertion is not analyzed further by GNATprove.

Similarly, consider two successive calls to the same procedure with a precondition:

```
Proc (Args);  --  precondition possibly not proved
Proc (Args);  --  precondition proved
```

The precondition of the second call will be reported as proved by GNATprove, even if the precondition of the first call is reported as not proved. This is because any execution that fails the first precondition is not analyzed further by GNATprove.

This applies to all proof checks, and to a lesser extent to flow analysis checks. For example, outputs of a subprogram are considered fully initialtized in a caller, as explained in *Data Initialization Policy*. In particular, such outputs are considered to have values that respect the constraints of their type, which is used during proof.

Thus, the user should be careful when justifying check messages, as the incorrect justification of a check message that could fail could also hide other possible failures later for the same execution of the analyzed program.

Direct Justification with Pragma Annotate

Check messages generated by GNATprove's flow analysis or proof can be selectively justified by adding a pragma `Annotate` in the source code. For example, the check message about a possible division by zero in the return expression below can be justified as follows:

```
return (X + Y) / (X - Y);
pragma Annotate (GNATprove, False_Positive,
                 "divide by zero", "reviewed by John Smith");
```

The pragma has the following form:

```
pragma Annotate (GNATprove, Category, Pattern, Reason);
```

where the following table explains the different entries:

Item	Explanation
GNATprove	is a fixed identifier
Category	is one of `False_Positive` or `Intentional`
Pattern	is a string literal describing the pattern of the check messages which shall be justified
Reason	is a string literal providing a justification for reviews

All arguments should be provided.

The *Category* currently has no impact on the behavior of the tool but serves a documentation purpose:

- `False_Positive` indicates that the check cannot fail, although GNATprove was unable to prove it.
- `Intentional` indicates that the check can fail but that it is not considered to be a bug.

Pattern is a pattern that is used to match against the text of the check message to justify (not including the initial `"low:  "`, `"medium:  "` or `"high:  "` prefix). The pattern follows the same rules as for pragma `Warnings`. It may contain asterisks, which match zero or more characters in the message, and no other characters are interpreted as regular expression notations (it is not necessary to put an asterisk at the start and the end of the message, since this is implied). The match is case insensitive.

Reason is a string provided by the user as a justification for reviews. This reason may be present in a GNATprove report.

Placement rules are as follows: in a statement list or declaration list, pragma `Annotate` applies to the preceding item in the list, ignoring other pragma `Annotate`. If there is no preceding item, the pragma applies to the enclosing construct. For example, if the pragma is the first element of the then-branch of an if-statement, it will apply to condition in the if-statement.

If the preceding or enclosing construct is a subprogram body, the pragma applies to both the subprogram body and the spec including its contract. This allows to place a justification for a check message issued by GNATprove either on the spec when it is relevant for callers. Note that this placement of a justification is ineffective on subprograms analyzed only in the context of their calls (see details in *Contextual Analysis of Subprograms Without Contracts*).

An aspect on a package or subprogram declaration/body can be used instead of a pragma at the beginning of the corresponding declaration list inside the declaration/body:

```
package Pack with
  Annotate => (GNATprove, False_Positive,
               "divide by zero", "reviewed by John Smith")
```

(continues on next page)

(continued from previous page)

```
is
   ...

procedure Proc with
  Annotate => (GNATprove, False_Positive,
               "divide by zero", "reviewed by John Smith")
is
   ...
```

As a point of caution, the following placements of pragma Annotate will apply the pragma to a possibly large range of source lines:

- when the pragma appears in a statement list after a block, it will apply to the entire block (e.g. an if statement including all branches, or a loop including the loop body).
- when the pragma appears directly after a subprogram body, it will apply to the entire body and the spec of the subprogram.

Users should take care to not justify checks which were not intended to be justified, when placing pragma Annotate in such places.

```
   procedure Do_Something_1 (X, Y : in out Integer) with
     Depends => ((X, Y) => (X, Y));
   pragma Annotate (GNATprove, Intentional, "incorrect dependency ""Y => X""",
                    "Dependency is kept for compatibility reasons");
```

or on the body when it is an implementation choice that need not be visible to users of the unit:

```
   procedure Do_Something_2 (X, Y : in out Integer) with
     Depends => ((X, Y) => (X, Y));
```

```
   procedure Do_Something_2 (X, Y : in out Integer) is
   begin
      X := X + Y;
      Y := Y + 1;
   end Do_Something_2;
   pragma Annotate (GNATprove, Intentional, "incorrect dependency ""Y => X""",
                    "Currently Y does not depend on X, but may change later");
```

Pragmas `Annotate` of the form above that do not justify any check message are useless and result in a warning by GNATprove. Like other warnings emitted by GNATprove, this warning is treated like an error if the switch `--warnings=error` is set.

Indirect Justification with Pragma Assume

Check messages generated by GNATprove's proof can alternatively be justified indirectly by adding a *Pragma Assume* in the source code, which allows the check to be proved. For example, the check message about a possible integer overflow in the assignment statement below can be justified as follows:

```
   procedure Next_Tick is
   begin
      pragma Assume (Clock_Ticks < Natural'Last,
                     "Device uptime is short enough that Clock_Ticks is less than 1_000␣
```

(continues on next page)

(continued from previous page)

```
↪always");
      Clock_Ticks := Clock_Ticks + 1;
   end Next_Tick;
```

Using pragma `Assume` is more powerful than using pragma `Annotate`, as the property assumed may be used to prove more than one check. Thus, one should in general use pragma `Annotate` rather than pragma `Assume` to justify simple runtime checks. There are some cases though where using a pragma `Assume` may be preferred. In particular:

- To keep assumptions local:

```
pragma Assume (<External_Call's precondition>,
               "because for these internal reasons I know it holds");
External_Call;
```

  If the precondition of `External_Call` changes, it may not be valid anymore to assume it here, though the assumption will stay True for the same reasons it used to be. Incompatible changes in the precondition of `External_Call` will lead to a failure in the proof of External_Call's precondition.

- To sum up what is expected from the outside world so that it can be reviewed easily:

```
External_Find (A, E, X);
pragma Assume (X = 0 or (X in A'Range and A (X) = E),
               "because of the documentation of External_Find");
```

  Maintenance and review is easier with a single pragma `Assume` than if it is spread out into various pragmas `Annotate`. If the information is required at several places, the pragma `Assume` can be factorized into a procedure:

```
function External_Find_Assumption (A : Array, E : Element, X : Index) return Boolean
is (X = 0 or (X in A'Range and A (X) = E))
with Ghost;

procedure Assume_External_Find_Assumption (A : Array, E : Element, X : Index) with
 Ghost,
 Post => External_Find_Assumption (A, E, X)
is
   pragma Assume (External_Find_Assumption (A, E, X),
                  "because of the documentation of External_Find");
end Assume_External_Find_Assumption;

External_Find (A, E, X);
Assume_External_Find_Assumption (A, E, X);
```

In general, assumptions should be kept as small as possible (only assume what is needed for the code to work). Indirect justifications with pragma `Assume` should be carefully inspected as they can easily introduce errors in the verification process.

7.3.5 Sharing Proof Results with Others

GNATprove stores proof results in so-called session files. If session files are shared with others (e.g. via Configuration Management), either between members of the same team, or between the developer and users of a library, others can reproduce/recheck the proofs using the `--replay` option of GNATprove.

For a single project, proof results are typically stored in many session files, each of them having the filename `why3session.xml`. To avoid name clashes, the files are stored in subdirectories that correspond to subparts of the project (such as unit names and subprogram names). By default, these directories are stored in the `gnatprove` subdirectory of the object directory of the project. If the `Proof_Dir` attribute is set in the *Project Attributes*, the session directories will be stored in a `sessions` subdirectory of this directory. To generate session files, GNATprove should be run without the `--replay` option.

To share the session files, we recommend adding the `why3session.xml` files to version control. Note that the session directories may contain other files (the so-called shapes files `why3shapes` or `why3shapes.gz`). We advise against adding these other files to version control. To avoid version control conflicts, it can be advantageous to avoid updates to the session files by each developer, and instead update these files periodically using a centralized mechanism. For example, a nightly run on a server, or a dedicated team member, can be responsible for updating the proof directory with the latest version generated by GNATprove.

If a user has access to the session files (e.g. via the just-described version control) for a project, he can use the `--replay` option to reproduce/recheck the proofs that are stored in the session files. See *Running GNATprove from the Command Line* for more details on this command line option.

7.3.6 Sharing Proof Results Via a Cache

GNATprove can cache and share results between distinct runs of the tool. This feature can be enabled using the `--memcached-server` switch. This switch accepts two arguments separated by a colon, and there are two different forms:

- The switch is of the form `--memcached-server=file:<directory>`, that is, the part before the colon is the string `file`, and the part after it is a directory that exists.
- The switch is of the form `--memcached-server=<hostname>:<portnumber>`, with the hostname being different from "file".

If the switch is of the first form, GNATprove uses the specified directory to store results between runs of the tool. Note that this directory will tend to grow over time and should be deleted and recreated from time to time.

If the switch is of the second form, GNATprove will attempt to connect to a Memcached server (see https://memcached.org/) located at the specified hostname and port, to cache intermediate results between runs.

In both cases, significant speedups can be observed after the cache is filled with an initial GNATprove run.

7.3.7 Managing Assumptions

Because GNATprove analyzes separately subprograms and packages, its results depend on assumptions about other subprograms and packages. For example, the verification that a subprogram is free from run-time errors depends on the property that all the subprograms it calls implement their specified contract. If a program is completely analyzed with GNATprove, GNATprove will report messages on those other subprograms, if they might not implement their contract correctly. But in general, a program is partly in SPARK and partly in other languages, mostly Ada, C and assembly languages. Thus, assumptions on parts of the program that cannot be analyzed with GNATprove need to be recorded for verification by other means, like testing, manual analysis or reviews.

Partial Listing of Detailed Assumptions

When switch `--assumptions` is used, GNATprove generates information about remaining assumptions in its result file `gnatprove.out`. These remaining assumptions need to be justified to ensure that the desired verification objectives are met. An assumption on a subprogram may be generated in various cases:

- the subprogram was not analyzed (for example because it is marked `SPARK_Mode => Off`)
- the subprogram was not completely verified by GNATprove (that is, some unproved checks remain)

Note that currently, only assumptions on called subprograms are output, and not assumptions on calling subprograms.

The following table explains the meaning of assumptions and claims which gnatprove may output:

Assumption	Explanation
effects on parameters and global variables	The subprogram does not read or write any other parameters or global variables than what is described in its spec (signature + data dependencies).
absence of run-time errors	The subprogram is free from run-time errors.
the postcondition	The postconditon of the subprogram holds after each call of the subprogram.

Complete List of Assumptions

For the sake of these assumptions, we define a *precisely supported address specification* to be an address clause or aspect whose expression is a reference to the Address attribute on a part of a standalone object or constant. We define an *imprecisely supported address specification* to be an address clause or aspect that is not a precisely supported address specification.

For the sake of these assumptions, we define an *object with an imprecisely supported address* to be either a standalone object with an address clause or aspect that is an imprecisely supported address specification or an object that is a *reachable part* of an object with an imprecisely supported address (a component of a composite value or the designated object of an access value).

The following assumptions need to be addressed when using SPARK on all or part of a program:

- [SPARK_JUSTIFICATION] All justifications of check messages should be reviewed (see *Justifying Check Messages*), both when using *Direct Justification with Pragma Annotate* and when using *Indirect Justification with Pragma Assume*.
- [SPARK_EXTERNAL] The modeling of *Interfaces to the Physical World* needs to be reviewed for objects whose value may be modified concurrently.
  - They should be *effectively volatile* in SPARK (see SPARK RM 7.1.2), so that GNATprove takes into account possible concurrent changes in the object's value. The warning *imprecisely supported address specification* is guaranteed to be issued in cases where review is required.
  - They should be *synchronized* in SPARK (see SPARK RM 9) to prevent race conditions which could lead to reading invalid values. The warning *imprecisely supported address specification* is guaranteed to be issued in cases where review is required.
  - They should have specified all necessary *Properties of Volatile Variables* corresponding to their usage. The warning *imprecisely supported address specification* is guaranteed to be issued in cases where review is required.
- [SPARK_ALIASING_ADDRESS] Aliases between objects with an imprecisely supported address specification are ignored by GNATprove. Reviews are necessary to ensure that:

  - The objects themselves are annotated with the `Asynchronous_Writers` volatile property if they can be affected by the modification of another object. The warning *imprecisely supported address specification* is guaranteed to be issued in cases where review is required.
  - Other objects visible from SPARK code which might be affected by a modification of such a variable have the `Asynchronous_Writers` volatile property set to True. The warning *imprecisely supported address specification* is guaranteed to be issued in cases where review is needed.
  - Other objects visible from SPARK code which might be affected by a modification of such a variable have valid values for their type when read. The warning *imprecisely supported address specification* is guaranteed to be issued in cases where review is needed.

- [SPARK_VALID] Attribute 'Valid is currently assumed to always return True, as no invalid value can be constructed in SPARK (see *Data Validity*). If assumptions [SPARK_ALIASING_ADDRESS], [SPARK_EXTERNAL_VALID], and [ADA_EXTERNAL] are satisfied, then this assumption will be satisfied as well. However, it is valuable to explicitly state this assumption because it highlights an important consequence of compliance with the other assumptions.
- [SPARK_EXTERNAL_VALID] Values read from objects whose address is specified are assumed to be valid values. This assumption is limited to objects with an imprecisely supported address (because an explicit check is emitted otherwise). Currently there is no model of invalidity or undefinedness. The onus is on the user to ensure that all values read from an external source are valid. The use of an invalid value invalidates any proofs associated with the value. The warning *imprecisely supported address specification* is guaranteed to be issued in cases where review is required.
- [SPARK_STORAGE_ERROR] As explained in section *Dealing with Storage_Error*, GNATprove does not issue messages about possible memory exhaustion, which leads to raising exception `Storage_Error` at runtime. The computation of suitable stack and heap sizes should be performed independently.
- [SPARK_TARGET_AND_RUNTIME] When the target configuration and runtime library for running the program are different from those on the host when GNATprove is run, the target configuration (see *Specifying the Target Architecture and Implementation-Defined Behavior*) and runtime library (see *Using the GNAT Target Runtime Directory*) should be set, so that GNATprove correctly interprets the behavior of the program at runtime.
- [SPARK_FLOATING_POINT] When using floating-point numbers, GNATprove relies on the *Semantics of Floating Point Operations* as defined in IEEE-754. The compiler, OS, and hardware should all be configured so that IEEE-754 semantics are respected.
- [SPARK_COMPILATION_SWITCHES] Compilation switches that change the behavior of the program should be the same between compilation and analysis. This is in particular the case for *Overflow Modes*.
- [SPARK_ITERABLE] When a type is annotated with an `Iterable` aspect:
  - the function `Has_Element` shall be such that, for any container object `Container` and cursor object `Cursor`, `Has_Element (Container, Cursor)` only evaluates to True if `Cursor` is accessible from `First (Container)` using the function `Next`, and
  - for any container object `Container`, the iteration from `First (Container)` through the function `Next` shall reach a cursor `Cursor` for which `Has_Element (Container, Cursor)` evaluates to False in a finite number of steps.
- [SPARK_ITERABLE_FOR_PROOF] When a type has an `Iterable_For_Proof` annotation,
  - the function `Contains` shall be such that, for any container object `Container` and any element `E`, `Contains (Container, E)` evaluates to True if and only if there is a cursor object `Cursor` such that `Has_Element (Container, Cursor)` evaluates to True and `E` is the result of `Element (Container, Cursor)`, or
  - the function `Model` shall be such that, for any container object `Container` and any element `E`, there is a cursor object `Cursor` such that `Has_Element (Container, Cursor)` evaluates to True and `E` is the result of `Element (Container, Cursor)` if and only if there is a cursor object `M_Cursor` for the model

type such that `Has_Element (Model (Container), M_Cursor)` evaluates to True and `E` is the result of `Element (Model (Container), M_Cursor)`.

- [SPARK_INITIALIZED_ATTRIBUTE] GNATprove assumes that the `Initialized` attribute is not referenced in any SPARK code that is executed. This assumption is necessary because evaluation of the `Initialized` attribute during execution is based on `Valid_Scalars`, and `Valid_Scalars` sometimes evaluates to True on uninitialized data. Note that, despite this assumption, it can be valuable during testing to execute contracts and other ghost code that references the `Initialized` attribute, as long as the executable code of the product itself does not reference the `Initialized` attribute.
- [SPARK_OVERRIDING_AND_TASKING] If there are overriding operations called using a dispatching call, then GNATprove assumes that the overriding operation does not have any adverse tasking-related effects. In particular, GNATprove assumes that the overriding operation:
  - does not call protected entries,
  - does not suspend on suspension objects,
  - does not lock protected objects with calls to protected subprograms,
  - does not call Ada.Task_Identification.Current_Task.

In addition, the following assumptions need to be addressed when using SPARK on only part of a program:

- [ADA_TASKING] If entry points for concurrent tasks (either OS tasks or units of computations scheduled by a runtime component) are not identified as tasks in SPARK, then during each invocation of a SPARK subprogram from such a task such that the SPARK subprogram is not being called directly or indirectly from another SPARK subprogram in the same task, the Global contract and by-reference parameters of the subprogram shall not conflict with either (a) the Global contract and by-reference parameters of any other such subprogram executing concurrently in another such task or (b) the Global contract of any concurrent task identified as a task in SPARK. Two global objects and/or by-reference parameters referring to the same object are said to conflict if both (1) they are not both synchronized and (2) at least one can be modified by the callee.

  In addition, calls from SPARK units to subprograms which are not analyzed by GNATprove should not have any adverse tasking-related effects. In particular, GNATprove assumes that such calls do not cause tasks visible from SPARK to:
  - call protected entries that they are not calling in a way which is visible from SPARK,
  - suspend on suspension objects on which they do not suspend in a way which is visible from SPARK.
- [ADA_EXTERNAL] Objects accessed outside of SPARK, either directly for statically allocated objects, or through their address or a pointer for all objects, should comply with the assumptions described in [SPARK_EXTERNAL], [SPARK_ALIASING_ADDRESS] and [SPARK_EXTERNAL_VALID].
- [ADA_EXTERNAL_ABSTRACT_STATE] The modeling of *Interfaces to the Physical World* needs to be reviewed for abstract states whose value may be modified concurrently, when their refinement is not in SPARK. These abstract states should comply with the assumptions described in [SPARK_EXTERNAL].
- [ADA_EXTERNAL_NAME] Objects annotated with an aspect `External_Name` or `Link_Name` should comply with the assumptions described in [SPARK_EXTERNAL], [SPARK_ALIASING_ADDRESS] and [SPARK_EXTERNAL_VALID].
- [ADA_PRIVATE_TYPES] Private types whose full view is not analyzed, yet are used in SPARK code, need to comply with the implicit or explicit contracts used by GNATprove to analyze references to these types. This concerns:
  - private types and private type extensions declared in a package with a `pragma SPARK_Mode (Off);` in its private part,
  - type completions in a non-SPARK package body.

  The (explicit or implicit) type contract to check is made up of:

  - *Default Initial Condition* (explicit or implicit, no runtime error shall occur during default initialization of an object of this type unless its default initial condition does not refer to the current type instance or only refers to its discriminants and it evaluates to False)
  - Ownership annotations (implicit, if a type is not annotated with Ownership, copying it around shall not create visible aliasing and if it is not annotated with Needs_Reclamation, its finalization shall not leak resources or memory).

  In addition, the default initialization of values of the type and the evaluation of its potential type invariant or subtype predicate shall not access any mutable state.

- [ADA_TAGGED_TYPES] When a tagged type `T` visible in SPARK is extended outside of SPARK code, extensions of `T` whose full view is not analyzed by GNATprove shall not break the assumptions on values of type `T'Class`. In particular, they should abide by its *Default Initial Condition*, and should not add components which require a specific handling with respect to ownership.
- [ADA_RECURSIVE_TYPES] Recursive data-structures accessed by SPARK code but created out of SPARK should not be cyclic even if they are constant (but sharing is OK).
- [ADA_ELABORATION] If a package is not analyzed but is part of the application code, its elaboration:
  - shall not modify any global state visible from SPARK unless it is part of the package's own state.
  - shall always terminate normally.

  In addition, if the package specification is referenced, directly or indirectly, from a SPARK unit, it needs to comply with the implicit or explicit contracts used by GNATprove to analyze these user packages.

  The (explicit or implicit) package contract to check is made up of:

  - `Initializes` contracts (explicit or implicitly generated by GNATprove)
  - `Initial_Condition` (only explicit)
  - the aliases constraints of SPARK (implicit - there shall not be any aliases in the global state visible from SPARK after the package elaboration)
- [ADA_SUBPROGRAMS] Subprograms that are not analyzed, yet are called from SPARK code, need to comply with the implicit or explicit contracts used by GNATprove to analyze calls to these subprograms. This concerns:
  - subprograms whose body is not given for analysis; and
  - subprograms whose body is marked `SPARK_Mode => Off`, either explicitly or implicitly (inherited from the enclosing scope).

  Note that we consider here both non-generic subprograms and instantiations of generic subprograms, never generic subprograms themselves.

  The (explicit or implicit) subprogram contract to check is made up of:

  - *Type Contracts* of parameters, result (for a function) and global objects produced as outputs from the non-SPARK callee to the SPARK caller
  - *Postconditions* (only explicit)
  - *Contract Cases* (only explicit)
  - *Data Dependencies* (explicit or implicitly generated by GNATprove)
  - *Flow Dependencies* (explicit or implicitly generated by GNATprove)
  - *Exceptional Contracts* (explicit or implicit) - the exceptional contract should list all exceptions that might be propagated by the subprogram and the associated postconditions should hold whenever an exception is propagated

  - *Subprogram Termination* (only explicit except for functions without side effects which should always return in SPARK) - subprograms annotated with `Always_Terminates` should terminate (return normally or raise an exception) whenever the associated boolean condition evaluates to True on entry of the subprogram, assuming that primary stack, secondary stack, and heap memory allocations never fail. Other subprograms are not restricted
  - the aliasing constraints of SPARK (implicit - the subprogram shall not introduce any visible aliases between its parameters, accessed global objects, and return value if any, unless it is a traversal function, in which case its return value shall be a part of its traversed parameter, or unless the aliases introduced are compatible with assumption [SPARK_ALIASING_ADDRESS])
  - parameter modes - in particular, parameters of mode *in* which are not considered to be variable should not be modified, including the values designated by their potential access-to-variable subcomponents, and parameters of mode *out* which are not subject to relaxed initialization (see *Aspect Relaxed_Initialization and Ghost Attribute Initialized*) should be entirely initialized.

  Note that this also applies to subprograms which are called indirectly from SPARK code, either through a dispatching call or through a call to an access-to-subprogram, and to (predefined) operators like "=".

- [ADA_CALLS] Calls to SPARK subprograms from subprograms that are not analyzed need to comply with the implicit or explicit preconditions used by GNATprove to analyze the called SPARK subprograms. This concerns the same subprograms as considered in [ADA_SUBPROGRAMS].

  The (explicit or implicit) precondition to check is made up of:

  - *Type Contracts* of both parameters and global objects taken as input by the SPARK callee from the non-SPARK caller
  - *Preconditions* (explicit)
  - the aliasing constraints of SPARK (implicit) - the context shall not alias the callee's parameters and accessed global objects in ways that are not allowed in SPARK
  - the initialization of inputs (implicit) - parameters of mode *in* or *in out* and global variables of mode *Input* or *In_Out* which are not subject to relaxed initialization (see *Aspect Relaxed_Initialization and Ghost Attribute Initialized*) should be entirely initialized

- [ADA_OBJECT_ADDRESSES] When the body of a function is not analyzed by GNATprove, its result should not depend on the address of parts of its parameters or global inputs unless it is annotated with `Volatile_Function`. When the body of a procedure, entry or function with side effects is not analyzed by GNATprove, none of its outputs should depend on the address of parts of its parameters or global inputs unless the output is volatile for reading, or its value depends on an input which is volatile for reading as stated in a Depends contract.

- [ADA_STATE_ABSTRACTION] Units whose body is not analyzed, yet are used from SPARK code, need to declare suitable *State Abstraction*, and subprograms defining the API of such a unit should have correct *Data Dependencies* describing how a subprogram reads or writes parts of the state abstraction. The state abstraction may represent program variables, but also states of the OS, aspects of the file system, attributes of the underlying hardware, etc.

  All entities that are part of the SPARK-compatible spec of the unit need to comply with the implicit or explicit contracts used by GNATprove to analyze use of these entities. This concerns:

  - the package itself (see *Package Contracts*); and
  - the API of the package.

- [ADA_LOGICAL_EQUAL] If the aspect or pragma `Logical_Equal` is used on a function whose implementation is not analyzed, yet called from SPARK code, the implementation of this function should correspond to the logical equality for the corresponding type as used by GNATprove. See *Annotation for Accessing the Log-*

ical Equality for a Type for information about the logical equality. Note that this assumption does not apply to functions without any implementation.

- [ADA_INLINE_FOR_PROOF] If the aspect or pragma `Inline_For_Proof` is used on a function with a post-condition whose implementation is not analyzed, yet called from SPARK code, and the function has a post-condition whose expression is syntactically a relation using the '=' relational_operator (or an expression that parenthesizes such a relation), where one side of the relation is syntactically an attribute_reference to the Result attribute of the function, then GNATprove assumes that the value of the postcondition expression is true if and only if the function return value is logically equal to an Ada copy of the value of the other side of the relation.

In addition, the following assumptions need to be addressed when calling GNATprove on only part of a SPARK program at a time (either on an individual unit or on a group of units), while providing only the specs of those units that are not analyzed (not their bodies), so that the complete SPARK program is analyzed by calling GNATprove multiple times with different sets of unit bodies being available:

- [PARTIAL_GLOBAL] Subprograms which are called across the boundary of those units analyzed together should have a Global contract describing their effect on global data, otherwise they will be assumed to have no effect on global data. The warning *assumed Global null* is guaranteed to be issued in cases where review is required.
- [PARTIAL_TERMINATION] Procedures, entries and functions with side effects which are called across the boundary of those units analyzed together should be annotated to specify under which condition they shall terminate using the `Always_Terminates` aspect. Otherwise, these subprograms will be assumed to never terminate (if they are annotated with `No_Return`) or always terminate (otherwise). The warning *assumed Always_Terminates* is guaranteed to be issued in cases where review is required.
- [PARTIAL_TASKING] If no single run of GNATprove analyzes all units that define tasks, then for each run of GNATprove, all tasks *not* defined in units analyzed during that run of GNATprove must comply with [ADA_TASKING] as if those tasks were not SPARK tasks. Note: The environment task, which is present in every Ada partition, is considered by GNATprove to be defined by the unit that defines the main subprogram of that Ada partition. Note also: If an Ada partition defines no tasks other than the environment task, then that Ada partition is trivially in compliance with this assumption.
- [PARTIAL_ACYCLIC_ANALYSIS] Consider the directed graph where the nodes are the compilation units and there is an edge from a unit A to a unit B if there is a with clause for B in the specification or the body of A. All (bodies of) units of a strongly connected component in this graph should be analyzed as part of a single analysis of GNATprove. This is so GNATprove can detect that the analyses of strongly connected units depend on each other and use its internal mechanism to avoid unsoundness.

In addition, the following assumptions need to be addressed when compiling the program with another compiler than GNAT:

- [GNAT_SPARKLIB_LEMMAS] When using lemmas from the *SPARK Lemma Library*, GNAT-specific lemmas (e.g. on fixed-point arithmetic) should be reviewed to ensure that the same semantics is used in the compiler. The name of such lemmas starts with "GNAT" and the associated comment explains how it is specific to GNAT.
- [GNAT_PEDANTIC] The switch `--pedantic` should be used as explained in section *Specifying the Target Architecture and Implementation-Defined Behavior* to warn about possible implementation-defined behavior, and the resulting warnings if any should be reviewed.
- [GNAT_PORTABILITY] The section *Ensure Portability of Programs* should be reviewed for possible differences in implementation defined behavior between GNAT/GNATprove and the chosen compiler (e.g. regarding choice of base type for scalars).

7.4 How to Write Subprogram Contracts

GNATprove relies on contracts to perform its analysis. User-specified subprogram contracts are assumed to analyze a subprogram's callers, and verified when the body of the subprogram is analyzed.

By default, no contracts are compulsory in GNATprove. In the absence of user-provided contracts, GNATprove internally generates default contracts, which may or not be suitable depending on the verification objective:

- data dependencies (`Global`)

  See *Generation of Dependency Contracts*. The generated contract may be exact when completed from user-specified flow dependencies (Depends), or precise when generated from a body in SPARK, or coarse when generated from a body in full Ada.

- flow dependencies (`Depends`)

  See *Generation of Dependency Contracts*. The contract is generated from the user-specified or generated data dependencies, by considering that all outputs depend on all inputs.

- precondition (`Pre`)

  A default precondition of `True` is used in absence of a user-specified precondition.

- postcondition (`Post`)

  A default postcondition of `True` is used in absence of a user-specified postcondition, except for expression functions. For the latter, the body of the expression function is used to generate a matching postcondition. See *Expression Functions*.

- exceptional contract (`Exceptional_Cases`)

  As a default, procedures are not considered to raise any exceptions (`Exceptional_Cases => (others => False)`).

- termination contract (`Always_Terminates`)

  The contract is generated from a body in SPARK and a default contract of `False` is used if the body is in full Ada. If there is no body, a default contract of `True` (or `False` for No_Return procedures) is used.

Knowing which contracts to write depends on the specific verification objectives to achieve.

7.4.1 Generation of Dependency Contracts

By default, GNATprove does not require the user to write data dependencies (introduced with aspect `Global`) and flow dependencies (introduced with aspect `Depends`), as it can automatically generate them from the program.

This behavior can be disabled using the `--no-global-generation` switch, which means a missing data dependency is the same as `Global => null`. Note that this option also forces `--no-inlining` (see *Contextual Analysis of Subprograms Without Contracts*).

Note: GNATprove does not generate warning or check messages when the body of a subprogram does not respect a generated contract. Indeed, the generated contract is a safe over-approximation of the real contract, hence it is unlikely that the subprogram body respects it. The generated contract is used instead to verify proper initialization and respect of dependency contracts in the callers of the subprogram.

Note: Intrinsic subprograms such as arithmetic operations, and shift/rotate functions without user-provided functional contracts (precondition, postcondition or contract cases) are handled specially by GNATprove.

Note: The `--no-global-generation` switch makes GNATprove behave more like the previous SPARK 2005 tools, which makes this switch attractive for project trying to migrate to the new GNATprove tools, or for projects that maintain dual annotations.

Auto Completion for Incomplete Contracts

When only the data dependencies (resp. only the flow dependencies) are given on a subprogram, GNATprove completes automatically the subprogram contract with the matching flow dependencies (resp. data dependencies).

Writing Only the Data Dependencies

When only the data dependencies are given on a subprogram, GNATprove completes them with flow dependencies that have all outputs depending on all inputs. This is a safe over-approximation of the real contract of the subprogram, which allows to detect all possible errors of initialization and contract violation in the subprogram and its callers, but which may also lead to false alarms because it is imprecise.

Take for example procedures `Add` and `Swap` for which data dependencies are given, but no flow dependencies:

```
1  package Only_Data_Dependencies with
2    SPARK_Mode
3  is
4     V : Integer;
5
6     procedure Add (X : Integer) with
7       Global => (In_Out => V);
8
9     procedure Swap (X : in out Integer) with
10      Global => (In_Out => V);
11
12    procedure Call_Add (X, Y : Integer) with
13      Global  => (In_Out => V),
14      Depends => (V =>+ (X, Y));
15
16    procedure Call_Swap (X, Y : in out Integer) with
17      Global  => (In_Out => V),
18      Depends => (X => Y, Y => X, V => V);
19
20 end Only_Data_Dependencies;
```

GNATprove completes the contract of `Add` and `Swap` with flow dependencies that are equivalent to:

```
procedure Add (X : Integer) with
  Global  => (In_Out => V),
  Depends => (V =>+ X);

procedure Swap (X : in out Integer) with
  Global  => (In_Out => V),
  Depends => ((X, V) => (X, V));
```

Other flow dependencies with fewer dependencies between inputs and outputs would be compatible with the given data dependencies of `Add` and `Swap`. GNATprove chooses the contracts with the most dependencies. Here, this corresponds

to the actual contract for Add, but to an imprecise contract for Swap:

```
package body Only_Data_Dependencies with
  SPARK_Mode
is
   procedure Add (X : Integer) is
   begin
      V := V + X;
   end Add;

   procedure Swap (X : in out Integer) is
      Tmp : constant Integer := V;
   begin
      V := X;
      X := Tmp;
   end Swap;

   procedure Call_Add (X, Y : Integer) is
   begin
      Add (X);
      Add (Y);
   end Call_Add;

   procedure Call_Swap (X, Y : in out Integer) is
   begin
      Swap (X);
      Swap (Y);
      Swap (X);
   end Call_Swap;

end Only_Data_Dependencies;
```

This results in false alarms when GNATprove verifies the dependency contract of procedure Call_Swap which calls Swap, while it succeeds in verifying the dependency contract of Call_Add which calls Add:

```
only_data_dependencies.adb:6:14: high: overflow check might fail, cannot prove lower␣
↪bound for V + X
    6 |      V := V + X;
      |           ~~^~~
  e.g. when V = Integer'First
        and X = -1
  reason for check: result of addition must fit in a 32-bits machine integer
  possible fix: subprogram at only_data_dependencies.ads:6 should mention X and V in a␣
↪precondition
    6 |   procedure Add (X : Integer) with
      |   ^ here

only_data_dependencies.ads:18:18: medium: missing dependency "X => V"
   18 |     Depends => (X => Y, Y => X, V => V);
      |                 ^ here

only_data_dependencies.ads:18:18: medium: missing self-dependency "X => X"
   18 |     Depends => (X => Y, Y => X, V => V);
```

(continues on next page)

(continued from previous page)

```
      |                ^ here

only_data_dependencies.ads:18:26: medium: missing dependency "Y => V"
   18 |     Depends => (X => Y, Y => X, V => V);
      |                         ^ here

only_data_dependencies.ads:18:26: medium: missing self-dependency "Y => Y"
   18 |     Depends => (X => Y, Y => X, V => V);
      |                         ^ here

only_data_dependencies.ads:18:34: medium: missing dependency "V => X"
   18 |     Depends => (X => Y, Y => X, V => V);
      |                                 ^ here

only_data_dependencies.ads:18:34: medium: missing dependency "V => Y"
   18 |     Depends => (X => Y, Y => X, V => V);
      |                                 ^ here
```

The most precise dependency contract for `Swap` would be:

```
procedure Swap (X : in out Integer) with
  Global  => (In_Out => V),
  Depends => (V => X, X => V);
```

If you add this precise contract in the program, then GNATprove can also verify the dependency contract of `Call_Swap`.

Note that the generated dependency contracts are used in the analysis of callers, but GNATprove generates no warnings or check messages if the body of `Add` or `Swap` have fewer flow dependencies, as seen above. That's a difference between these contracts being present in the code or auto completed.

Writing Only the Flow Dependencies

When only the flow dependencies are given on a subprogram, GNATprove completes it with the only compatible data dependencies.

Take for example procedures `Add` and `Swap` as previously, expect now flow dependencies are given, but no data dependencies:

```
1  package Only_Flow_Dependencies with
2    SPARK_Mode
3  is
4     V : Integer;
5
6     procedure Add (X : Integer) with
7       Depends => (V =>+ X);
8
9     procedure Swap (X : in out Integer) with
10      Depends => (V => X, X => V);
11
12    procedure Call_Add (X, Y : Integer) with
13      Global  => (In_Out => V),
14      Depends => (V =>+ (X, Y));
```

(continues on next page)

(continued from previous page)

```
15
16    procedure Call_Swap (X, Y : in out Integer) with
17      Global  => (In_Out => V),
18      Depends => (X => Y, Y => X, V => V);
19
20 end Only_Flow_Dependencies;
```

The body of the unit is the same as before:

```
1  package body Only_Flow_Dependencies with
2    SPARK_Mode
3  is
4     procedure Add (X : Integer) is
5     begin
6        V := V + X;
7     end Add;
8
9     procedure Swap (X : in out Integer) is
10       Tmp : constant Integer := V;
11    begin
12       V := X;
13       X := Tmp;
14    end Swap;
15
16    procedure Call_Add (X, Y : Integer) is
17    begin
18       Add (X);
19       Add (Y);
20    end Call_Add;
21
22    procedure Call_Swap (X, Y : in out Integer) is
23    begin
24       Swap (X);
25       Swap (Y);
26       Swap (X);
27    end Call_Swap;
28
29 end Only_Flow_Dependencies;
```

GNATprove verifies the data and flow dependencies of all subprograms, including `Call_Add` and `Call_Swap`, based on the completed contracts for `Add` and `Swap`.

Precise Generation for SPARK Subprograms

When no data or flow dependencies are given on a SPARK subprogram, GNATprove generates precise data and flow dependencies by using path-sensitive flow analysis to track data flows in the subprogram body:

- if a variable is written completely on all paths in a subprogram body, it is considered an output of the subprogram; and
- other variables that are written in a subprogram body are considered both inputs and outputs of the subprogram (even if they are not read explicitly, their output value may depend on their input value); and
- if a variable is only read in a subprogram body, it is considered an input of the subprogram; and

- all outputs are considered to potentially depend on all inputs.

Case 1: No State Abstraction

Take for example a procedure `Set_Global` without contract which initializes a global variable `V` and is called in a number of contexts:

```
package Gen_Global with
  SPARK_Mode
is
   procedure Set_Global;

   procedure Do_Nothing;

   procedure Set_Global_Twice;

end Gen_Global;
```

```
package body Gen_Global with
  SPARK_Mode
is
   V : Boolean;

   procedure Set_Global is
   begin
      V := True;
   end Set_Global;

   procedure Do_Nothing is
   begin
      null;
   end Do_Nothing;

   procedure Set_Global_Twice is
   begin
      Set_Global;
      Set_Global;
   end Set_Global_Twice;

   procedure Set_Global_Conditionally (X : Boolean) with
     Global  => (Output => V),
     Depends => (V => X)
   is
   begin
      if X then
         Set_Global;
      else
         V := False;
      end if;
   end Set_Global_Conditionally;

end Gen_Global;
```

GNATprove generates data and flow dependencies for procedure `Set_Global` that are equivalent to:

```
procedure Set_Global with
  Global  => (Output => V),
  Depends => (V => null);
```

Note that the above contract would be illegal as given, because it refers to global variable `V` which is not visible at the point where `Set_Global` is declared in `gen_global.ads`. Instead, a user who would like to write this contract on `Set_Global` would have to use abstract state.

That generated contract for `Set_Global` allows GNATprove to both detect possible errors when calling `Set_Global` and to verify contracts given by the user on callers of `Set_Global`. For example, procedure `Set_Global_Twice` calls `Set_Global` twice in a row, which makes the first call useless as the value written in `V` is immediately overwritten by the second call. This is detected by GNATprove, which issues two warnings on line 18:

```
gen_global.adb:18:07: warning: statement has no effect
   18 |      Set_Global;
      |      ^~~~~~~~~

gen_global.adb:18:07: warning: "V" is set by "Set_Global" but not used after the call
   18 |      Set_Global;
      |      ^~~~~~~~~
```

GNATprove also uses the generated contract for `Set_Global` to analyze procedure `Set_Global_Conditionally`, which allows it to verify the contract given by the user for `Set_Global_Conditionally`:

```
procedure Set_Global_Conditionally (X : Boolean) with
  Global  => (Output => V),
  Depends => (V => X)
```

Case 2: State Abstraction Without Dependencies

If an abstract state (see *State Abstraction*) is declared by the user but no dependencies are specified on subprogram declarations, then GNATprove generates data and flow dependencies which take abstract state into account.

For example, take unit `Gen_Global` previously seen, where an abstract state `State` is defined for package `Gen_Abstract_Global`, and refined into global variable `V` in the body of the package:

```
1  package Gen_Abstract_Global with
2    SPARK_Mode,
3    Abstract_State => State
4  is
5     procedure Set_Global;
6
7     procedure Set_Global_Twice;
8
9     procedure Set_Global_Conditionally (X : Boolean) with
10      Global  => (Output => State),
11      Depends => (State => X);
12
13 end Gen_Abstract_Global;
```

```
package body Gen_Abstract_Global with
  SPARK_Mode,
  Refined_State => (State => V)
is
   V : Boolean;

   procedure Set_Global is
   begin
      V := True;
   end Set_Global;

   procedure Set_Global_Twice is
   begin
      Set_Global;
      Set_Global;
   end Set_Global_Twice;

   procedure Set_Global_Conditionally (X : Boolean) with
     Refined_Global  => (Output => V),
     Refined_Depends => (V => X)
   is
   begin
      if X then
         Set_Global;
      else
         V := False;
      end if;
   end Set_Global_Conditionally;

end Gen_Abstract_Global;
```

We have chosen here to declare procedure `Set_Global_Conditionally` in `gen_abstract_global.ads`, and so to express its user contract abstractly. We could also have kept it local to the unit.

GNATprove gives the same results on this unit as before: it issues warnings for the possible error in `Set_Global_Twice` and it verifies the contract given by the user for `Set_Global_Conditionally`:

```
gen_abstract_global.adb:14:07: warning: statement has no effect
   14 |      Set_Global;
      |      ^~~~~~~~~

gen_abstract_global.adb:14:07: warning: "V" constituent of "State" is set by "Set_Global
↪" but not used after the call
   14 |      Set_Global;
      |      ^~~~~~~~~
```

Case 3: State Abstraction Without Refined Dependencies

If abstract state is declared by the user and abstract dependencies are specified on subprogram declarations, but no refined dependencies are specified on subprogram implementations (as described *State Abstraction and Dependencies*), then GNATprove generates refined data and flow dependencies for subprogram implementations.

For example, take unit `Gen_Abstract_Global` previously seen, where only abstract data and flow dependencies are specified:

```
package Gen_Refined_Global with
  SPARK_Mode,
  Abstract_State => State
is
   procedure Set_Global with
     Global => (Output => State);

   procedure Set_Global_Twice with
     Global => (Output => State);

   procedure Set_Global_Conditionally (X : Boolean) with
     Global  => (Output => State),
     Depends => (State => X);

end Gen_Refined_Global;
```

```
package body Gen_Refined_Global with
  SPARK_Mode,
  Refined_State => (State => V)
is
   V : Boolean;

   procedure Set_Global is
   begin
      V := True;
   end Set_Global;

   procedure Set_Global_Twice is
   begin
      Set_Global;
      Set_Global;
   end Set_Global_Twice;

   procedure Set_Global_Conditionally (X : Boolean) is
   begin
      if X then
         Set_Global;
      else
         Set_Global;
      end if;
   end Set_Global_Conditionally;

end Gen_Refined_Global;
```

GNATprove gives the same results on this unit as before: it issues warnings for the possible error in

`Set_Global_Twice` and it verifies the contract given by the user for `Set_Global_Conditionally`:

```
gen_refined_global.adb:14:07: warning: statement has no effect
   14 |      Set_Global;
      |      ^~~~~~~~~~

gen_refined_global.adb:14:07: warning: "V" constituent of "State" is set by "Set_Global"␣
↪but not used after the call
   14 |      Set_Global;
      |      ^~~~~~~~~~
```

Note that although abstract and refined dependencies are the same here, this is not always the case, and GNATprove will use the more precise generated dependencies to analyze calls to subprograms inside the unit.

Coarse Generation for non-SPARK Subprograms

When no data or flow dependencies are given on a non-SPARK subprogram, GNATprove generates coarser data and flow dependencies based on the reads and writes to variables in the subprogram body:

- if a variable is written in a subprogram body, it is considered both an input and an output of the subprogram; and
- if a variable is only read in a subprogram body, it is considered an input of the subprogram; and
- all outputs are considered to potentially depend on all inputs.

For example, take unit `Gen_Global` previously seen, where the body of `Set_Global` is marked with `SPARK_Mode => Off`:

```
1  package Gen_Ada_Global with
2    SPARK_Mode
3  is
4     procedure Set_Global;
5
6     procedure Set_Global_Twice;
7
8  end Gen_Ada_Global;
```

```
1  package body Gen_Ada_Global with
2    SPARK_Mode
3  is
4     V : Boolean;
5
6     procedure Set_Global with
7       SPARK_Mode => Off
8     is
9     begin
10       V := True;
11    end Set_Global;
12
13    procedure Set_Global_Twice is
14    begin
15       Set_Global;
16       Set_Global;
17    end Set_Global_Twice;
```

(continues on next page)

(continued from previous page)

```
18
19    procedure Set_Global_Conditionally (X : Boolean) with
20      Global  => (Output => V),
21      Depends => (V => X)
22    is
23    begin
24       if X then
25          Set_Global;
26       else
27          V := False;
28       end if;
29    end Set_Global_Conditionally;
30
31 end Gen_Ada_Global;
```

GNATprove generates a data and flow dependencies for procedure `Set_Global` that are equivalent to:

```
procedure Set_Global with
  Global  => (In_Out => V),
  Depends => (V => V);
```

This is a safe over-approximation of the real contract for `Set_Global`, which allows to detect all possible errors of initialization and contract violation in `Set_Global` callers, but which may also lead to false alarms because it is imprecise. Here, GNATprove generates a wrong high message that the call to `Set_Global` on line 25 reads an uninitialized value for `V`:

```
gen_ada_global.adb:25:10: high: "V" is not an input in the Global contract of subprogram
↪"Set_Global_Conditionally" at line 19
   25 |         Set_Global;
      |         ^~~~~~~~~
  either make "V" an input in the Global contract or initialize it before use
```

This is because the generated contract for `Set_Global` is not precise enough, and considers `V` as an input of the procedure. Even if the body of `Set_Global` is not in SPARK, the user can easily provide the precise information to GNATprove by adding a suitable contract to `Set_Global`, which requires to define an abstract state `State` like in the previous section:

```
procedure Set_Global with
  Global  => (Output => State),
  Depends => (State => null);
```

With such a user contract on `Set_Global`, GNATprove can verify the contract of `Set_Global_Conditionally` without false alarms.

Writing Dependency Contracts

Since GNATprove generates data and flow dependencies, you don't need in general to add such contracts if you don't want to.

The main reason to add such contracts is when you want GNATprove to verify that the implementation respects specified data dependencies and flow dependencies. For those projects submitted to certification, verification of data coupling and input/output relations may be a required verification objective, which can be achieved automatically with GNATprove provided the specifications are written as contracts.

Even if you write dependency contracts for the publicly visible subprograms, which describe the services offered by the unit, there is no need to write similar contracts on internal subprograms defined in the unit body. GNATprove can generate data and flow dependencies on these.

Also, as seen in the previous section, the data and flow dependencies generated by GNATprove may be imprecise, in which case it is necessary to add manual contracts to avoid false alarms.

7.4.2 Infeasible Subprogram Contracts

A contract is said to be *infeasible* if, for some values of its inputs satisfying the precondition of the subprogram, there exists no values of its outputs satisfying its postcondition. As an example of an infeasible (implict) contract, consider the function `Add` below. It states that it returns a valid integer value equal to `X + Y` for all valid integer values `X` and `Y`. This is not possible for some valid values of `X` and `Y`, so this contract is infeasible:

```
function Add (X, Y : Integer) return Integer is (X + Y);
```

Infeasible contracts are an issue when they occur on functions. Indeed, such functions can lead to the generation of an incorrect assumption which might invalidate the verification of every subprogram which directly or indirectly calls the function.

Generally, SPARK checks that subprograms correctly implement their implicit and explicit contracts, so infeasible contracts can only occur on subprograms which might not return normally on some inputs. As functions shall always terminate normally in SPARK, infeasible contracts on functions can never occur on a verified program entirely written in the SPARK subset. When working on code partially in another language, GNATprove also assumes that this is the case for subprograms that are not verified by the tool, see *Managing Assumptions*. Therefore, special care should be taken when *Writing Contracts on Imported Subprograms*.

To limit the impact of infeasible contracts, GNATprove inserts by default implicit guards, so that potentially incorrect postconditions are only used on input values on which the unverified subprogram is actually called. This is not a full proof system however, so users should not rely on it to avoid incorrect assumptions in the tool. What is more, these guards have a non-negligible impact on prover performance. So if in your project, all subprograms are in the SPARK subset, or you have confidence in the contracts you wrote for the subprograms which are not in SPARK, you can disable these guards using the `--function-sandboxing=off` option.

Note: The effects of an infeasible contract can sometimes be detected by enabling warnings produced by proof using the switch `--proof-warnings`. An incorrect assumption might cause branches and code snippets to be wrongly flagged as dead by this mechanism.

7.4.3 Writing Contracts for Program Integrity

The most common use of contracts is to ensure program integrity, that is, the program keeps running within safe boundaries. For example, this includes the fact that the control flow of the program cannot be circumvented (e.g. through a buffer overflow vulnerability) and that data is not corrupted (e.g. data invariants are preserved).

Preconditions can be written to ensure program integrity, and in particular they ensure:

- absence of run-time errors (AoRTE): no violations of language rules which would lead to raising an exception at run time (preconditions added to all subprograms which may raise a run-time error); and
- defensive programming: no execution of a subprogram from an unexpected state (preconditions added to subprograms in the public API, to guard against possible misuse by client units); and
- support of maintenance: prevent decrease in integrity (regressions, code rot) introduced during program evolution (preconditions added to internal subprograms, to guard against violations of the conditions to call these subprograms inside the unit itself); and
- invariant checking: ensure key data invariants are maintained throughout execution (preconditions added to all subprograms which may break the invariant).

For example, unit `Integrity` contains examples of all four kinds of preconditions:

- Precondition `X >= 0` on procedure `Seen_One` ensures AoRTE, as otherwise a negative value for `X` would cause the call to `Update` to fail a range check, as `Update` expects a non-negative value for its parameter.
- Precondition `X < Y` on procedure `Seen_Two` ensures defensive programming, as the logic of the procedure is only correctly updating global variables `Max1` and `Max2` to the two maximal values seen if parameters `X` and `Y` are given in strictly increasing order.
- Precondition `X > Max2` on procedure `Update` ensures support of maintenance, as this internal procedure relies on this condition on its parameter to operate properly.
- Precondition `Invariant` on procedure `Update` ensures invariant checking, as the property that `Max2` is less than `Max1` expressed in `Invariant` should be always respected.

```
pragma Assertion_Policy (Pre => Check);

package Integrity with
  SPARK_Mode
is
   procedure Seen_One (X : Integer) with
     Pre => X >= 0;  --  AoRTE

   procedure Seen_Two (X, Y : Natural) with
     Pre => X < Y;  --  defensive programming

end Integrity;
```

```
package body Integrity with
  SPARK_Mode
is
   Max1 : Natural := 0;  --  max value seen
   Max2 : Natural := 0;  --  second max value seen

   function Invariant return Boolean is
      (Max2 <= Max1);

```

(continues on next page)

(continued from previous page)

```
   procedure Update (X : Natural) with
     Pre => X > Max2 and then  --  support of maintenance
            Invariant          --  invariant checking
   is
   begin
      if X > Max1 then
         Max2 := Max1;
         Max1 := X;
      elsif X < Max1 then
         Max2 := X;
      end if;
   end Update;

   procedure Seen_One (X : Integer) is
   begin
      if X > Max2 then
         Update (X);
      end if;
   end Seen_One;

   procedure Seen_Two (X, Y : Natural) is
   begin
      if X > Max1 then
         Max1 := Y;
         Max2 := X;
      elsif X > Max2 then
         Update (Y);
         Seen_One (X);
      else
         Seen_One (Y);
      end if;
   end Seen_Two;

end Integrity;
```

Note that `pragma Assertion_Policy (Pre => Check)` in `integrity.ads` ensures that the preconditions on the public procedures `Seen_One` and `Seen_Two` are always enabled at run time, while the precondition on internal subprogram `Update` is only enabled at run time if compiled with switch `-gnata` (typically set only for debugging or testing). GNATprove always takes contracts into account, whatever value of `Assertion_Policy`.

GNATprove cannot verify that all preconditions on `Integrity` are respected. Namely, it cannot verify that the call to `Update` inside `Seen_One` respects its precondition, as it is not known from the calling context that `Invariant` holds:

```
integrity.adb:26:10: high: precondition might fail, cannot prove Invariant
   26 |         Update (X);
      |         ^~~~~~~~~
  e.g. when Max1 = 0
        and Max2 = 1
  possible fix: precondition of subprogram at integrity.ads:6 should mention Max1 and␣
↪Max2
    6 |   procedure Seen_One (X : Integer) with
      |   ^ here
```

Note that, although `Invariant` is not required to hold either on entry to `Seen_Two`, the tests performed in if-statements in the body of `Seen_Two` ensure that `Invariant` holds when calling `Update` inside `Seen_Two`.

To prove completely the integrity of unit `Integrity`, it is sufficient to add `Invariant` as a precondition and postcondition on every subprogram which modifies the value of global variables `Max1` and `Max2`:

```
pragma Assertion_Policy (Pre => Check);

package Integrity_Proved with
  SPARK_Mode
is
   procedure Seen_One (X : Integer) with
     Pre  => X >= 0 and then   --  AoRTE
             Invariant,        --  invariant checking
     Post => Invariant;        --  invariant checking

   procedure Seen_Two (X, Y : Natural) with
     Pre  => X < Y and then    --  defensive programming
             Invariant,        --  invariant checking
     Post => Invariant;        --  invariant checking

   function Invariant return Boolean;

end Integrity_Proved;
```

```
package body Integrity_Proved with
  SPARK_Mode
is
   Max1 : Natural := 0;  --  max value seen
   Max2 : Natural := 0;  --  second max value seen

   function Invariant return Boolean is (Max2 <= Max1);

   procedure Update (X : Natural) with
     Pre  => X > Max2 and then  --  support of maintenance
             Invariant,         --  invariant checking
     Post => Invariant          --  invariant checking
   is
   begin
      if X > Max1 then
         Max2 := Max1;
         Max1 := X;
      elsif X < Max1 then
         Max2 := X;
      end if;
   end Update;

   procedure Seen_One (X : Integer) is
   begin
      if X > Max2 then
         Update (X);
      end if;
   end Seen_One;

```

(continues on next page)

(continued from previous page)

```
30    procedure Seen_Two (X, Y : Natural) is
31    begin
32       if X > Max1 then
33          Max1 := Y;
34          Max2 := X;
35       elsif X > Max2 then
36          Update (Y);
37          Seen_One (X);
38       else
39          Seen_One (Y);
40       end if;
41    end Seen_Two;
42
43 end Integrity_Proved;
```

Here is the result of running GNATprove:

```
integrity_proved.adb:12:14: info: postcondition proved
integrity_proved.adb:26:10: info: precondition proved
integrity_proved.adb:26:18: info: range check proved
integrity_proved.adb:36:10: info: precondition proved
integrity_proved.adb:37:10: info: precondition proved
integrity_proved.adb:39:10: info: precondition proved
integrity_proved.ads:9:14: info: postcondition proved
integrity_proved.ads:14:14: info: postcondition proved
integrity_proved.ads:16:13: info: implicit aspect Always_Terminates on "Invariant" has␣
↪been proved, subprogram will terminate
```

7.4.4 Writing Contracts for Functional Correctness

Going beyond program integrity, it is possible to express functional properties of the program as subprogram contracts. Such a contract can express either partially or completely the behavior of the subprogram. Typical simple functional properties express the range/constraints for parameters on entry and exit of subprograms (encoding their *type-state*), and the state of the module/program on entry and exit of subprograms (encoding a safety or security automaton). For those projects submitted to certification, expressing a subprogram requirement or specification as a complete functional contract allows GNATprove to verify automatically the implementation against the requirement/specification.

For example, unit `Functional` is the same as `Integrity_Proved` seen previously, with additional functional contracts:

- The postcondition on procedure `Update` (expressed as a `Post` aspect) is a complete functional description of the behavior of the subprogram. Note the use of an if-expression.
- The postcondition on procedure `Seen_Two` (expressed as a `Post` aspect) is a partial functional description of the behavior of the subprogram.
- The postcondition on procedure `Seen_One` (expressed as a `Contract_Cases` aspect) is a complete functional description of the behavior of the subprogram. There are three cases which correspond to different possible behaviors depending on the values of parameter `X` and global variables `Max1` and `Max2`. The benefit of expressing the postcondition as contract cases is both the gain in readability (no need to use `'Old` for the guards, as in the postcondition of `Update`) and the automatic verification that the cases are disjoint and complete.

Note that global variables `Max1` and `Max2` are referred to through public accessor functions `Max_Value_Seen` and `Second_Max_Value_Seen`. These accessor functions can be declared after the contracts in which they appear, as

contracts are semantically analyzed only at the end of package declaration.

```
pragma Assertion_Policy (Pre => Check);

package Functional with
  SPARK_Mode
is
   procedure Seen_One (X : Integer) with
     Pre  => X >= 0 and then   --  AoRTE
             Invariant,        --  invariant checking
     Post => Invariant,        --  invariant checking
     Contract_Cases =>         --  full functional
       (X > Max_Value_Seen =>
          --  max value updated
          Max_Value_Seen = X and
          Second_Max_Value_Seen = Max_Value_Seen'Old,
        X > Second_Max_Value_Seen and
        X < Max_Value_Seen =>
          --  second max value updated
          Max_Value_Seen = Max_Value_Seen'Old and
          Second_Max_Value_Seen = X,
        X = Max_Value_Seen or
        X <= Second_Max_Value_Seen =>
          --  no value updated
          Max_Value_Seen = Max_Value_Seen'Old and
          Second_Max_Value_Seen = Second_Max_Value_Seen'Old);

   procedure Seen_Two (X, Y : Natural) with
     Pre  => X < Y and then               --  defensive programming
             Invariant,                   --  invariant checking
     Post => Invariant and then           --  invariant checking
             Max_Value_Seen > 0 and then  --  partial functional
             Max_Value_Seen /= Second_Max_Value_Seen;

   function Invariant return Boolean;

   function Max_Value_Seen return Integer;

   function Second_Max_Value_Seen return Integer;

end Functional;
```

```
package body Functional with
  SPARK_Mode
is
   Max1 : Natural := 0;  --  max value seen
   Max2 : Natural := 0;  --  second max value seen

   function Invariant return Boolean is (Max2 <= Max1);

   function Max_Value_Seen return Integer is (Max1);

   function Second_Max_Value_Seen return Integer is (Max2);
```

(continues on next page)

(continued from previous page)

```
12
13    procedure Update (X : Natural) with
14      Pre  => X > Max2 and then      --  support of maintenance
15              Invariant,             --  invariant checking
16      Post => Invariant and then     --  invariant checking
17              (if X > Max1'Old then  --  complete functional
18                 Max2 = Max1'Old and Max1 = X
19               elsif X < Max1'Old then
20                 Max2 = X and Max1 = Max1'Old
21               else
22                 Max2 = Max2'Old and Max1 = Max1'Old)
23    is
24    begin
25       if X > Max1 then
26          Max2 := Max1;
27          Max1 := X;
28       elsif X < Max1 then
29          Max2 := X;
30       end if;
31    end Update;
32
33    procedure Seen_One (X : Integer) is
34    begin
35       if X > Max2 then
36          Update (X);
37       end if;
38    end Seen_One;
39
40    procedure Seen_Two (X, Y : Natural) is
41    begin
42       if X > Max1 then
43          Max1 := Y;
44          Max2 := X;
45       elsif X > Max2 then
46          Update (Y);
47          Seen_One (X);
48       else
49          Seen_One (Y);
50       end if;
51    end Seen_Two;
52
53 end Functional;
```

GNATprove manages to prove automatically almost all of these functional contracts, except for the postcondition of `Seen_Two` (note in particular the proof that the contract cases for `Seen_One` on line 10 are disjoint and complete):

```
functional.ads:31:14: medium: postcondition might fail, cannot prove Max_Value_Seen /=␣
↪Second_Max_Value_Seen
   31 |             Max_Value_Seen /= Second_Max_Value_Seen;
      |             ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  e.g. when Max1 = 2
```

(continues on next page)

(continued from previous page)

```
        and Max2 = 2
```

The counterexample displayed for the postcondition not proved corresponds to a case where `Max1 = Max2 = 2` on entry to procedure `Seen_Two`. By highlighting the path for the counterexample in GNAT Studio (see *Running GNATprove from GNAT Studio*), the values of parameters for this counterexample are also displayed, here `X = 0` and `Y = 1`. With these values, `Max1` and `Max2` would still be equal to 2 on exit, thus violating the part of the postcondition stating that `Max_Value_Seen /= Second_Max_Value_Seen`.

Another way to see it is to run GNATprove in mode `per_path` (see *Running GNATprove from the Command Line* or *Running GNATprove from GNAT Studio*), and highlight the path on which the postcondition is not proved, which shows that when the last branch of the if-statement is taken, the following property is not proved:

```
functional.ads:31:14: medium: postcondition might fail, cannot prove Max_Value_Seen /=␣
↪(Second_Max_Value_Seen)
```

The missing piece of information here is that `Max1` and `Max2` are never equal, except when they are both zero (the initial value). This can be added to function `Invariant` as follows:

```
   function Invariant return Boolean is
     (if Max1 = 0 then Max2 = 0 else Max2 < Max1);
```

With this more precise definition for `Invariant`, all contracts are now proved by GNATprove:

```
functional_proved.adb:17:14: info: postcondition proved
functional_proved.adb:37:10: info: precondition proved
functional_proved.adb:37:18: info: range check proved
functional_proved.adb:47:10: info: precondition proved
functional_proved.adb:48:10: info: precondition proved
functional_proved.adb:50:10: info: precondition proved
functional_proved.ads:9:14: info: postcondition proved
functional_proved.ads:10:06: info: disjoint contract cases proved
functional_proved.ads:10:06: info: complete contract cases proved
functional_proved.ads:11:28: info: contract case proved
functional_proved.ads:16:28: info: contract case proved
functional_proved.ads:21:36: info: contract case proved
functional_proved.ads:29:14: info: postcondition proved
functional_proved.ads:33:13: info: implicit aspect Always_Terminates on "Invariant" has␣
↪been proved, subprogram will terminate
functional_proved.ads:35:13: info: implicit aspect Always_Terminates on "Max_Value_Seen"␣
↪has been proved, subprogram will terminate
functional_proved.ads:37:13: info: implicit aspect Always_Terminates on "Second_Max_
↪Value_Seen" has been proved, subprogram will terminate
```

In general, it may be needed to further refine the preconditions of subprograms to be able to prove their functional postconditions, to express either specific constraints on their calling context, or invariants maintained throughout the execution.

7.4.5 Writing Contracts on Main Subprograms

Parameterless procedures and parameterless functions with Integer return type, that are in their own compilation unit, are identified by GNATprove as potential main subprograms. These subprograms are special because they can serve as an entry point to the program. If a main subprogram has a precondition, SPARK will generate a check that this precondition holds at the beginning of the execution of the main subprogram, assuming the `Initial_Condition` aspects of all with'ed packages.

Note that apart from this additional check, main subprograms behave like any other subprogram. They can be called from anywhere, and their preconditions need to be checked when they are called.

7.4.6 Writing Contracts on Imported Subprograms

Contracts are particularly useful to specify the behavior of imported subprograms, which cannot be analyzed by GNATprove. It is compulsory to specify in data dependencies the global variables these imported subprograms may read and/or write, otherwise GNATprove assumes `null` data dependencies (no global variable read or written). It is also compulsory to specify procedures which may not terminate with aspect `Always_Terminates` (see *Contracts for Termination*), otherwise GNATprove assumes that imported subprograms always terminate. Note that a function is in general expected to terminate in SPARK, so functions that do otherwise should be replaced by procedures with a suitable annotation.

Note: A subprogram whose implementation is not available to GNATprove, either because the corresponding unit body has not been developed yet, or because the unit body is not part of the files analyzed by GNATprove (see *Specifying Files To Analyze* and *Excluding Files From Analysis*), is treated by GNATprove like an imported subprogram. The latter includes in particular subprograms from library projects that are externally built.

Note: Intrinsic subprograms such as arithmetic operations and shift/rotate functions are handled specially by GNATprove. Except for shift/rotate operations with a user-provided functional contract (precondition, postcondition or contract cases) which are treated like regular functions.

For example, unit `Gen_Imported_Global` is a modified version of the `Gen_Abstract_Global` unit seen previously in *Generation of Dependency Contracts*, where procedure `Set_Global` is imported from C:

```
1  package Gen_Imported_Global with
2    SPARK_Mode,
3    Abstract_State => (State with External =>
4                         (Async_Writers,
5                          Async_Readers => False,
6                          Effective_Reads => False,
7                          Effective_Writes => False))
8  is
9     procedure Set_Global with
10      Import,
11      Convention => C,
12      Global => (Output => State),
13      Always_Terminates;
14
15    procedure Set_Global_Twice;
16
17    procedure Set_Global_Conditionally (X : Boolean) with
```

(continues on next page)

(continued from previous page)

```
     Global  => (Output => State),
     Depends => (State => X);

end Gen_Imported_Global;
```

Note that we added data dependencies to procedure `Set_Global`, which can be used to analyze its callers. We did not add flow dependencies, as they are the same as the auto completed ones (see *Auto Completion for Incomplete Contracts*).

```
with System.Storage_Elements;

package body Gen_Imported_Global with
  SPARK_Mode,
  Refined_State => (State => V)
is
   pragma Warnings (GNATprove, Off, "assuming correct volatile properties");
   pragma Warnings (GNATprove, Off, "assuming no concurrent accesses");
   V : Integer with
     Size => 32,
     Volatile,
     Async_Writers => True,
     Async_Readers => False,
     Effective_Reads => False,
     Effective_Writes => False,
     Address => System.Storage_Elements.To_Address (16#8000_0000#);
   pragma Warnings (GNATprove, On, "assuming correct volatile properties");
   pragma Warnings (GNATprove, On, "assuming no concurrent accesses");

   procedure Set_Global_Twice is
   begin
      Set_Global;
      Set_Global;
   end Set_Global_Twice;

   procedure Set_Global_Conditionally (X : Boolean) with
     Refined_Global  => (Output => V),
     Refined_Depends => (V => X)
   is
   begin
      if X then
         Set_Global;
      else
         V := 42;
      end if;
   end Set_Global_Conditionally;

end Gen_Imported_Global;
```

Note that we added an `Address` aspect to global variable `V`, so that it can be read/written from a C file; this also requires the variable to be volatile, which in turn requires the abstract state to be marked as external.

GNATprove gives the same results on this unit as before: it issues warnings for the possible error in `Set_Global_Twice` and it verifies the contract given by the user for `Set_Global_Conditionally`:

```
gen_imported_global.adb:22:07: warning: statement has no effect
   22 |      Set_Global;
      |      ^~~~~~~~~

gen_imported_global.adb:22:07: warning: "V" constituent of "State" is set by "Set_Global
↪" but not used after the call
   22 |      Set_Global;
      |      ^~~~~~~~~
```

It is also possible to add functional contracts on imported subprograms, which GNATprove uses to prove properties of their callers. It is compulsory to specify in a precondition the conditions for calling these imported subprograms without errors, otherwise GNATprove assumes a default precondition of `True` (no constraints on the calling context). One benefit of these contracts is that they are verified at run time when the corresponding assertion is enabled in Ada (either with pragma `Assertion_Policy` or compilation switch `-gnata`).

Note: A subprogram whose implementation is not in SPARK is treated by GNATprove almost like an imported subprogram, except that coarse data and flow dependencies are generated (see *Coarse Generation for non-SPARK Subprograms*). In particular, unless the user adds a precondition to such a subprogram, GNATprove assumes a default precondition of `True`.

For example, unit `Functional_Imported` is a modified version of the `Functional_Proved` unit seen previously in *Writing Contracts for Functional Correctness*, where procedures `Update` and `Seen_One` are imported from C:

```
1  pragma Assertion_Policy (Pre => Check);
2
3  package Functional_Imported with
4    SPARK_Mode,
5    Abstract_State => Max_And_Snd,
6    Initializes => Max_And_Snd
7  is
8     procedure Seen_One (X : Integer) with
9       Import,
10      Convention => C,
11      Global => (In_Out => Max_And_Snd),
12      Always_Terminates,
13      Pre  => X >= 0 and then   --  AoRTE
14              Invariant,        --  invariant checking
15      Post => Invariant,        --  invariant checking
16      Contract_Cases =>         --  full functional
17        (X > Max_Value_Seen =>
18           --  max value updated
19           Max_Value_Seen = X and
20           Second_Max_Value_Seen = Max_Value_Seen'Old,
21         X > Second_Max_Value_Seen and
22         X < Max_Value_Seen =>
23           --  second max value updated
24           Max_Value_Seen = Max_Value_Seen'Old and
25           Second_Max_Value_Seen = X,
26         X = Max_Value_Seen or
27         X <= Second_Max_Value_Seen =>
28           --  no value updated
```

(continues on next page)

(continued from previous page)

```
         Max_Value_Seen = Max_Value_Seen'Old and
         Second_Max_Value_Seen = Second_Max_Value_Seen'Old);

   procedure Seen_Two (X, Y : Natural) with
     Pre  => X < Y and then               --  defensive programming
             Invariant,                   --  invariant checking
     Post => Invariant and then           --  invariant checking
             Max_Value_Seen > 0 and then  --  partial functional
             Max_Value_Seen /= Second_Max_Value_Seen;

   function Invariant return Boolean;

   function Max_Value_Seen return Integer;

   function Second_Max_Value_Seen return Integer;

end Functional_Imported;
```

```
with System.Storage_Elements;

package body Functional_Imported with
  SPARK_Mode,
  Refined_State => (Max_And_Snd => (Max, Snd))
is
   Max : Natural := 0  --  max value seen
     with Address  => System.Storage_Elements.To_Address (16#8000_0000#),
          Warnings => Off;

   Snd : Natural := 0  --  second max value seen
     with Address  => System.Storage_Elements.To_Address (16#8000_0004#),
          Warnings => Off;

   function Invariant return Boolean is
     (if Max = 0 then Snd = 0 else Snd < Max);

   function Max_Value_Seen return Integer is (Max);

   function Second_Max_Value_Seen return Integer is (Snd);

   procedure Update (X : Natural) with
     Import,
     Convention => C,
     Global => (In_Out => (Max, Snd)),
     Always_Terminates,
     Pre  => X > Snd and then       --  support of maintenance
             Invariant,             --  invariant checking
     Post => Invariant and then     --  invariant checking
             (if X > Max'Old then   --  complete functional
                Snd = Max'Old and Max = X
              elsif X < Max'Old then
                Snd = X and Max = Max'Old
              else
```

(continues on next page)

(continued from previous page)

```
                Snd = Snd'Old and Max = Max'Old);

   procedure Seen_Two (X, Y : Natural) is
   begin
      if X > Max then
         Max := Y;
         Snd := X;
      elsif X > Snd then
         Update (Y);
         Seen_One (X);
      else
         Seen_One (Y);
      end if;
   end Seen_Two;

end Functional_Imported;
```

Note that we added data dependencies to the imported procedures, as GNATprove would assume otherwise incorrectly `null` data dependencies.

As before, all contracts are proved by GNATprove:

```
functional_imported.adb:43:10: info: precondition proved
functional_imported.adb:44:10: info: precondition proved
functional_imported.adb:46:10: info: precondition proved
functional_imported.ads:6:03: info: flow dependencies proved
functional_imported.ads:16:06: info: disjoint contract cases proved
functional_imported.ads:16:06: info: complete contract cases proved
functional_imported.ads:35:14: info: postcondition proved
functional_imported.ads:39:13: info: implicit aspect Always_Terminates on "Invariant"␣
↪has been proved, subprogram will terminate
functional_imported.ads:41:13: info: implicit aspect Always_Terminates on "Max_Value_Seen
↪" has been proved, subprogram will terminate
functional_imported.ads:43:13: info: implicit aspect Always_Terminates on "Second_Max_
↪Value_Seen" has been proved, subprogram will terminate
```

7.4.7 Contextual Analysis of Subprograms Without Contracts

It may be convenient to create local subprograms without necessarily specifying a contract for these. GNATprove attempts to perform a contextual analysis of these local subprograms without contract, at each call site, as if the code of the subprograms was inlined. Thus, the analysis proceeds in that case as if it had the most precise contract for the local subprogram, in the context of its calls.

Let's consider as previously a subprogram which adds two to its integer input:

```
package Arith_With_Local_Subp
  with SPARK_Mode
is
   procedure Add_Two (X : in out Integer) with
     Pre  => X <= Integer'Last - 2,
     Post => X = X'Old + 2;
```

(continues on next page)

(continued from previous page)

```
end Arith_With_Local_Subp;
```

And let's implement it by calling two local subprograms without contracts (which may or not have a separate declaration), which each increment the input by one:

```
package body Arith_With_Local_Subp
  with SPARK_Mode
is
   --  Local procedure without external visibility
   procedure Increment_In_Body (X : in out Integer) is
   begin
      X := X + 1;
   end Increment_In_Body;

   procedure Add_Two (X : in out Integer) is

      --  Local procedure defined inside Add_Two
      procedure Increment_Nested (X : in out Integer) is
      begin
         X := X + 1;
      end Increment_Nested;

   begin
      Increment_In_Body (X);
      Increment_Nested (X);
   end Add_Two;

end Arith_With_Local_Subp;
```

GNATprove would not be able to prove that the addition in `Increment_In_Body` or `Increment_Nested` cannot overflow in any context. If it was using only the default contract for these subprograms, it also would not prove that the contract of `Add_Two` is respected. But since it analyzes these subprograms in the context of their calls only, it proves here that no overflow is possible, and that the two increments correctly implement the contract of `Add_Two`:

```
arith_with_local_subp.adb:7:14: info: overflow check proved, in call inlined at arith_
↪with_local_subp.adb:19
arith_with_local_subp.adb:15:17: info: overflow check proved, in call inlined at arith_
↪with_local_subp.adb:20
arith_with_local_subp.ads:6:14: info: postcondition proved
arith_with_local_subp.ads:6:24: info: overflow check proved
```

This contextual analysis is available only for regular functions (not expression functions) or procedures that are not externally visible (not declared in the public part of the unit), without contracts (any of Global, Depends, Pre, Post, Contract_Cases), and respect the following conditions:

- not dispatching
- not marked `No_Return`
- not a generic instance
- not defined in a generic instance
- not defined in a protected type

- without a parameter of unconstrained record type with discriminant dependent components
- without a parameter or result of deep type (access type or composite type containing an access type)
- not a traversal function
- without an annotation to skip part of the analysis (see *Annotation for Skipping Parts of the Analysis for an Entity*)
- without an annotation to hide or unhide information on another entity (see *Annotation for Hiding Information*)

Subprograms that respects all of the above conditions are candidates for contextual analysis, and calls to such subprograms are inlined provided the subprogram and its calls respect the following additional conditions:

- does not contain nested subprogram or package declarations or instantiations
- not recursive
- has a single point of return at the end of the subprogram
- not called in an assertion or a contract
- not called in a potentially unevaluated context
- not called before its body is seen

If any of the above conditions is violated, GNATprove issues an info message to explain why the subprogram could not be analyzed in the context of its calls, and then proceeds to analyze it normally, using the default contract. Otherwise, both flow analysis and proof are done for the subprogram in the context of its calls.

Note that it is very simple to prevent contextual analysis of a local subprogram, by adding a contract to it, for example a simple `Pre => True` or `Global => null`. To prevent contextual analysis of all subprograms, pass the switch `--no-inlining` to GNATprove. This may be convenient during development if the ultimate goal is to add contracts to subprograms to analyze them separately, as contextual analysis may cause the analysis to take much more time and memory.

7.4.8 Subprogram Termination

GNATprove can be used to verify the termination of subprograms. It will do it unconditionnally for regular functions and package elaboration (which shall have no side effects in SPARK), and on demand for procedures, entries and functions with side effects (see *Aspect Side_Effects*). In the following example, we specify that the five procedures should terminate using the `Always_Terminates` aspect (see *Contracts for Termination*):

```
1  package Terminating_Annotations with SPARK_Mode is
2
3     procedure P_Rec (X : Natural) with
4       Always_Terminates;
5
6     procedure P_While (X : Natural) with
7       Always_Terminates;
8
9     procedure P_Not_SPARK (X : Natural) with
10      Always_Terminates;
11
12    procedure Not_SPARK (X : Natural);
13    procedure P_Call (X : Natural) with
14      Always_Terminates;
15
16    procedure P_Term (X : Natural) with
17      Always_Terminates,
```

(continues on next page)

(continued from previous page)

```
     Subprogram_Variant => (Decreases => X);
end Terminating_Annotations;
```

To verify these annotations, GNATprove will look for while loops with no loop variants, recursive calls, and calls to procedures or entries which are not known to terminate. If it cannot make sure that the annotated subprogram will always terminate, it will then emit a failed check. As an example, let us consider the following implementation of the five procedures:

```
package body Terminating_Annotations with SPARK_Mode is

   procedure P_Rec (X : Natural) is
   begin
      if X = 0 then
         return;
      else
         P_Rec (X - 1);
      end if;
   end P_Rec;

   procedure P_While (X : Natural) is
      Y : Natural := X;
   begin
      while Y > 0 loop
         Y := Y - 1;
      end loop;
   end P_While;

   procedure P_Not_SPARK (X : Natural) with SPARK_Mode => Off is
      Y : Natural := X;
   begin
      while Y > 0 loop
         Y := Y - 1;
      end loop;
   end P_Not_SPARK;

   procedure Not_SPARK (X : Natural) with SPARK_Mode => Off is
   begin
      null;
   end Not_SPARK;

   procedure P_Call (X : Natural) is
   begin
      Not_SPARK (X);
   end P_Call;

   procedure P_Term (X : Natural) is
      Y : Natural := X;
   begin
      P_Rec (Y);
      P_While (Y);
      P_Not_SPARK (Y);
      P_Call (Y);
```

(continues on next page)

(continued from previous page)

```
45
46       while Y > 0 loop
47          pragma Loop_Variant (Decreases => Y);
48          Y := Y - 1;
49       end loop;
50
51       if X = 0 then
52          return;
53       else
54          P_Term (X - 1);
55       end if;
56    end P_Term;
57 end Terminating_Annotations;
```

As can be easily verified by review, all these procedures terminate. However, GNATprove will fail to verify that `P_Rec`, `P_While`, and `P_Call` always terminate:

```
1
2  terminating_annotations.adb:8:10: medium: aspect Always_Terminates on "P_Rec" could be␣
   ↪incorrect, subprogram is recursive
3      8 |         P_Rec (X - 1);
4        |         ^~~~~~~~~~~~
5    possible fix: annotate "P_Rec" with a Subprogram_Variant aspect
6
7  terminating_annotations.adb:15:19: medium: aspect Always_Terminates on "P_While" could␣
   ↪be incorrect, loop might be nonterminating
8     15 |      while Y > 0 loop
9        |                  ^~~~
10   possible fix: add loop variant in the loop body
11
12 terminating_annotations.adb:35:07: medium: aspect Always_Terminates on "P_Call" could be␣
   ↪incorrect, call to "Not_SPARK" might be nonterminating
13    35 |      Not_SPARK (X);
14       |      ^~~~~~~~~~~~
15   possible fix: annotate "Not_SPARK" with aspect Always_Terminates
```

Let us look at each procedure to understand what happens. The procedure `P_Rec` is recursive, and `P_While` contains a while loop. Both cases can theoretically lead to an infinite path in the subprogram, which is why GNATprove cannot verify that they terminate. GNATprove does not complain about not being able to verify the termination of `P_Not_SPARK`. Clearly, it is not because it could verify it, as it contains exactly the same loop as `P_While`. It is because, as the body of `P_Not_SPARK` has been excluded from analysis using `SPARK_Mode => Off`, GNATprove does not attempt to prove that it terminates. When looking at the body of `P_Call`, we can see that it calls a procedure `Not_SPARK`. Clearly, this procedure always returns, as it does not do anything. But, as the body of `No_SPARK` has been hidden from analysis using `SPARK_Mode => Off`, GNATprove cannot deduce that it terminates. As a result, it stays in the safe side, and assumes that `Not_SPARK` could loop, which causes the verification of `P_Call` to fail. Finally, GNATprove is able to verify that `P_Term` terminates, though it contains both a while loop and a recursive call. Indeed, we have bounded both the number of possible iterations of the loop and the number of recursive calls using a `Loop_Variant` (for the loop iterations) and a `Subprogram_Variant` (for the recursive calls). Also note that, though it was not able to prove termination of `P_Rec`, `P_While`, and `P_Call`, GNATprove will still trust the annotation when verifying `P_Term`.

7.5 How to Write Object Oriented Contracts

Object Oriented Programming (OOP) may require the use of special *Class-Wide Subprogram Contracts* for dispatching subprograms, so that GNATprove can check Liskov Substitution Principle on every overriding subprogram.

7.5.1 Object Oriented Code Without Dispatching

In the special case where OOP is used without dispatching, it is possible to use the regular *Subprogram Contracts* instead of the special *Class-Wide Subprogram Contracts*.

For example, consider a variant of the `Logging` and `Range_Logging` units presented in *Class-Wide Subprogram Contracts*, where no dispatching is allowed. Then, it is possible to use regular preconditions and postconditions as contracts, provided `Log_Type` is publicly declared as an untagged private type in both units:

```
package Logging_No_Dispatch with
  SPARK_Mode
is
   Max_Count : constant := 10_000;

   type Log_Count is range 0 .. Max_Count;

   type Log_Type is private;

   function Log_Size (Log : Log_Type) return Log_Count;

   procedure Init_Log (Log : out Log_Type) with
     Post => Log_Size (Log) = 0;

   procedure Append_To_Log (Log : in out Log_Type; Incr : in Integer) with
     Pre  => Log_Size (Log) < Max_Count,
     Post => Log_Size (Log) = Log_Size (Log)'Old + 1;

private

   subtype Log_Index is Log_Count range 1 .. Max_Count;
   type Integer_Array is array (Log_Index) of Integer;

   type Log_Type is tagged record
      Log_Data : Integer_Array;
      Log_Size : Log_Count;
   end record;

   function Log_Size (Log : Log_Type) return Log_Count is (Log.Log_Size);

end Logging_No_Dispatch;
```

```
with Logging_No_Dispatch; use Logging_No_Dispatch;

package Range_Logging_No_Dispatch with
  SPARK_Mode
is
   type Log_Type is private;
```

(continues on next page)

(continued from previous page)

```
7
8     function Log_Size (Log : Log_Type) return Log_Count;
9
10    function Log_Min (Log : Log_Type) return Integer;
11
12    function Log_Max (Log : Log_Type) return Integer;
13
14    procedure Init_Log (Log : out Log_Type) with
15      Post => Log_Size (Log) = 0 and
16              Log_Min (Log) = Integer'Last and
17              Log_Max (Log) = Integer'First;
18
19    procedure Append_To_Log (Log : in out Log_Type; Incr : in Integer) with
20      Pre  => Log_Size (Log) < Logging_No_Dispatch.Max_Count,
21      Post => Log_Size (Log) = Log_Size (Log)'Old + 1 and
22              Log_Min (Log) = Integer'Min (Log_Min (Log)'Old, Incr) and
23              Log_Max (Log) = Integer'Max (Log_Max (Log)'Old, Incr);
24
25  private
26
27    type Log_Type is tagged record
28      Log : Logging_No_Dispatch.Log_Type;
29      Min_Entry : Integer;
30      Max_Entry : Integer;
31    end record;
32
33    function Log_Size (Log : Log_Type) return Log_Count is (Log_Size (Log.Log));
34
35    function Log_Min (Log : Log_Type) return Integer is (Log.Min_Entry);
36    function Log_Max (Log : Log_Type) return Integer is (Log.Max_Entry);
37
38  end Range_Logging_No_Dispatch;
```

7.5.2 Writing Contracts on Dispatching Subprograms

Whenever dispatching is used, the contract that applies in proof to a dispatching call is the class-wide contract, defined as the first one present in the following list:

1. the class-wide precondition (resp. postcondition) attached to the subprogram
2. or otherwise the class-wide precondition (resp. postcondition) being inherited by the subprogram from the subprogram it overrides
3. or otherwise the default class-wide precondition (resp. postcondition) of `True`.

For abstract subprograms (on interfaces or regular tagged types), only a class-wide contract can be specified. For other dispatching subprograms, it is possible to specify both a regular contract and a class-wide contract. In such a case, GNATprove uses the regular contract to analyze static calls to the subprogram and the class-wide contract to analyze dispatching calls to the subprogram, and it checks that the specific contract is a refinement of the class-wide contract, as explained in *Mixing Class-Wide and Specific Subprogram Contracts*.

Let's consider the various cases that may occur when overriding a subprogram:

```
package Geometry with
  SPARK_Mode
is
   type Shape is tagged record
      Pos_X, Pos_Y : Float;
   end record;

   function Valid (S : Shape) return Boolean is
     (S.Pos_X in -100.0 .. 100.0 and S.Pos_Y in -100.0 .. 100.0);

   procedure Operate (S : in out Shape) with
     Pre'Class => Valid (S);

   procedure Set_Default (S : in out Shape) with
     Post'Class => Valid (S);

   procedure Set_Default_Repeat (S : in out Shape) with
     Post'Class => Valid (S);

   procedure Set_Default_No_Post (S : in out Shape);

   type Rectangle is new Shape with record
      Len_X, Len_Y : Float;
   end record;

   function Valid (S : Rectangle) return Boolean is
     (Valid (Shape (S)) and S.Len_X in 0.0 .. 10.0 and S.Len_Y in 0.0 .. 10.0);

   procedure Operate (S : in out Rectangle);

   procedure Set_Default (S : in out Rectangle);

   procedure Set_Default_Repeat (S : in out Rectangle) with
     Post'Class => Valid (S);

   procedure Set_Default_No_Post (S : in out Rectangle) with
     Post'Class => Valid (S);

end Geometry;
```

In package `Geometry`, a type `Shape` is derived in a type `Rectangle`. A function `Shape.Valid` defines what it is to be a valid shape. It is overridden by `Rectangle.Valid` which defines what it is to be a valid rectangle. Here, a valid rectangle is also a valid shape, but that need not be the case. Procedure `Set_Default` and its variants demonstrate the various configurations that can be found in practice:

1. The overridden subprogram `Shape.Set_Default` defines a class-wide contract (here only a postcondition), which is inherited in the overriding subprogram `Rectangle.Set_Default`. By the semantics of Ada, the postcondition of `Shape.Set_Default` calls `Shape.Valid`, while the inherited postcondition of `Rectangle.Set_Default` calls `Rectangle.Valid`.

2. Both the overridden subprogram `Shape.Set_Default_Repeat` and the overriding subprogram `Rectangle.Set_Default_Repeat` define a class-wide contract (here only a postcondition). Here, since the contract is simply repeated, this is equivalent to case 1 above of inheriting the contract: the postcondition of `Shape.Set_Default_Repeat` calls `Shape.Valid`, while the postcondition of `Rectangle.Set_Default_Repeat`

calls `Rectangle.Valid`.

3. Only the overriding subprogram `Rectangle.Set_Default_No_Post` defines a class-wide contract (here only a postcondition). The default class-wide postcondition of `True` is used for the overridden `Shape.Set_Default_No_Post`.

In case 1, the overriding subprogram satisfies Liskov Substitution Principle by construction, so GNATprove emits no check in that case. Note that this is not the same as saying that `Shape.Set_Default` and `Rectangle.Set_Default` have the same contract: here the two postconditions differ, as one calls `Shape.Valid`, while the other calls `Rectangle.Valid`.

In case 2, GNATprove checks that Liskov Substitution Principle is verified between the contracts of the overridden and the overriding subprograms. Here, it checks that the postcondition of `Rectangle.Set_Default_Repeat` is stronger than the postcondition of `Shape.Set_Default_Repeat`.

In case 3, GNATprove also checks that Liskov Substitution Principle is verified between the default contract of the overridden subprogram and the specified contract of the overriding subprograms. Here, only a postcondition is specified for `Rectangle.Set_Default_No_Post`, so it is indeed stronger than the default postcondition of `Shape.Set_Default_No_Post`.

Hence the results of GNATprove's analysis on this program:

```
geometry.ads:8:13: info: implicit aspect Always_Terminates on "Valid" has been proved,␣
↪subprogram will terminate
geometry.ads:26:13: info: implicit aspect Always_Terminates on "Valid" has been proved,␣
↪subprogram will terminate
geometry.ads:34:20: info: class-wide postcondition is stronger than overridden one
geometry.ads:37:20: info: class-wide postcondition is stronger than overridden one
```

Let's consider now calls to these subprograms in procedure `Use_Geometry`:

```
1  with Geometry; use Geometry;
2
3  procedure Use_Geometry (S : in out Shape'Class) with
4    SPARK_Mode
5  is
6  begin
7     S.Set_Default;
8     S.Operate;
9
10    S.Set_Default_Repeat;
11    S.Operate;
12
13    S.Set_Default_No_Post;
14    S.Operate;
15 end Use_Geometry;
```

Here are the results of GNATprove's analysis on this program:

```
geometry.ads:9:13: info: implicit aspect Always_Terminates on "Valid" has been proved,␣
↪subprogram will terminate
geometry.ads:31:13: info: implicit aspect Always_Terminates on "Valid" has been proved,␣
↪subprogram will terminate
geometry.ads:44:20: info: class-wide postcondition is stronger than overridden one
geometry.ads:49:20: info: class-wide postcondition is stronger than overridden one
use_geometry.adb:8:05: info: precondition proved
```

(continues on next page)

(continued from previous page)

```
use_geometry.adb:11:05: info: precondition proved

use_geometry.adb:14:05: medium: precondition might fail
   14 |   S.Operate;
      |   ~^~~~~~~
  possible fix: call at line 13 should mention S (for argument S) in a postcondition
   13 |   S.Set_Default_No_Post;
      |    ^ here
```

Parameter `S` is of class-wide type `Shape'Class`, so it can be dynamically of both types `Shape` or `Rectangle`. All calls on `S` are dispatching. In this program, GNATprove needs to check that the precondition of the calls to `Operate` is satisfied. As procedures `Shape.Set_Default` and `Shape.Set_Default_Repeat` state precisely this condition in postcondition, the precondition to the calls to `Operate` that follow can be proved. As procedure `Shape.Set_Default_No_Post` has no postcondition, the precondition to the last call to `Operate` cannot be proved. Note that these proofs take into account both possible types of `S`, for example:

- If `S` is dynamically a shape, then the call to `Shape.Set_Default` on line 7 ensures that `Shape.Valid` holds, which ensures that the precondition to the call to `Shape.Operate` is satisfied on line 8.
- If `S` is dynamically a rectangle, then the call to `Rectangle.Set_Default` on line 7 ensures that `Rectangle.Valid` holds, which ensures that the precondition to the call to `Rectangle.Operate` is satisfied on line 8.

7.5.3 Writing Contracts on Subprograms with Class-wide Parameters

Subprograms with class-wide parameters are not in general dispatching subprograms, hence they are specified through regular *Subprogram Contracts*, not *Class-Wide Subprogram Contracts*. Inside the regular contract, calls on primitive subprograms of the class-wide parameters are dispatching though, like in the code. For example, consider procedure `More_Use_Geometry` which takes four class-wide parameters of type `Shape'Class`, which can all be dynamically of both types `Shape` or `Rectangle`:

```
1  with Geometry; use Geometry;
2
3  procedure More_Use_Geometry (S1, S2, S3, S4 : in out Shape'Class) with
4    SPARK_Mode,
5    Pre => S1.Valid
6  is
7  begin
8     S1.Operate;
9
10    if S2.Valid then
11       S2.Operate;
12    end if;
13
14    S3.Set_Default;
15    S3.Operate;
16
17    S4.Operate;
18 end More_Use_Geometry;
```

The precondition of `More_Use_Geometry` specifies that `S1.Valid` holds, which takes into account both possible types of `S1`:

- If `S1` is dynamically a shape, then the precondition specifies that `Shape.Valid` holds, which ensures that the precondition to the call to `Shape.Operate` is satisfied on line 8.

- If `S1` is dynamically a rectangle, then the precondition specifies that `Rectangle.Valid` holds, which ensures that the precondition to the call to `Rectangle.Operate` is satisfied on line 8.

Similarly, the test on `S2.Valid` on line 10 ensures that the precondition to the call to `S2.Operate` on line 11 is satisfied, and the call to `S3.Set_Default` on line 14 ensures through its postcondition that the precondition to the call to `S3.Operate` on line 15 is satisfied. But no precondition or test or call ensures that the precondition to the call to `S4.Operate` on line 17 is satisfied. Hence the results of GNATprove's analysis on this program:

```
geometry.ads:9:13: info: implicit aspect Always_Terminates on "Valid" has been proved,␣
↪subprogram will terminate
geometry.ads:31:13: info: implicit aspect Always_Terminates on "Valid" has been proved,␣
↪subprogram will terminate
geometry.ads:43:20: info: class-wide postcondition is stronger than overridden one
geometry.ads:47:20: info: class-wide postcondition is stronger than overridden one
more_use_geometry.adb:8:06: info: precondition proved
more_use_geometry.adb:11:09: info: precondition proved
more_use_geometry.adb:15:06: info: precondition proved

more_use_geometry.adb:17:06: medium: precondition might fail
   17 |   S4.Operate;
      |   ~~^~~~~~~~
  possible fix: precondition of subprogram at line 3 should mention S4
    3 |procedure More_Use_Geometry (S1, S2, S3, S4 : in out Shape'Class) with
      |^ here
```

7.6 How to Write Package Contracts

Like for subprogram contracts, GNATprove can generate default package contracts when not specified by a user. By default, GNATprove does not require the user to write any package contracts.

The default state abstraction generated by GNATprove maps every internal global variable to a different internal abstract state (which is not really *abstract* as a result).

The default package initialization generated by GNATprove lists all variables initialized either at declaration or in the package body statements. The generated Initializes aspect is an over-approximation of the actual Initializes aspect. All outputs are considered to be initialized from all inputs. For example, consider package `Init_Data` which initializes all its global variables during elaboration, from either constants or variables:

```
1 package External_Data with
2   SPARK_Mode
3 is
4    Val : Integer with Import;
5 end External_Data;
```

```
1 with External_Data;
2 pragma Elaborate_All(External_Data);
3
4 package Init_Data with
5   SPARK_Mode
6 is
7    pragma Elaborate_Body;
8    Start_From_Zero     : Integer := 0;
9    Start_From_Val      : Integer := External_Data.Val;
```

(continues on next page)

(continued from previous page)

```
   Start_From_Zero_Bis : Integer;
   Start_From_Val_Bis  : Integer;
end Init_Data;
```

```
package body Init_Data with
  SPARK_Mode
is
begin
   Start_From_Zero_Bis := 0;
   Start_From_Val_Bis  := External_Data.Val;
end Init_Data;
```

GNATprove generates a package initialization contract on package `Init_Data` which is equivalent to:

```
Initializes => (Start_From_Zero     => External_Data.Val,
                Start_From_Zero_Bis => External_Data.Val,
                Start_From_Val      => External_Data.Val,
                Start_From_Val_Bis  => External_Data.Val)
```

As a result, GNATprove can check that global variables are properly initialized when calling the main procedure `Main_Proc`, and it does not issue any message when analyzing this code:

```
with Init_Data;
procedure Main_Proc with
  SPARK_Mode
is
   Tmp : Integer;
begin
   Tmp := Init_Data.Start_From_Zero;
   Tmp := Init_Data.Start_From_Val;
   Tmp := Init_Data.Start_From_Zero_Bis;
   Tmp := Init_Data.Start_From_Val_Bis;
end Main_Proc;
```

The user may specify explicitly package contracts to:

- name explicitly the parts of state abstraction that can be used in subprogram dependency contracts, in order to *Address Data and Control Coupling*; or
- improve scalability and running time of GNATprove's analysis, as a single explicit abstract state may be mapped to hundreds of concrete global variables, which would otherwise be considered separately in the analysis; or
- check that initialization of global data at elaboration is as specified in the specified package initialization contracts.

7.7 How to Write Loop Invariants

As described in *Loop Invariants*, proving properties of subprograms that contain loops may require the addition of explicit loop invariant contracts. This section describes a systematic approach for writing loop invariants.

7.7.1 Automatic Unrolling of Simple For-Loops

GNATprove automatically unrolls simple for-loops, defined as:

- for-loops over a range,
- with a number of iterations smaller than 20,
- without *Loop Invariants* or *Loop Variants*,
- that declare no local variables, or only variables of scalar type.

In addition, GNATprove always unrolls loops of the form `for J in 1 .. 1 loop` that don't have a *Loop Invariants* or *Loop Variants*, even when they declare local variables of non-scalar type.

As a result of unrolling, GNATprove conveys the exact meaning of the loop to provers, without requiring a loop invariant. While this is quite powerful, it is best applied to loops where the body of the loop is small, otherwise the unrolling may lead to complex formulas that provers cannot prove.

For example, consider the subprograms `Init` and `Sum` below:

```
package Loop_Unrolling with
  SPARK_Mode
is
   subtype Index is Integer range 1 .. 10;
   type Arr is array (Index) of Integer;

   procedure Init (A : out Arr) with
     Post => (for all J in Index => A(J) = J);

   function Sum (A : Arr) return Integer with
     Pre  => (for all J in Index => A(J) = J),
     Post => Sum'Result = (A'First + A'Last) * A'Length / 2;

end Loop_Unrolling;
```

```
package body Loop_Unrolling with
  SPARK_Mode
is
   procedure Init (A : out Arr) is
   begin
      for J in Index loop
         A (J) := J;
      end loop;
   end Init;

   function Sum (A : Arr) return Integer is
      Result : Integer := 0;
   begin
      for J in Index loop
```

(continues on next page)

(continued from previous page)

```
         Result := Result + A (J);
      end loop;
      return Result;
   end Sum;

end Loop_Unrolling;
```

As the loops in both subprograms are simple for-loops, GNATprove unrolls them and manages to prove the postconditions of `Init` and `Sum` without requiring a loop invariant:

```
loop_unrolling.adb:15:27: info: overflow check proved
loop_unrolling.ads:7:20: info: initialization of "A" proved
loop_unrolling.ads:8:14: info: postcondition proved
loop_unrolling.ads:10:13: info: implicit aspect Always_Terminates on "Sum" has been␣
↪proved, subprogram will terminate
loop_unrolling.ads:12:14: info: postcondition proved
```

Automatic loop unrolling can be disabled locally by explicitly adding a default loop invariant at the start of the loop:

```
for X in A .. B loop
   pragma Loop_Invariant (True);
   ...
end loop;
```

It can also be disabled globally by using the switch `--no-loop-unrolling`.

7.7.2 Automatically Generated Loop Invariants

In general, GNATprove relies on the user to manually supply the necessary information about variables modified by loop statements in the loop invariant. Though variables which are not modified in the loop need not be mentioned in the invariant, it is usually necessary to state explicitly the preservation of unmodified object parts, such as record or array components. In particular, when a loop modifies a collection, which can be either an array or a container (see *Formal Containers Library*), it may be necessary to state in the loop invariant those parts of the collection that have not been modified up to the current iteration. This property called *frame condition* in the scientific literature is essential for GNATprove, which otherwise must assume that all elements in the collection may have been modified. Special care should be taken to write adequate frame conditions, as they usually look obvious to programmers, and so it is very common to forget to write them and not being able to realize what's the problem afterwards.

To alleviate this problem, the GNATprove tool generates automatically frame conditions in some cases. As examples of use of such generated frame conditions, consider the code of procedures `Update_Arr` and `Update_Rec` below:

```
package Frame_Condition with
  SPARK_Mode
is
   type Index is range 1 .. 100;
   type Arr is array (Index) of Integer;

   procedure Update_Arr (A : in out Arr; Idx : Index) with
     Post => A(Idx + 1 .. A'Last) = A(Idx + 1 .. A'Last)'Old;

   type Rec is record
      A : Arr;
```

(continues on next page)

(continued from previous page)

```
      X : Integer;
   end record;

   procedure Update_Rec (R : in out Rec) with
     Post => R.X = R.X'Old;

end Frame_Condition;
```

```
package body Frame_Condition with
  SPARK_Mode
is
   procedure Update_Arr (A : in out Arr; Idx : Index) is
   begin
      for J in A'First .. Idx loop
         A(J) := Integer(J);
      end loop;
   end Update_Arr;

   procedure Update_Rec (R : in out Rec) is
   begin
      for J in R.A'Range loop
         R.A(J) := Integer(J);
      end loop;
   end Update_Rec;

end Frame_Condition;
```

Without this feature, GNATprove would not be able to prove the postconditions of either procedure because:

- To prove the postcondition of `Update_Arr`, one needs to know that only the indexes up to `Idx` have been updated in the loop.
- To prove the postcondition of `Update_Rec`, one needs to know that only the component `A` of record `R` has been updated in the loop.

Thanks to this feature, GNATprove automatically proves the postconditions of both procedures, without the need for loop invariants:

```
frame_condition.ads:8:14: info: range check proved
frame_condition.ads:8:14: info: postcondition proved
frame_condition.ads:8:37: info: range check proved
frame_condition.ads:16:14: info: postcondition proved
```

In particular, it is able to infer the preservation of unmodified components of record variables. It also handles unmodified components of array variables as long as they are preserved at every index in the array. As an example, consider the following loop which only updates some record component of a nested data structure:

```
procedure Preserved_Fields with SPARK_Mode is
   type R is record
      F1 : Integer := 0;
      F2 : Integer := 0;
   end record;

   type R_Array is array (1 .. 100) of R;
```

(continues on next page)

(continued from previous page)

```
8
9     type R_Array_Record is record
10       F3 : R_Array;
11       F4 : R_Array;
12    end record;
13
14    D : R_Array_Record;
15
16 begin
17    for I in 1 .. 100 loop
18       D.F3 (I).F1 := 0;
19       pragma Assert (for all J in 1 .. 100 =>
20                         D.F3 (J).F2 = D.F3'Loop_Entry (J).F2);
21       pragma Assert (D.F4 = D.F4'Loop_Entry);
22    end loop;
23
24 end Preserved_Fields;
```

Despite the absence of a loop invariant in the above code, GNATprove is able to prove that the assertions on lines 19-21 about variable `D` which is modified in the loop are proved, thanks to the generated loop invariants:

```
1 preserved_fields.adb:14:04: info: initialization of "D" proved
2 preserved_fields.adb:19:22: info: assertion proved
3 preserved_fields.adb:21:22: info: assertion proved
```

Note that GNATprove will not generate a frame condition for a record component if the record variable is modified as a whole either through an assignment or through a procedure call, et cetera, even if the component happens to be preserved by the modification.

GNATprove can also infer preservation of unmodified array components for arrays that are only updated at constant indexes or at indexes equal to the loop index. As an example, consider the following loops, only updating some cells of a matrix of arrays:

```
1  procedure Preserved_Components with SPARK_Mode is
2
3     type A is array (1 .. 100) of Natural with Default_Component_Value => 1;
4
5     type A_Matrix is array (1 .. 100, 1 .. 100) of A;
6
7     M : A_Matrix;
8
9  begin
10    L1: for I in 1 .. 100 loop
11       M (I, 1) (1 .. 50) := (others => 0);
12       pragma Assert
13         (for all K1 in 1 .. 100 =>
14            (for all K2 in 1 .. 100 =>
15                 (for all K3 in 1 .. 100 =>
16                      (if K1 > I or else K2 /= 1 or else K3 > 50 then
17                             M (K1, K2) (K3) = M'Loop_Entry (K1, K2) (K3)))));
18    end loop L1;
19
20    L2: for I in 1 .. 99 loop
```

(continues on next page)

(continued from previous page)

```
21       M (I + 1, I) (I .. 100) := (others => 0);
22       pragma Assert
23         (for all K1 in 1 .. 100 =>
24            (for all K2 in 1 .. 100 =>
25                 (for all K3 in 1 .. 100 =>
26                      (if K1 > I + 1 then
27                             M (K1, K2) (K3) = M'Loop_Entry (K1, K2) (K3)))));
28       pragma Assert
29         (for all K1 in 1 .. 100 =>
30            (for all K2 in 1 .. 100 =>
31                 (for all K3 in 1 .. 100 =>
32                      (if K3 < K2 then
33                             M (K1, K2) (K3) = M'Loop_Entry (K1, K2) (K3)))));
34    end loop L2;
35
36 end Preserved_Components;
```

Despite the absence of a loop invariant in the above code, GNATprove can succesfully verify the assertion on line 13 thanks to the generated loop invariant. Note that loop invariant generation for preserved array components is based on heuristics, and that it is therefore far from complete. In particular, it does not handle updates to variable indexes different from the loop index, as can be seen by the failed attempt to verify the assertion on line 22. GNATprove does not either handle dependences between indexes in an update, resulting in the failed attempt to verify the assertion on line 33:

```
1  preserved_components.adb:7:04: info: initialization of "M" proved
2  preserved_components.adb:13:10: info: assertion proved
3  preserved_components.adb:21:07: info: range check proved
4  preserved_components.adb:21:21: info: index check proved
5  preserved_components.adb:21:26: info: index check proved
6  preserved_components.adb:21:31: info: length check proved
7  preserved_components.adb:21:34: info: length check proved
8
9  preserved_components.adb:27:30: medium: assertion might fail, cannot prove M (K1, K2)␣
   ↪(K3) = M'Loop_Entry (K1, K2) (K3)
10    27 |                             M (K1, K2) (K3) = M'Loop_Entry (K1, K2) (K3)));
11       |                             ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
12   possible fix: loop at line 20 should mention M in a loop invariant
13    20 |   L2: for I in 1 .. 99 loop
14       |                        ^ here
15
16 preserved_components.adb:33:30: medium: assertion might fail, cannot prove M (K1, K2)␣
   ↪(K3) = M'Loop_Entry (K1, K2) (K3)
17    33 |                             M (K1, K2) (K3) = M'Loop_Entry (K1, K2) (K3)));
18       |                             ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
19   possible fix: loop at line 20 should mention M in a loop invariant
20    20 |   L2: for I in 1 .. 99 loop
21       |                        ^ here
```

7.7.3 The Four Properties of a Good Loop Invariant

A loop invariant can describe more or less precisely the behavior of a loop. What matters is that the loop invariant allows proving absence of run-time errors in the subprogram, that the subprogram respects its contract, and that the loop invariant itself holds at each iteration of the loop. There are four properties that a good loop invariant should fulfill:

1. [INIT] It should be provable in the first iteration of the loop.
2. [INSIDE] It should allow proving absence of run-time errors and local assertions inside the loop.
3. [AFTER] It should allow proving absence of run-time errors, local assertions and the subprogram postcondition after the loop.
4. [PRESERVE] It should be provable after the first iteration of the loop.

As a first example, here is a variant of the search algorithm described in *SPARK Tutorial*, which returns whether a collection contains a desired value, and if so, at which index. The collection is implemented as an array.

The specification of `Linear_Search` is given in file `linear_search.ads`. The postcondition of `Search` expresses that, either the search returns a result within the array bounds, in which case it is the desired index, otherwise the array does not contain the value searched.

```
package Linear_Search
  with SPARK_Mode
is
   type Opt_Index is new Natural;

   subtype Index is Opt_Index range 1 .. Opt_Index'Last - 1;

   No_Index : constant Opt_Index := 0;

   type Ar is array (Index range <>) of Integer;

   function Search (A : Ar; I : Integer) return Opt_Index with
     Post => (if Search'Result in A'Range then A (Search'Result) = I
              else (for all K in A'Range => A (K) /= I));

end Linear_Search;
```

The implementation of `Linear_Search` is given in file `linear_search.adb`. The loop invariant of `Search` expresses that, at the end of each iteration, if the loop has not been exited before, then the value searched is not in the range of indexes between the start of the array `A'First` and the current index `Pos`. The loop variant is used to prove termination of the loop.

```
package body Linear_Search
  with SPARK_Mode
is

   function Search (A : Ar; I : Integer) return Opt_Index is
   begin
      for Pos in A'Range loop
         if A (Pos) = I then
            return Pos;
         end if;

         pragma Loop_Invariant (for all K in A'First .. Pos => A (K) /= I);
```

(continues on next page)

(continued from previous page)

```
13       end loop;
14
15       return No_Index;
16    end Search;
17
18 end Linear_Search;
```

With this loop invariant, GNATprove is able to prove all checks in `Linear_Search`, both those related to absence of run-time errors and those related to verification of contracts:

```
1 linear_search.adb:9:20: info: range check proved
2 linear_search.adb:12:33: info: loop invariant preservation proved
3 linear_search.adb:12:33: info: loop invariant initialization proved
4 linear_search.adb:12:67: info: index check proved
5 linear_search.ads:12:13: info: implicit aspect Always_Terminates on "Search" has been␣
  ↪proved, subprogram will terminate
6 linear_search.ads:13:14: info: postcondition proved
7 linear_search.ads:13:57: info: index check proved
8 linear_search.ads:14:48: info: index check proved
```

In particular, the loop invariant fulfills all four properties that we listed above:

1. [INIT] It is proved in the first iteration (message on line 3).
2. [INSIDE] It allows proving absence of run-time errors inside the loop (messages on lines 1 and 4).
3. [AFTER] It allows proving absence of run-time errors after the loop (messages on lines 6 and 7) and the subprogram postcondition (message on line 5).
4. [PRESERVE] It is proved after the first iteration (message on line 2).

Note that the loop invariant closely resembles the second line in the postcondition of the subprogram, except with a different range of values in the quantification: instead of stating a property for all indexes in the array `A`, the loop invariant states the same property for all indexes up to the current loop index `Pos`. In fact, if we equate `Pos` to `A'Last` for the last iteration of the loop, the two properties are equal. This explains here how the loop invariant allows proving the subprogram postcondition when the value searched is not found.

Note also that we chose to put the loop invariant at the end of the loop. We could as easily put it at the start of the loop. In that case, the range of values in the quantification should be modified to state that, at the start of each iteration, if the loop has not been exited before, then the value searched is not in the range of indexes between the start of the array `A'First` and the current index `Pos` *excluded*:

```
pragma Loop_Invariant (for all K in A'First .. Pos - 1 => A (K) /= I);
```

Indeed, the test for the value at index `Pos` is done after the loop invariant in that case.

We will now demonstrate techniques to complete a loop invariant so that it fulfills all four properties [INIT], [INSIDE], [AFTER] and [PRESERVE], on a more complex algorithm searching in an ordered collection of elements. Like the naive search algorithm just described, this algorithm returns whether the collection contains the desired value, and if so, at which index. The collection is also implemented as an array.

The specification of this `Binary_Search` is given in file `binary_search.ads`:

```
1 package Binary_Search
2   with SPARK_Mode
3 is
4    type Opt_Index is new Natural;
```

(continues on next page)

(continued from previous page)

```
   subtype Index is Opt_Index range 1 .. Opt_Index'Last - 1;

   No_Index : constant Opt_Index := 0;

   type Ar is array (Index range <>) of Integer;

   function Empty (A : Ar) return Boolean is (A'First > A'Last);

   function Sorted (A : Ar) return Boolean is
      (for all I1 in A'Range =>
         (for all I2 in I1 .. A'Last => A (I1) <= A (I2)));

   function Search (A : Ar; I : Integer) return Opt_Index with
     Pre  => Sorted (A),
     Post => (if Search'Result in A'Range then A (Search'Result) = I
              else (for all Index in A'Range => A (Index) /= I));

end Binary_Search;
```

The implementation of `Binary_Search` is given in file `binary_search.adb`:

```
package body Binary_Search
  with SPARK_Mode
is

   function Search (A : Ar; I : Integer) return Opt_Index is
      Left  : Index;
      Right : Index;
      Med   : Index;
   begin
      if Empty (A) then
         return No_Index;
      end if;

      Left  := A'First;
      Right := A'Last;

      if Left = Right and A (Left) = I then
         return Left;
      elsif A (Left) > I or A (Right) < I then
         return No_Index;
      end if;

      while Left <= Right loop
         pragma Loop_Variant (Increases => Left, Decreases => Right);

         Med := Left + (Right - Left) / 2;

         if A (Med) < I then
            Left := Med + 1;
         elsif A (Med) > I then
```

(continues on next page)

(continued from previous page)

```
31             Right := Med - 1;
32          else
33             return Med;
34          end if;
35       end loop;
36
37       return No_Index;
38    end Search;
39
40 end Binary_Search;
```

Note that, although function `Search` has a loop, we have not given an explicit loop invariant yet, so the default loop invariant of `True` will be used by GNATprove. We have added a loop variant to prove termination of the function. We are running GNATprove with a prover timeout of 60 seconds (switch `--timeout=60`) to get the results presented in the rest of this section.

7.7.4 Proving a Loop Invariant in the First Iteration

Property [INIT] is the easiest one to prove. This is equivalent to proving a pragma Assert in the sequence of statements obtained by unrolling the loop once. In particular, if the loop invariant is at the start of the loop, this is equivalent to proving a pragma Assert just before the loop. Therefore, the usual techniques for investigating unproved checks apply, see *How to Investigate Unproved Checks*.

7.7.5 Completing a Loop Invariant to Prove Checks Inside the Loop

Let's start by running GNATprove on program `Binary_Search` without loop invariant. It generates two medium messages, one corresponding to a possible run-time check failure, and one corresponding to a possible failure of the postcondition:

```
binary_search.adb:28:16: medium: array index check might fail
   28 |         if A (Med) < I then
      |               ^~~
  reason for check: value must be a valid index into the array
  possible fix: loop at line 23 should mention Med in a loop invariant
   23 |      while Left <= Right loop
      |                          ^ here

binary_search.ads:21:49: medium: postcondition might fail, cannot prove A (Index) /= I
   21 |              else (for all Index in A'Range => A (Index) /= I));
      |                                                ^~~~~~~~~~~~~~
```

We will focus here on the message inside the loop, which corresponds to property [INSIDE]. The problem is that variable `Med` varies in the loop, so GNATprove only knows that its value is in the range of its type `Index` at the start of an iteration (line 23), and that it is then assigned the value of `Left + (Right - Left) / 2` (line 24) before being used as an index into array `A` (lines 26 and 28) and inside expressions assigned to `Left` and `Right` (lines 27 and 29).

As `Left` and `Right` also vary in the loop, GNATprove cannot use the assignment on line 24 to compute a more precise range for variable `Med`, hence the message on index check.

What is needed here is a loop invariant that states that the values of `Left` and `Right` stay within the bounds of `A` inside the loop:

```
      while Left <= Right loop
         pragma Loop_Variant (Increases => Left, Decreases => Right);
         pragma Loop_Invariant (Left in A'Range and Right in A'Range);
```

With this simple loop invariant, GNATprove now reports that the check on line 26 is proved. GNATprove computes that the value assigned to `Med` in the loop is also within the bounds of `A`.

7.7.6 Completing a Loop Invariant to Prove Checks After the Loop

With the simple loop invariant given before, GNATprove still reports that the postcondition of `Search` may fail, which corresponds to property [AFTER]. By instructing GNATprove to prove checks progressively, as seens in *Proving SPARK Programs*, we even get a precise message pointing to the part of the postcondition that could not be proved:

```
binary_search.ads:21:49: medium: postcondition might fail, cannot prove A (Index) /= I
   21 |              else (for all Index in A'Range => A (Index) /= I));
      |                                                ^~~~~~~~~~~~~~
```

Here, the message shows that the second line of the postcondition could not be proved. This line expresses that, in the case where `Search` returns `No_Index` after the loop, the array `A` should not contain the value searched `I`.

One can very easily check that, if GNATprove can prove this property, it can also prove the postcondition. Simply insert a pragma Assert after the loop stating this property:

```
         end if;
      end loop;
      pragma Assert (for all Index in A'Range => A (Index) /= I);
```

GNATprove now succeeds in proving the postcondition, but it fails to prove the assertion:

```
binary_search.adb:37:50: medium: assertion might fail, cannot prove A (Index) /= I
   37 |      pragma Assert (for all Index in A'Range => A (Index) /= I);
      |                                                 ^~~~~~~~~~~~~~
  possible fix: precondition of subprogram at binary_search.ads:18 should mention I
   18 |   function Search (A : Ar; I : Integer) return Opt_Index with
      |   ^ here
```

The problem is that GNATprove only knows what the user specified about `A` in the precondition, namely that it is sorted in ascending order. Nowhere it is said that `A` does not contain the value `I`. Note that adding this assertion is not compulsory. It simply helps identifying what is needed to achieve property [AFTER], but it can be removed afterwards.

What is needed here is a loop invariant stating that, if `A` contains the value `I`, it must be at an index in the range `Left. .Right`, so when the loop exits because `Left > Right` (so the loop test becomes false), `A` cannot contain the value `I`.

One way to express this property is to state that the value of `A` at index `Left - 1` is less than `I`, while the value of `A` at index `Right + 1` is greater than `I`. Taking into account the possibility that there are no such indexes in `A` if either `Left` or `Right` are at the boundaries of the array, we can express it as follows:

```
      while Left <= Right loop
         pragma Loop_Variant (Increases => Left, Decreases => Right);
         pragma Loop_Invariant (Left in A'Range and Right in A'Range);
```

(continues on next page)

(continued from previous page)

```
      pragma Loop_Invariant (Left = A'First or else A (Left - 1) < I);
      pragma Loop_Invariant (Right = A'Last or else I < A (Right + 1));
```

GNATprove manages to prove these additional loop invariants, but it still cannot prove the assertion after the loop. The reason is both simple and far-reaching. Although the above loop invariant together with the property that the array is sorted imply the property we want to express, it still requires additional work for the prover to reach the same conclusion, which may prevent automatic proof in the allocated time. In that case, it is better to express the equivalent but more explicit property directly, as follows:

```
   while Left <= Right loop
      pragma Loop_Variant (Increases => Left, Decreases => Right);
      pragma Loop_Invariant (Left in A'Range and Right in A'Range);
      pragma Loop_Invariant
        (for all Index in A'First .. Left - 1 => A (Index) < I);
      pragma Loop_Invariant
        (for all Index in A'Range =>
           (if Index > Right then I < A (Index)));
```

GNATprove now proves the assertion after the loop. In general, it is simpler to understand the relationship between the loop invariant and the checks that follow the loop when the loop invariant is directly followed by the exit statement that controls loop termination. In a "for" or "while" loop, this can mean it is easier to place the Loop_Invariant pragmas at the *end* of the loop body, where they precede the (implicit) exit statement. In such cases, the loop invariant is more likely to resemble the postcondition.

7.7.7 Proving a Loop Invariant After the First Iteration

With the loop invariant given before, GNATprove also proves that the loop invariant of `Search` holds after the first iteration, which corresponds to property [PRESERVE]. In fact, we have now arrived at a loop invariant which allows GNATprove to prove all checks for subprogram `Search`.

This is not always the case. In general, when the loop invariant is not proved after the first iteration, the problem is that the loop invariant is not precise enough. The only information that GNATprove knows about the value of variables that are modified in the loop, at each loop iteration, is the information provided in the loop invariant. If the loop invariant is missing some crucial information about these variables, which is needed to prove the loop invariant after N iterations, GNATprove won't be able to prove that the loop invariant holds at each iteration.

In loops that modify variables of composite types (records and arrays), it is usually necessary at this stage to add in the loop invariant some information about those parts of the modified variables which are not modified by the loop, or which are not modified in the first N iterations of the loop. Otherwise, GNATprove assumes that these parts may also be modified, which can prevent it from proving the preservation of the loop invariant. See *Loop Invariants* for an example where this is needed.

In other cases, it may be necessary to guide the prover with intermediate assertions. A rule of thumb for deciding which properties to assert, and where to assert them, is to try to locate at which program point the prover does not success in proving the property of interest, and to restate other properties that are useful for the proof.

In yet other cases, where the difficulty is related to the size of the loop rather than the complexity of the properties, it may be useful to factor the loop into local subprograms so that the subprograms' preconditions and postconditions provide the intermediate assertions that are needed to prove the loop invariant.

7.8 How to Investigate Unproved Checks

One of the most challenging aspects of formal verification is the analysis of failed proofs. If GNATprove fails to prove automatically that a run-time check or an assertion holds, there might be various reasons:

- [CODE] The check or assertion does not hold, because the code is wrong.
- [ASSERT] The assertion does not hold, because it is incorrect.
- [SPEC] The check or assertion cannot be proved, because of some missing assertions about the behavior of the program.
- [MODEL] The check or assertion is not proved because of current limitations in the model used by GNATprove.
- [TIMEOUT] The check or assertion is not proved because the prover timeouts.
- [PROVER] The check or assertion is not proved because the prover is not smart enough.

7.8.1 Investigating Incorrect Code or Assertion

The first step is to check whether the code is incorrect [CODE] or the assertion is incorrect [ASSERT], or both. Since run-time checks and assertions can be executed at run time, one way to increase confidence in the correction of the code and assertions is to test the program on representative inputs. The following GNAT switches can be used:

- `-gnato`: enable run-time checking of intermediate overflows
- `-gnat-p`: reenable run-time checking even if `-gnatp` was used to suppress all checks
- `-gnata`: enable run-time checking of assertions

7.8.2 Investigating Unprovable Properties

The second step is to consider whether the property is provable [SPEC]. A check or assertion might be unprovable because a necessary annotation is missing:

- the precondition of the enclosing subprogram might be too weak; or
- the postcondition of a subprogram called might be too weak; or
- a loop invariant for an enclosing loop might be too weak; or
- a loop invariant for a loop before the check or assertion might be too weak.

In particular, GNATprove does not look into subprogram bodies, so all the necessary information for calls should be explicit in the subprogram contracts. GNATprove may emit a tentative fix for the unprovable property when it suspects a missing precondition, postcondition or loop invariant to be the cause of the unprovability. The fix part follows the usual message of the form:

```
file:line:col: severity: check might fail
```

with a text such as:

```
possible fix: subprogram at line xxx should mention Var in a precondition
possible fix: loop at line xxx should mention Var in a loop invariant
possible fix: call at line xxx should mention Var in a postcondition
```

A focused manual review of the code and assertions can efficiently diagnose many cases of missing annotations. Even when an assertion is quite large, GNATprove precisely locates the part that it cannot prove, which can help figuring

out the problem. It may useful to simplify the code during this investigation, for example by adding a simpler assertion and trying to prove it.

GNATprove provides path information that might help the code review. You can display inside the editor the path on which the proof failed, as described in *Running GNATprove from GNAT Studio*. In some cases, a counterexample is also generated on the path, with values of variables which exhibit the problem (see *Understanding Counterexamples*). In many cases, this is sufficient to spot a missing assertion.

A property can also be conceptually provable, but the model used by GNATprove can currently not reason about it [MODEL]. (See *GNATprove Limitations* for a list of the current limitations in GNATprove.) In particular using the following features of the language may yield checks that should be true, but cannot be proved:

- Floating point arithmetic
- The specific value of dispatching calls when the tag is known

In the cases where no prover can prove the check, the missing information can usually be added using `pragma Assume`.

It may be difficult sometimes to distinguish between unprovable properties and prover shortcomings (the next section). The most generally useful action to narrow down the issue to its core is to insert assertions in the code that *test* whether the property (or part of it) can be proved at some specific point in the program. For example, if a postcondition states a property (P or Q), and the implementation contains many branches and paths, try adding assertions that P holds or Q holds where they are expected to hold. This can help distinguish between the two cases:

- In the case of an unprovable property, this may point to a specific path in the program, and a specific part of the property, which cause the issue.
- In the case of a prover shortcoming, this may also help provers to manage to prove both the assertion and the property. Then, it is good practice to keep in the code only those assertions that help getting automatic proof, and to remove other assertions that were inserted during interaction.

When using switch `--info`, GNATprove issues information messages regarding internal decisions that could influence provability:

- whether candidate loops for *Automatic Unrolling of Simple For-Loops* are effectively unrolled or not;
- whether candidate subprograms for *Contextual Analysis of Subprograms Without Contracts* are effectively inlined for proof or not;
- whether possible subprogram nontermination impacts the proof of calls to that subprogram (see the note in the section on *Subprogram Termination*)

7.8.3 Investigating Prover Shortcomings

The last step is to investigate if the prover would find a proof given enough time [TIMEOUT] or if another prover can find a proof [PROVER]. To that end, GNATprove provides switch `--level`, usable either from the command-line (see *Running GNATprove from the Command Line*), inside GNAT Studio (see *Running GNATprove from GNAT Studio*) or inside GNATbench (see *Running GNATprove from GNATbench*). The level of 0 is only adequate for simple proofs. In general, one should increase the level of proof (up to level 4) until no more automatic proofs can be obtained.

As described in the section about *Running GNATprove from the Command Line*, switch `--level` is equivalent to setting directly various lower level switches like `--timeout`, `--prover`, and `--proof`. Hence, one can also set more powerful (and thus leading to longer proof time) values for the individual switches rather than using the predefined combinations set through `--level`.

Note that for the above experiments, it is quite convenient to use the *SPARK → Prove Line* or *SPARK → Prove Subprogram* menus in GNAT Studio, as described in *Running GNATprove from GNAT Studio* and *Running GNATprove from GNATbench*, to get faster results for the desired line or subprogram.

A current limitation of automatic provers is that they don't handle floating-point arithmetic very precisely, in particular when there are either a lot of operations, or some non-linear operations (multiplication, division, exponentiation).

Another common limitation of automatic provers is that they don't handle non-linear arithmetic well. For example, they might fail to prove simple checks involving multiplication, division, modulo or exponentiation.

In that case, a user may either:

- add in the code a call to a lemma from the SPARK lemma library (see details in *Manual Proof Using SPARK Lemma Library*), or
- add in the code a call to a user lemma (see details in *Manual Proof Using User Lemmas*), or
- add an assumption in the code (see details in *Indirect Justification with Pragma Assume*), or
- add a justification in the code (see details in *Direct Justification with Pragma Annotate*), or
- import the formula passed to the automatic prover in the language of an interactive prover, and complete the proof interactively (see details in *Calling an Interactive Prover From GNAT Studio*), or
- manually review the unproved checks and record that they can be trusted (for example by storing the result of GNATprove under version control).

For advanced users, in particular those who would like to do manual proof, we will provide a description of the format of the proof files generated by GNATprove, so that users can understand the actual files passed to the prover. Each individual file is stored under the sub-directory `gnatprove` of the project object directory (default is the project directory). The file name follows the convention:

```
<file>_<line>_<column>_<check>_<num>.<ext>
```

where:

- `file` is the name of the Ada source file for the check
- `line` is the line where the check appears
- `column` is the column
- `check` is an identifier for the check
- `num` is an optional number and identifies different paths through the program, between the start of the subprogram and the location of the check
- `ext` is the extension corresponding to the file format chosen. The format of the file depends on the prover used. For example, files for Alt-Ergo are are in Why3 format, and files for cvc5 are in SMTLIB2 format.

For example, the proof files generated for prover Alt-Ergo for a range check at line 160, column 42, of the file `f.adb` are stored in:

```
f.adb_160_42_range_check.why
f.adb_160_42_range_check_2.why
f.adb_160_42_range_check_3.why
...
```

Corresponding proof files generated for prover cvc5 are stored in:

```
f.adb_160_42_range_check.smt2
f.adb_160_42_range_check_2.smt2
f.adb_160_42_range_check_3.smt2
...
```

To be able to inspect these files, you should instruct GNATprove to keep them around by adding the switch `-d` to GNATprove's command line. You can also use the switch `-v` to get a detailed log of which proof files GNATprove is producing and attempting to prove.

7.8.4 Looking at Machine-Parsable GNATprove Output

GNATprove generates files which contain the results of SPARK analysis in machine-parsable form. These files are located in the `gnatprove` subdirectory of the project object directory, and have the suffix `.spark`. The structure of these files exposes internal details such as the exact way some checks are proved, therefore the structure of these files may change. Still, we provide here the structure of these files for convenience.

At various places in these files, we refer to entities. These are Ada entities, either subprograms or packages. Entities are defined by their name and their source location (file and line). In JSON this translates to the following dictionary for entities:

```
{ "name" : string,
  "file" : string,
  "line" : int }
```

A `.spark` file is of this form:

```
{ "spark"         : list spark_result,
  "flow"          : list flow_result,
  "pragma_assume" : list assume_result,
  "proof"         : list proof_result }
```

Each entry is mapped to a list of entries whose format is described below.

The `spark_result` entry is simply an entity, with an extra field for spark status, so that the entire dictionary looks like this:

```
spark_result = { "name"  : string,
                 "file"  : string,
                 "line"  : int,
                 "spark" : string }
```

Field "spark" takes value in "spec", "all" or "no" to denote respectively that only the spec is in SPARK, both spec/body are in SPARK (or spec is in SPARK for a package without body), or the spec is not in SPARK.

Entries for proof are of the following form:

```
proof_result =
  { "file"       : string,
    "line"       : int,
    "col"        : int,
    "suppressed" : string,
    "rule"       : string,
    "severity"   : string,
    "tracefile"  : string,
    "check_tree" : list goal,
    "msg_id"     : int,
    "how_proved" : string,
    "entity"     : entity }
```

- ("file", "line", "col") describe the source location of the message.
- "rule" describes the kind of check.
- "severity" describes the kind status of the message, possible values used by gnatwhy3 are "info", "low", "medium", "high" and "error".
- "tracefile" contains the name of a trace file, if any.

- "entity" contains the entity dictionary for the entity that this check belongs to.
- "msg_id" - if present indicates that this entry corresponds to a message issued on the commandline, with the exact same msg_id in brackets: "[#12]"
- "suppressed" - if present, the message is in fact suppressed by a pragma Annotate, and this field contains the justification message.
- "how_proved" - if present, indicates how the check has been proved (i.e. which prover). A special value is "interval", which designates the special interval analysis, done in the frontend. It has its own column in the summary table.
- "check_tree" basically contains a copy of the session tree in JSON format. It's a tree structure whose nodes are goals, transformations and proof attempts:

```
goal = { "transformations" : list trans,
         "pa"              : proof_attempt }

trans = { [transname : goal] }

proof_attempt = { [prover : infos] }

infos = { "time"   : float,
          "steps"  : integer,
          "result" : string }
```

Flow entries are of the same form as for proof. Differences are in the possible values for "rule", which can only be the ones for flow messages. Also "how_proved" field is never set.

The pragma Assume entries are of the form:

```
assume_result = { "file"   : string,
                  "line"   : int,
                  "col"    : int,
                  "entity" : entity }
```

- ("file", "line", "col") describe the source location of the pragma Assume statement.
- "entity" contains the entity dictionary for the entity that this pragma Assume belongs to.

7.8.5 Understanding Proof Strategies

We now explain in more detail how the provers are run on the logical formula(s) generated for a given check, a.k.a. Verification Conditions or VCs.

- In `per_check` mode, a single VC is generated for each check at the source level (e.g. an assertion, run-time check, or postcondition); in some cases two VCs can appear. Before attempting proof, this VC is then split into the conjuncts, that is, the parts that are combined with `and` or `and then`. All provers are tried on the VCs obtained in this way until one of them proves the VC or no more provers are left.
- In `per_path` mode, a VC is generated not only for each check at the source level, but for each path to the check. For example, for an assertion that appears after an if-statement, at least two VCs will be generated - one for each path through the if-statement. For each such VC, all provers are attempted. Unproved VCs are then further split into their conjuncts, and proof is again attempted.
- In `progressive` mode, first the actions described for `per_check` are tried. For all unproved VCs, the VC is then split into the paths that lead to the check, like for `per_path`. Each part is then attempted to be proved independently.

7.9 GNATprove by Example

GNATprove is based on advanced technology for modular static analysis and deductive verification. It is very different both from compilers, which do very little analysis of the code, and static analyzers, which execute symbolically the program. GNATprove does a very powerful local analysis of the program, but it generally does not cross subprogram boundaries. Instead, it uses the *Subprogram Contracts* provided by users to analyze calls. GNATprove also requires sometimes that users direct the analysis with *Assertion Pragmas*. Thus, it is essential to understand how GNATprove uses contracts and assertion pragmas. This section aims at providing a deeper insight into how GNATprove's flow analysis and proof work, through a step-by-step exploration of small code examples.

All the examples presented in this section, as well as some code snippets presented in the *Overview of SPARK Language*, are available in the example called `gnatprove_by_example` distributed with the SPARK toolset. It can be found in the `share/examples/spark` directory below the directory where the toolset is installed, and can be accessed from the IDE (either GNAT Studio or GNATBench) via the *Help* → *SPARK* → *Examples* menu item.

7.9.1 Basic Examples

The examples in this section have no loops, and do not use more complex features of SPARK like *Ghost Code*, *Interfaces to the Physical World*, or *Object Oriented Programming and Liskov Substitution Principle*.

Increment

Consider a simple procedure that increments its integer parameter `X`:

```
1 procedure Increment (X : in out Integer) with
2   SPARK_Mode
3 is
4 begin
5    X := X + 1;
6 end Increment;
```

As this procedure does not have a contract yet, GNATprove only checks that there are no possible reads of uninitialized data and no possible run-time errors in the procedure. Here, it issues a message about a possible overflow check failure on `X + 1`:

```
increment.adb:5:11: high: overflow check might fail, cannot prove upper bound for X + 1
    5 |   X := X + 1;
      |        ~~^~~
  e.g. when X = Integer'Last
  reason for check: result of addition must fit in a 32-bits machine integer
  possible fix: subprogram at line 1 should mention X in a precondition
    1 |procedure Increment (X : in out Integer) with
      |^ here
```

The counterexample displayed tells us that `Increment` could be called on value `Integer'Last` for parameter `X`, which would cause the increment to raise a run-time error. As suggested by the possible fix in the message issued by GNATprove, one way to eliminate this vulnerability is to add a precondition to `Increment` specifying that `X` should be less than `Integer'Last` when calling the procedure:

```
1 procedure Increment_Guarded (X : in out Integer) with
2   SPARK_Mode,
```

(continues on next page)

(continued from previous page)

```
  Pre => X < Integer'Last
is
begin
   X := X + 1;
end Increment_Guarded;
```

As this procedure has a contract now, GNATprove checks like before that there are no possible reads of uninitialized data and no possible run-time errors in the procedure, including in its contract, and that the procedure implements its contract. As expected, GNATprove now proves that there is no possible overflow check failure on `X + 1`:

```
increment_guarded.adb:6:11: info: overflow check proved
```

The precondition is usually the first contract added to a subprogram, but there are other *Subprogram Contracts*. Here is a version of `Increment` with:

- global dependencies (aspect `Global`) stating that the procedure reads and writes no global variables
- flow dependencies (aspect `Depends`) stating that the final value of parameter `X` only depends on its input value
- a precondition (aspect `Pre`) stating that parameter `X` should be less than `Integer'Last` on entry
- a postcondition (aspect `Post`) stating that parameter `X` should have been incremented by the procedure on exit

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

GNATprove checks that `Increment_Full` implements its contract, and that it cannot raise run-time errors or read uninitialized data. By default, GNATprove's output is empty in such a case, but we can request that it prints one line per check proved by using switch `--report=all`, which we do here:

```
increment_full.adb:3:03: info: data dependencies proved
increment_full.adb:4:03: info: flow dependencies proved
increment_full.adb:6:14: info: postcondition proved
increment_full.adb:6:24: info: overflow check proved
increment_full.adb:9:11: info: overflow check proved
```

As subprogram contracts are used to analyze callers of a subprogram, let's consider a procedure `Increment_Calls` that calls the different versions of `Increment` presented so far:

```
with Increment;
with Increment_Guarded;
with Increment_Full;

procedure Increment_Calls with
  SPARK_Mode
is
   X : Integer;
```

(continues on next page)

(continued from previous page)

```
9  begin
10    X := 0;
11    Increment (X);
12    Increment (X);
13
14    X := 0;
15    Increment_Guarded (X);
16    Increment_Guarded (X);
17
18    X := 0;
19    Increment_Full (X);
20    Increment_Full (X);
21 end Increment_Calls;
```

GNATprove proves all preconditions expect the one on the second call to `Increment_Guarded`:

```
increment_calls.adb:8:04: info: initialization of "X" proved
increment_calls.adb:15:04: info: precondition proved

increment_calls.adb:16:04: medium: precondition might fail
   16 |    Increment_Guarded (X);
      |    ^~~~~~~~~~~~~~~~~~~
  e.g. when X = Integer'Last
  possible fix: call at line 15 should mention X (for argument X) in a postcondition
   15 |    Increment_Guarded (X);
      |    ^ here
increment_calls.adb:19:04: info: precondition proved
increment_calls.adb:20:04: info: precondition proved
```

`Increment` has no precondition, so there is nothing to check here except the initialization of `X` when calling `Increment` on lines 11 and 12. But remember that GNATprove did issue a message about a true vulnerability on `Increment`'s implementation.

This vulnerability was corrected by adding a precondition to `Increment_Guarded`. This has the effect of pushing the constraint on callers, here procedure `Increment_Calls`. As expected, GNATprove proves that the first call to `Increment_Guarded` on line 15 satisfies its precondition. But it does not prove the same for the second call to `Increment_Guarded` on line 16, because the value of `X` on line 16 was set by the call to `Increment_Guarded` on line 15, and the contract of `Increment_Guarded` does not say anything about the possible values of `X` on exit.

As suggested by the possible fix in the message issued by GNATprove, a postcondition like the one on `Increment_Full` is needed so that GNATprove can check the second call to increment `X`. As expected, GNATprove proves that both calls to `Increment_Full` on lines 19 and 20 satisfy their precondition.

In some cases, the user is not interested in specifying and verifying a complete contract like the one on `Increment_Full`, typically for helper subprograms defined locally in a subprogram or package body. GNATprove allows performing *Contextual Analysis of Subprograms Without Contracts* for these local subprograms. For example, consider a local definition of `Increment` inside procedure `Increment_Local`:

```
1  procedure Increment_Local with
2    SPARK_Mode
3  is
4     procedure Increment (X : in out Integer) is
5     begin
6        X := X + 1;
```

(continues on next page)

(continued from previous page)

```
7     end Increment;
8
9     X : Integer;
10
11  begin
12     X := 0;
13     Increment (X);
14     Increment (X);
15     pragma Assert (X = 2);
16  end Increment_Local;
```

Although `Increment` has no contract (like the previous non-local version), GNATprove proves that this program is free from run-time errors, and that the assertion on line 15 holds:

```
increment_local.adb:6:14: info: overflow check proved, in call inlined at increment_
↪local.adb:13
increment_local.adb:6:14: info: overflow check proved, in call inlined at increment_
↪local.adb:14
increment_local.adb:9:04: info: initialization of "X" proved
increment_local.adb:15:19: info: assertion proved
```

Swap

Consider a simple procedure that swaps its integer parameters `X` and `Y`, whose simple-minded implementation is wrong:

```
1  procedure Swap_Bad (X, Y : in out Integer) with
2    SPARK_Mode
3  is
4  begin
5     X := Y;
6     Y := X;
7  end Swap_Bad;
```

As this procedure does not have a contract yet, GNATprove only checks that there are no possible reads of uninitialized data and no possible run-time errors in the procedure. Here, it simply issues a warning:

```

swap_bad.adb:1:21: warning: unused initial value of "X"
    1 |procedure Swap_Bad (X, Y : in out Integer) with
      |                     ^ here
```

But we know the procedure is wrong, so we'd like to get an error of some sort! We could not detect it with GNATprove because the error is functional, and GNATprove cannot guess the intended functionality of `Swap_Bad`. Fortunately, we can give this information to GNATprove by adding a contract to `Swap_Bad`.

One such contract is the flow dependencies introduced by aspect `Depends`. Here it specifies that the final value of `X` (resp. `Y`) should depend on the initial value of `Y` (resp. `X`):

```
1  procedure Swap_Bad_Depends (X, Y : in out Integer) with
2    SPARK_Mode,
3    Depends => (X => Y, Y => X)
4  is
```

(continues on next page)

(continued from previous page)

```
begin
   X := Y;
   Y := X;
end Swap_Bad_Depends;
```

GNATprove issues 3 check messages (and a warning) on `Swap_Bad_Depends`:

```
swap_bad_depends.adb:1:29: warning: unused initial value of "X"
    1 |procedure Swap_Bad_Depends (X, Y : in out Integer) with
      |                             ^ here

swap_bad_depends.adb:3:03: medium: missing dependency "null => X"
    3 |  Depends => (X => Y, Y => X)
      |  ^~~~~~~~~~~~~~~~~~~~~~~~~

swap_bad_depends.adb:3:23: medium: missing self-dependency "Y => Y"
    3 |  Depends => (X => Y, Y => X)
      |                      ^ here

swap_bad_depends.adb:3:28: medium: incorrect dependency "Y => X"
    3 |  Depends => (X => Y, Y => X)
      |                           ^ here
```

The last message informs us that the dependency `Y => X` stated in `Swap_Bad_Depends`'s contract is incorrect for the given implementation. That might be either an error in the code or an error in the contract. Here this is an error in the code. The two other messages are consequences of this error.

Another possible contract is the postcondition introduced by aspect `Post`. Here it specifies that the final value of `X` (resp. `Y`) is equal to the initial value of `Y` (resp. `X`):

```
procedure Swap_Bad_Post (X, Y : in out Integer) with
  SPARK_Mode,
  Post => X = Y'Old and Y = X'Old
is
begin
   X := Y;
   Y := X;
end Swap_Bad_Post;
```

GNATprove issues one check message on the unproved postcondition of `Swap_Bad_Post` (and a warning), with a counterexample giving concrete values of a wrong execution:

```
swap_bad_post.adb:1:26: warning: unused initial value of "X"
    1 |procedure Swap_Bad_Post (X, Y : in out Integer) with
      |                          ^ here

swap_bad_post.adb:3:25: high: postcondition might fail, cannot prove Y = X'Old
    3 |  Post => X = Y'Old and Y = X'Old
      |                        ^~~~~~~~
  e.g. when X'Old = 1
       and Y = 0
```

Both the check messages on `Swap_Bad_Depends` and on `Swap_Bad_Post` inform us that the intended functionality as expressed in the contracts is not implemented in the procedure. And looking again at the warning issued by GNATprove on `Swap_Bad`, this was already pointing at the same issue: swapping the values of `X` and `Y` should obviously lead to reading the initial value of `X`; the fact that this value is not used is a clear sign that there is an error in the implementation. The correct version of `Swap` uses a temporary value to hold the value of `X`:

```
procedure Swap (X, Y : in out Integer) with
  SPARK_Mode,
  Depends => (X => Y, Y => X),
  Post    => X = Y'Old and Y = X'Old
is
   Tmp : constant Integer := X;
begin
   X := Y;
   Y := Tmp;
end Swap;
```

GNATprove proves both contracts on `Swap` and it informs us that the postcondition was proved:

```
swap.adb:3:03: info: flow dependencies proved
swap.adb:4:14: info: postcondition proved
```

Let's now consider a well-known *in place* implementation of `Swap` that avoids introducing a temporary variable by using bitwise operations:

```
with Interfaces; use Interfaces;

procedure Swap_Modulo (X, Y : in out Unsigned_32) with
  SPARK_Mode,
  Post => X = Y'Old and Y = X'Old
is
begin
   X := X xor Y;
   Y := X xor Y;
   X := X xor Y;
end Swap_Modulo;
```

GNATprove understands the bitwise operations on values of modular types, and it proves here that the postcondition of `Swap_Modulo` is proved:

```
swap_modulo.adb:5:11: info: postcondition proved
```

GNATprove's flow analysis issues warnings like the one on `Swap_Bad` whenever it detects that some variables or statements are not used in the computation, which is likely uncovering an error. For example, consider procedure `Swap_Warn` which assigns `X` and `Tmp_Y` out of order:

```
procedure Swap_Warn (X, Y : in out Integer) with
  SPARK_Mode
is
   Tmp_X : Integer;
   Tmp_Y : Integer;
begin
   Tmp_X := X;
   X := Tmp_Y;
```

(continues on next page)

(continued from previous page)

```
   Tmp_Y := Y;
   Y := Tmp_X;
end Swap_Warn;
```

On this wrong implementation, GNATprove issues a high check message for the certain read of an uninitialized variable, and three warnings that point to unused constructs:

```
swap_warn.adb:1:25: warning: unused initial value of "Y"
    1 |procedure Swap_Warn (X, Y : in out Integer) with
      |                         ^ here

swap_warn.adb:8:09: high: "Tmp_Y" is not initialized
    8 |   X := Tmp_Y;
      |        ^~~~
```

In general, warnings issued by GNATprove's flow analysis should be carefully reviewed, as they may lead to the discovery of errors in the program.

Addition

Consider a simple function `Addition` that returns the sum of its integer parameters `X` and `Y`. As in *Increment*, we add a suitable precondition and postcondition for this function:

```
function Addition (X, Y : Integer) return Integer with
  SPARK_Mode,
  Depends => (Addition'Result => (X, Y)),
  Pre     => X + Y in Integer,
  Post    => Addition'Result = X + Y
is
begin
   return X + Y;
end Addition;
```

We also added flow dependencies to `Addition` for illustration purposes, but they are the same as the default generated ones (the result of the function depends on all its inputs), so are not in general given explicitly.

GNATprove issues a check message about a possible overflow in the precondition of `Addition`:

```
addition.adb:4:16: high: overflow check might fail, cannot prove lower bound for X + Y
    4 |  Pre     => X + Y in Integer,
      |             ~~^~~
  e.g. when X = Integer'First
        and Y = -1
  reason for check: result of addition must fit in a 32-bits machine integer
  possible fix: use pragma Overflow_Mode or switch -gnato13 or unit SPARK.Big_Integers
```

Indeed, if we call for example `Addition` on values `Integer'Last` for `X` and `1` for `Y`, the expression `X + Y` evaluated in the precondition does not fit in a machine integer and raises an exception at run time. In this specific case, some people may consider that it does not really matter that an exception is raised due to overflow as the failure of the precondition should also raise a run-time exception. But in general the precondition should not fail (just consider the

precondition `X + Y not in Integer` for example), and even here, the different exceptions raised may be treated differently (`Constraint_Error` in the case of an overflow, `Assertion_Error` in the case of a failing precondition).

One way to avoid this vulnerability is to rewrite the precondition so that no overflow can occur:

```
function Addition (X, Y : Integer) return Integer with
  SPARK_Mode,
  Depends => (Addition'Result => (X, Y)),
  Pre     => (X >= 0 and then Y <= Integer'Last - X) or else (X < 0 and then Y >= Integer
↪'First - X),
  Post    => Addition'Result = X + Y
is
begin
   return X + Y;
end Addition;
```

Although GNATprove proves that `Addition` implements its contract and is free from run-time errors, the rewritten precondition is not so readable anymore:

```
addition.adb:1:10: info: implicit aspect Always_Terminates on "Addition" has been proved,
↪ subprogram will terminate
addition.adb:3:03: info: flow dependencies proved
addition.adb:4:49: info: overflow check proved
addition.adb:4:97: info: overflow check proved
addition.adb:5:14: info: postcondition proved
addition.adb:5:34: info: overflow check proved
addition.adb:8:13: info: overflow check proved
```

A better way to achieve the same goal without losing in readability is to use the *Big Numbers Library* for arithmetic operations which could overflow:

```
with SPARK.Big_Integers;
use SPARK.Big_Integers;

function Addition (X, Y : Big_Integer) return Big_Integer with
  SPARK_Mode,
  Depends => (Addition'Result => (X, Y)),
  Post    => Addition'Result = X + Y
is
begin
   return X + Y;
end Addition;
```

In that case, GNATprove proves that there are no run-time errors in function `Addition`, and that it implements its contract:

```
addition.adb:4:10: info: implicit aspect Always_Terminates on "Addition" has been proved,
↪ subprogram will terminate
addition.adb:6:03: info: flow dependencies proved
addition.adb:7:14: info: postcondition proved
addition.adb:7:22: info: predicate check proved
addition.adb:7:32: info: predicate check proved
addition.adb:7:36: info: predicate check proved
addition.adb:10:11: info: predicate check proved
addition.adb:10:15: info: predicate check proved
```

Finally, we can choose to expand the range of applicability of the function, by accepting any values of inputs `X` and `Y`, and saturating when the addition would overflow the bounds of machine integers. That's what the rewritten function `Addition` does, and its saturating behavior is expressed in *Contract Cases*:

```
function Addition (X, Y : Integer) return Integer with
  SPARK_Mode,
  Contract_Cases => ((X + Y in Integer)    => Addition'Result = X + Y,
                     X + Y < Integer'First => Addition'Result = Integer'First,
                     X + Y > Integer'Last  => Addition'Result = Integer'Last)
is
begin
   if X < 0 and Y < 0 then -- both negative
      if X < Integer'First - Y then
         return Integer'First;
      else
         return X + Y;
      end if;

   elsif X > 0 and Y > 0 then -- both positive
      if X > Integer'Last - Y then
         return Integer'Last;
      else
         return X + Y;
      end if;

   else -- one positive or null, one negative or null, adding them is safe
      return X + Y;
   end if;
end Addition;
```

GNATprove proves that `Addition` implements its contract and is free from run-time errors:

```
addition.adb:1:10: info: implicit aspect Always_Terminates on "Addition" has been proved,
↪ subprogram will terminate
addition.adb:3:03: info: disjoint contract cases proved
addition.adb:3:03: info: complete contract cases proved
addition.adb:3:44: info: contract case proved
addition.adb:4:44: info: contract case proved
addition.adb:5:44: info: contract case proved
addition.adb:9:28: info: overflow check proved
addition.adb:12:19: info: overflow check proved
addition.adb:16:27: info: overflow check proved
addition.adb:19:19: info: overflow check proved
addition.adb:23:16: info: overflow check proved
```

Note that we analyzed this function in ELIMINATED overflow mode, using the switch `-gnato13`, otherwise there would be possible overflows in the guard expressions of the contract cases.

7.9.2 Loop Examples

The examples in this section contain loops, and thus require in general that users write suitable *Loop Invariants*. We start by explaining the need for a loop invariant, and we continue with a description of the most common patterns of loops and their loop invariant. We summarize each pattern in a table of the following form:

Loop Pattern	Loop Over Data Structure
Proof Objective	Establish property P.
Loop Behavior	Loops over the data structure and establishes P.
Loop Invariant	Property P is established for the part of the data structure looped over so far.

The examples in this section use the types defined in package `Loop_Types`:

```
with SPARK.Containers.Formal.Doubly_Linked_Lists;
with SPARK.Containers.Formal.Vectors;
with SPARK.Big_Integers; use SPARK.Big_Integers;

package Loop_Types
  with SPARK_Mode
is
   subtype Index_T is Positive range 1 .. 1000;
   subtype Opt_Index_T is Natural range 0 .. 1000;
   subtype Component_T is Natural;

   type Arr_T is array (Index_T) of Component_T;

   package Vectors is new SPARK.Containers.Formal.Vectors (Index_T, Component_T);
   subtype Vec_T is Vectors.Vector;

   package Lists is new SPARK.Containers.Formal.Doubly_Linked_Lists (Component_T);
   subtype List_T is Lists.List;

   type List_Cell;
   type List_Acc is access List_Cell;
   type List_Cell is record
      Value : Component_T;
      Next  : List_Acc;
   end record;

   function Length (L : access constant List_Cell) return Big_Natural is
     (if L = null then 0 else Length (L.Next) + 1)
   with Subprogram_Variant => (Structural => L);

   function At_End
     (L : access constant List_Cell) return access constant List_Cell
   is (L)
   with Ghost,
     Annotate => (GNATprove, At_End_Borrow);

   type Property is access function (X : Component_T) return Boolean;

   function For_All_List
```

(continues on next page)

(continued from previous page)

```
     (L : access constant List_Cell;
      P : not null Property) return Boolean
   is
     (L = null or else (P (L.Value) and then For_All_List (L.Next, P)))
   with
     Subprogram_Variant => (Structural => L);
   pragma Annotate (GNATprove, False_Positive, "call via access-to-subprogram",
                    "We only call For_All_List on terminating functions");

   type Relation is access function (X, Y : Component_T) return Boolean;

   function For_All_List
     (L1, L2 : access constant List_Cell;
      P      : not null Relation) return Boolean
   is
     ((L1 = null) = (L2 = null)
      and then
        (if L1 /= null
         then P (L1.Value, L2.Value)
         and then For_All_List (L1.Next, L2.Next, P)))
   with
     Subprogram_Variant => (Structural => L1);
   pragma Annotate (GNATprove, False_Positive, "call via access-to-subprogram",
                    "We only call For_All_List on terminating functions");

end Loop_Types;
```

As there is no built-in way to iterate over the elements of a recursive data structure, the first function `For_All_List` can be used to state that all elements of a list have a given property. The second variant of `For_All_List` takes two lists and states that both lists have the same number of elements and that the corresponding elements of both lists are related by the given relation. The function `At_End` is used to refer to the value of a borrowed list or a local borrower at the end of the borrow, see *Annotation for Referring to a Value at the End of a Local Borrow* for more explanations.

Note: Although the structural subprogram variant of `For_All_List` is proved, this is not sufficient to prove the termination of `For_All_List`, as we have no way for now to state on the access-to-subprogram type Property that all elements of this type must terminate. Therefore, we justify this check, see section on *Justifying Check Messages*.

The Need for a Loop Invariant

Consider a simple procedure that increments its integer parameter `X` a number `N` of times:

```
procedure Increment_Loop (X : in out Integer; N : Natural) with
  SPARK_Mode,
  Pre  => X <= Integer'Last - N,
  Post => X = X'Old + N
is
begin
   for I in 1 .. N loop
      X := X + 1;
   end loop;
```

(continues on next page)

(continued from previous page)

```
end Increment_Loop;
```

The precondition of `Increment_Loop` ensures that there is no overflow when incrementing `X` in the loop, and its postcondition states that `X` has been incremented `N` times. This contract is a generalization of the contract given for a single increment in *Increment*. GNATprove does not manage to prove either the absence of overflow or the postcondition of `Increment_Loop`:

```
increment_loop.adb:4:11: medium: postcondition might fail
    4 |  Post => X = X'Old + N
      |          ^~~~~~~~~~~~
  possible fix: loop at line 7 should mention X in a loop invariant
    7 |   for I in 1 .. N loop
      |                  ^ here

increment_loop.adb:8:14: medium: overflow check might fail, cannot prove upper bound for␣
↪X + 1
    8 |      X := X + 1;
      |           ~~^~~
  reason for check: result of addition must fit in a 32-bits machine integer
  possible fix: loop at line 7 should mention X in a loop invariant
    7 |   for I in 1 .. N loop
      |                  ^ here
```

As described in *How to Write Loop Invariants*, this is because variable `X` is modified in the loop, hence GNATprove knows nothing about it unless it is stated in a loop invariant. If we add such a loop invariant, as suggested by the possible explanation in the message issued by GNATprove, that describes precisely the value of `X` in each iteration of the loop:

```
procedure Increment_Loop_Inv (X : in out Integer; N : Natural) with
  SPARK_Mode,
  Pre  => X <= Integer'Last - N,
  Post => X = X'Old + N
is
begin
   for I in 1 .. N loop
      X := X + 1;
      pragma Loop_Invariant (X = X'Loop_Entry + I);
   end loop;
end Increment_Loop_Inv;
```

then GNATprove proves both the absence of overflow and the postcondition of `Increment_Loop_Inv`:

```
increment_loop_inv.adb:3:29: info: overflow check proved
increment_loop_inv.adb:4:11: info: postcondition proved
increment_loop_inv.adb:4:21: info: overflow check proved
increment_loop_inv.adb:8:14: info: overflow check proved
increment_loop_inv.adb:9:30: info: loop invariant preservation proved
increment_loop_inv.adb:9:30: info: loop invariant initialization proved
increment_loop_inv.adb:9:47: info: overflow check proved
```

Fortunately, many loops fall into some broad categories for which the loop invariant is known. In the following sections, we describe these common patterns of loops and their loop invariant, which involve in general iterating over the content of a collection (either an array, a container from the *Formal Containers Library*, or a pointer-based linked list).

Initialization Loops

This kind of loops iterates over a collection to initialize every element of the collection to a given value:

Loop Pattern	Separate Initialization of Each Element
Proof Objective	Every element of the collection has a specific value.
Loop Behavior	Loops over the collection and initializes every element of the collection.
Loop Invariant	Every element initialized so far has its specific value.

In the simplest case, every element is assigned the same value. For example, in procedure `Init_Arr_Zero` below, value zero is assigned to every element of array `A`:

```
with Loop_Types; use Loop_Types;

procedure Init_Arr_Zero (A : out Arr_T) with
  SPARK_Mode,
  Post => (for all J in A'Range => A(J) = 0)
is
   pragma Annotate (GNATprove, False_Positive, """A"" might not be initialized",
                    "Entire array is initialized element-by-element in a loop");
begin
   for J in A'Range loop
      A(J) := 0;
      pragma Loop_Invariant (for all K in A'First .. J => A(K) = 0);
      pragma Annotate (GNATprove, False_Positive, """A"" might not be initialized",
                       "Part of array up to index J is initialized at this point");
   end loop;
end Init_Arr_Zero;
```

The loop invariant expresses that all elements up to the current loop index `J` have the value zero. With this loop invariant, GNATprove is able to prove the postcondition of `Init_Arr_Zero`, namely that all elements of the array have value zero:

```
init_arr_zero.adb:3:26: info: justified that "A" might not be initialized in "Init_Arr_
↪Zero"
init_arr_zero.adb:5:11: info: postcondition proved
init_arr_zero.adb:5:36: info: justified that "A" might not be initialized
init_arr_zero.adb:12:30: info: loop invariant initialization proved
init_arr_zero.adb:12:30: info: loop invariant preservation proved
init_arr_zero.adb:12:59: info: justified that "A" might not be initialized
init_arr_zero.adb:12:61: info: index check proved
```

In the example above, pragma Annotate is used in `Init_Arr_Zero` to justify a message issued by flow analysis, about the possible read of uninitialized value `A(K)` in the loop invariant. Indeed, flow analysis is not currently able to infer that all elements up to the loop index `J` have been initialized, hence it issues a message that `"A" might not be initialized`. For more details, see section on *Justifying Check Messages*.

To verify this loop completely, it is possible to annotate `A` with the Relaxed_Initialization aspect to use proof to verify its correct initialization (see *Aspect Relaxed_Initialization and Ghost Attribute Initialized* for more details). In this case, the loop invariant should be extended to state that the elements of `A` have been initialized by the loop up to the current index:

```
with Loop_Types; use Loop_Types;

procedure Init_Arr_Zero (A : out Arr_T) with
  SPARK_Mode,
  Relaxed_Initialization => A,
  Post => A'Initialized and then (for all J in A'Range => A(J) = 0)
is
begin
   for J in A'Range loop
      A(J) := 0;
      pragma Loop_Invariant (for all K in A'First .. J => A(K)'Initialized);
      pragma Loop_Invariant (for all K in A'First .. J => A(K) = 0);
   end loop;
end Init_Arr_Zero;
```

Remark that the postcondition of `Init_Arr_Zero` also needs to state that `A` is entirely initialized by the call.

Consider now a variant of the same initialization loop over a vector:

```
with Loop_Types; use Loop_Types; use Loop_Types.Vectors;

procedure Init_Vec_Zero (V : in out Vec_T) with
  SPARK_Mode,
  Post => (for all J in First_Index (V) .. Last_Index (V) => Element (V, J) = 0)
is
begin
   for J in First_Index (V) .. Last_Index (V) loop
      Replace_Element (V, J, 0);
      pragma Loop_Invariant (Last_Index (V) = Last_Index (V)'Loop_Entry);
      pragma Loop_Invariant (for all K in First_Index (V) .. J => Element (V, K) = 0);
   end loop;
end Init_Vec_Zero;
```

Like before, the loop invariant expresses that all elements up to the current loop index `J` have the value zero. Another loop invariant is needed here to express that the length of the vector does not change in the loop: as variable `V` is modified in the loop, GNATprove does not know its length stays the same (for example, calling procedure `Append` or `Delete_Last` would change this length) unless the user says so in the loop invariant. This is different from arrays whose length cannot change. With this loop invariant, GNATprove is able to prove the postcondition of `Init_Vec_Zero`, namely that all elements of the vector have value zero:

```
init_vec_zero.adb:5:11: info: postcondition proved
init_vec_zero.adb:5:62: info: precondition proved
init_vec_zero.adb:5:74: info: range check proved
init_vec_zero.adb:9:07: info: precondition proved
init_vec_zero.adb:10:30: info: loop invariant preservation proved
init_vec_zero.adb:10:30: info: loop invariant initialization proved
init_vec_zero.adb:11:30: info: loop invariant preservation proved
init_vec_zero.adb:11:30: info: loop invariant initialization proved
init_vec_zero.adb:11:67: info: precondition proved
init_vec_zero.adb:11:79: info: range check proved
```

Similarly, consider a variant of the same initialization loop over a list:

```
with Loop_Types; use Loop_Types; use Loop_Types.Lists;
with Ada.Containers; use Ada.Containers; use Loop_Types.Lists.Formal_Model;

procedure Init_List_Zero (L : in out List_T) with
  SPARK_Mode,
  Post => (for all E of L => E = 0)
is
   Cu : Cursor := First (L);
begin
   while Has_Element (L, Cu) loop
      pragma Loop_Invariant (for all I in 1 .. P.Get (Positions (L), Cu) - 1 =>
                               Element (Model (L), I) = 0);
      Replace_Element (L, Cu, 0);
      Next (L, Cu);
   end loop;
end Init_List_Zero;
```

Contrary to arrays and vectors, lists are not indexed. Instead, a cursor can be defined to iterate over the list. The loop invariant expresses that all elements up to the current cursor `Cu` have the value zero. To access the element stored at a given position in a list, we use the function `Model` which computes the mathematical sequence of the elements stored in the list. The position of a cursor in this sequence is retrieved using the `Positions` function. Contrary to the case of vectors, no loop invariant is needed to express that the length of the list does not change in the loop, because the postcondition remains provable here even if the length of the list changes. With this loop invariant, GNATprove is able to prove the postcondition of `Init_List_Zero`, namely that all elements of the list have value zero:

```
init_list_zero.adb:6:11: info: postcondition proved
init_list_zero.adb:11:30: info: loop invariant initialization proved
init_list_zero.adb:11:30: info: loop invariant preservation proved
init_list_zero.adb:11:49: info: precondition proved
init_list_zero.adb:12:32: info: precondition proved
init_list_zero.adb:13:07: info: precondition proved
init_list_zero.adb:14:07: info: precondition proved
```

The case of sets and maps is similar to the case of lists.

Note: The parameter of `Init_Vec_Zero` and `Init_List_Zero` is an in out parameter. This is because some components of the vector/list parameter are preserved by the initialization procedure (in particular the component corresponding to its length). This is different from `Init_Arr_Zero` which takes an out parameter, as all components of the array are initialized by the procedure (the bounds of an array are not modifiable, hence considered separately from the parameter mode).

Consider now a variant of the same initialization loop over a pointer-based list:

```
with Loop_Types; use Loop_Types;

package P with
  SPARK_Mode
is
   function Is_Zero (X : Component_T) return Boolean is
     (X = 0);

   procedure Init_List_Zero (L : access List_Cell) with
```

(continues on next page)

(continued from previous page)

```
     Post => For_All_List (L, Is_Zero'Access);
end P;
```

```
with Loop_Types; use Loop_Types;

package body P with
  SPARK_Mode
is
   procedure Init_List_Zero (L : access List_Cell) is
      B : access List_Cell := L;
   begin
      while B /= null loop
         pragma Loop_Invariant
           (if For_All_List (At_End (B), Is_Zero'Access)
            then For_All_List (At_End (L), Is_Zero'Access));
         B.Value := 0;
         B := B.Next;
      end loop;
   end Init_List_Zero;
end P;
```

Like in the other variants, the postcondition of `Init_List_Zero` states that the elements of the list `L` after the call are all `0`. It uses the `For_All_List` function from `Loop_Types` to quantify over all the elements of the list. The loop iterates over the list `L` using a local borrower `B` which is a local variable which borrows the ownership of a part of a datastructure for the duration of its scope, see *Borrowing* for more details. The loop invariant uses the `At_End` function to express properties about the values of `L` and `B` at the end of the borrow. It states that the elements of `L` at the end of the borrow will all be `0` if the elements of `B` at the end of the borrow are all `0`. This is provable because we know while verifying the invariant that the already traversed elements were all set to `0` and that they can no longer be changed during the scope of `B`. With this loop invariant, GNATprove is able to prove the postcondition of `Init_List_Zero`:

```
p.adb:11:13: info: loop invariant initialization proved
p.adb:11:13: info: loop invariant preservation proved
p.adb:11:49: info: null exclusion check proved
p.adb:12:51: info: null exclusion check proved
p.adb:13:11: info: pointer dereference check proved
p.adb:14:16: info: pointer dereference check proved
p.ads:6:13: info: implicit aspect Always_Terminates on "Is_Zero" has been proved,␣
↪subprogram will terminate
p.ads:10:14: info: postcondition proved
p.ads:10:38: info: null exclusion check proved
```

Consider now a case where the value assigned to each element is not the same. For example, in procedure `Init_Arr_Index` below, each element of array `A` is assigned the value of its index:

```
with Loop_Types; use Loop_Types;

procedure Init_Arr_Index (A : out Arr_T) with
  SPARK_Mode,
  Post => (for all J in A'Range => A(J) = J)
is
   pragma Annotate (GNATprove, False_Positive, """A"" might not be initialized",
                    "Entire array is initialized element-by-element in a loop");
```

(continues on next page)

(continued from previous page)

```
begin
   for J in A'Range loop
      A(J) := J;
      pragma Loop_Invariant (for all K in A'First .. J => A(K) = K);
      pragma Annotate (GNATprove, False_Positive, """A"" might not be initialized",
                       "Part of array up to index J is initialized at this point");
   end loop;
end Init_Arr_Index;
```

The loop invariant expresses that all elements up to the current loop index `J` have the value of their index. With this loop invariant, GNATprove is able to prove the postcondition of `Init_Arr_Index`, namely that all elements of the array have the value of their index:

```
init_arr_index.adb:3:27: info: justified that "A" might not be initialized in "Init_Arr_
↪Index"
init_arr_index.adb:5:11: info: postcondition proved
init_arr_index.adb:5:36: info: justified that "A" might not be initialized
init_arr_index.adb:12:30: info: loop invariant initialization proved
init_arr_index.adb:12:30: info: loop invariant preservation proved
init_arr_index.adb:12:59: info: justified that "A" might not be initialized
init_arr_index.adb:12:61: info: index check proved
```

As for `Init_Arr_Zero` above, it is possible to annotate `A` with the Relaxed_Initialization aspect to use proof to verify its correct initialization:

```
with Loop_Types; use Loop_Types;

procedure Init_Arr_Index (A : out Arr_T) with
  SPARK_Mode,
  Relaxed_Initialization => A,
  Post => A'Initialized and then (for all J in A'Range => A(J) = J)
is
begin
   for J in A'Range loop
      A(J) := J;
      pragma Loop_Invariant (for all K in A'First .. J => A(K)'Initialized);
      pragma Loop_Invariant (for all K in A'First .. J => A(K) = K);
   end loop;
end Init_Arr_Index;
```

Everything is proved by GNATprove:

```
init_arr_index.adb:6:11: info: postcondition proved
init_arr_index.adb:6:59: info: initialization check proved
init_arr_index.adb:11:30: info: loop invariant initialization proved
init_arr_index.adb:11:30: info: loop invariant preservation proved
init_arr_index.adb:11:61: info: index check proved
init_arr_index.adb:12:30: info: loop invariant preservation proved
init_arr_index.adb:12:30: info: loop invariant initialization proved
init_arr_index.adb:12:59: info: initialization check proved
init_arr_index.adb:12:61: info: index check proved
```

Similarly, variants of `Init_Vec_Zero` and `Init_List_Zero` that assign a different value to each element of the

collection would be proved by GNATprove.

Mapping Loops

This kind of loops iterates over a collection to map every element of the collection to a new value:

Loop Pattern	Separate Modification of Each Element
Proof Objective	Every element of the collection has an updated value.
Loop Behavior	Loops over the collection and updates every element of the collection.
Loop Invariant	Every element updated so far has its specific value.

In the simplest case, every element is assigned a new value based only on its initial value. For example, in procedure `Map_Arr_Incr` below, every element of array `A` is incremented by one:

```
with Loop_Types; use Loop_Types;

procedure Map_Arr_Incr (A : in out Arr_T) with
  SPARK_Mode,
  Pre  => (for all J in A'Range => A(J) /= Component_T'Last),
  Post => (for all J in A'Range => A(J) = A'Old(J) + 1)
is
begin
   for J in A'Range loop
      A(J) := A(J) + 1;
      pragma Loop_Invariant (for all K in A'First .. J => A(K) = A'Loop_Entry(K) + 1);
      --  The following loop invariant is generated automatically by GNATprove:
      --  pragma Loop_Invariant (for all K in J + 1 .. A'Last => A(K) = A'Loop_Entry(K));
   end loop;
end Map_Arr_Incr;
```

The loop invariant expresses that all elements up to the current loop index `J` have been incremented (using *Attribute Loop_Entry*). With this loop invariant, GNATprove is able to prove the postcondition of `Map_Arr_Incr`, namely that all elements of the array have been incremented:

```
map_arr_incr.adb:6:11: info: postcondition proved
map_arr_incr.adb:6:52: info: overflow check proved
map_arr_incr.adb:10:20: info: overflow check proved
map_arr_incr.adb:11:30: info: loop invariant initialization proved
map_arr_incr.adb:11:30: info: loop invariant preservation proved
map_arr_incr.adb:11:61: info: index check proved
map_arr_incr.adb:11:79: info: index check proved
map_arr_incr.adb:11:82: info: overflow check proved
```

Note that the commented loop invariant expressing that other elements have not been modified is not needed, as it is an example of *Automatically Generated Loop Invariants*.

Consider now a variant of the same initialization loop over a vector:

```
pragma Unevaluated_Use_Of_Old (Allow);
with Loop_Types; use Loop_Types; use Loop_Types.Vectors;
use Loop_Types.Vectors.Formal_Model;

```

(continues on next page)

(continued from previous page)

```
procedure Map_Vec_Incr (V : in out Vec_T) with
  SPARK_Mode,
  Pre  => (for all I in 1 .. Last_Index (V) =>
              Element (V, I) /= Component_T'Last),
  Post => Last_Index (V) = Last_Index (V)'Old
  and then (for all I in 1 .. Last_Index (V) =>
                Element (V, I) = Element (Model (V)'Old, I) + 1)
is
begin
   for J in 1 .. Last_Index (V) loop
      pragma Loop_Invariant (Last_Index (V) = Last_Index (V)'Loop_Entry);
      pragma Loop_Invariant
        (for all I in 1 .. J - 1 =>
           Element (V, I) = Element (Model (V)'Loop_Entry, I) + 1);
      pragma Loop_Invariant
        (for all I in J .. Last_Index (V) =>
           Element (V, I) = Element (Model (V)'Loop_Entry, I));
      Replace_Element (V, J, Element (V, J) + 1);
   end loop;
end Map_Vec_Incr;
```

Like before, we need an additionnal loop invariant to state that the length of the vector is not modified by the loop. The other two invariants are direct translations of those used for the loop over arrays: the first one expresses that all elements up to the current loop index `J` have been incremented, and the second one expresses that other elements have not been modified. Note that, as formal vectors are limited, we need to use the `Model` function of vectors to express the set of elements contained in the vector before the loop (using attributes `Loop_Entry` and `Old`). With this loop invariant, GNATprove is able to prove the postcondition of `Map_Vec_Incr`, namely that all elements of the vector have been incremented:

```
map_vec_incr.adb:8:16: info: precondition proved
map_vec_incr.adb:8:28: info: range check proved
map_vec_incr.adb:9:11: info: postcondition proved
map_vec_incr.adb:11:18: info: precondition proved
map_vec_incr.adb:11:30: info: range check proved
map_vec_incr.adb:11:35: info: precondition proved
map_vec_incr.adb:11:59: info: range check proved
map_vec_incr.adb:11:62: info: overflow check proved
map_vec_incr.adb:15:30: info: loop invariant initialization proved
map_vec_incr.adb:15:30: info: loop invariant preservation proved
map_vec_incr.adb:17:10: info: loop invariant initialization proved
map_vec_incr.adb:17:10: info: loop invariant preservation proved
map_vec_incr.adb:18:12: info: precondition proved
map_vec_incr.adb:18:24: info: range check proved
map_vec_incr.adb:18:29: info: precondition proved
map_vec_incr.adb:18:60: info: range check proved
map_vec_incr.adb:18:63: info: overflow check proved
map_vec_incr.adb:20:10: info: loop invariant initialization proved
map_vec_incr.adb:20:10: info: loop invariant preservation proved
map_vec_incr.adb:21:12: info: precondition proved
map_vec_incr.adb:21:24: info: range check proved
map_vec_incr.adb:21:29: info: precondition proved
map_vec_incr.adb:21:60: info: range check proved
```

(continues on next page)

(continued from previous page)

```
map_vec_incr.adb:22:07: info: precondition proved
map_vec_incr.adb:22:30: info: precondition proved
map_vec_incr.adb:22:45: info: overflow check proved
```

Similarly, consider a variant of the same mapping loop over a list:

```
with Loop_Types; use Loop_Types; use Loop_Types.Lists;
with Ada.Containers; use Ada.Containers; use Loop_Types.Lists.Formal_Model;

procedure Map_List_Incr (L : in out List_T) with
  SPARK_Mode,
  Pre  => (for all E of L => E /= Component_T'Last),
  Post => Length (L) = Length (L)'Old
  and then (for all I in 1 .. Length (L) =>
                Element (Model (L), I) = Element (Model (L'Old), I) + 1)
is
   Cu : Cursor := First (L);
begin
   while Has_Element (L, Cu) loop
      pragma Loop_Invariant (Length (L) = Length (L)'Loop_Entry);
      pragma Loop_Invariant
        (for all I in 1 .. P.Get (Positions (L), Cu) - 1 =>
           Element (Model (L), I) = Element (Model (L'Loop_Entry), I) + 1);
      pragma Loop_Invariant
        (for all I in P.Get (Positions (L), Cu) .. Length (L) =>
           Element (Model (L), I) = Element (Model (L'Loop_Entry), I));
      Replace_Element (L, Cu, Element (L, Cu) + 1);
      Next (L, Cu);
   end loop;
end Map_List_Incr;
```

Like before, we need to use a cursor to iterate over the list. The loop invariants express that all elements up to the current loop index `J` have been incremented and that other elements have not been modified. Note that it is necessary to state here that the length of the list is not modified during the loop. It is because the length is used to bound the quantification over the elements of the list both in the invariant and in the postcondition. With this loop invariant, GNATprove is able to prove the postcondition of `Map_List_Incr`, namely that all elements of the list have been incremented:

```
map_list_incr.adb:7:11: info: postcondition proved
map_list_incr.adb:9:18: info: precondition proved
map_list_incr.adb:9:43: info: precondition proved
map_list_incr.adb:9:70: info: overflow check proved
map_list_incr.adb:14:30: info: loop invariant initialization proved
map_list_incr.adb:14:30: info: loop invariant preservation proved
map_list_incr.adb:16:10: info: loop invariant initialization proved
map_list_incr.adb:16:10: info: loop invariant preservation proved
map_list_incr.adb:16:29: info: precondition proved
map_list_incr.adb:17:12: info: precondition proved
map_list_incr.adb:17:37: info: precondition proved
map_list_incr.adb:17:71: info: overflow check proved
map_list_incr.adb:19:10: info: loop invariant preservation proved
map_list_incr.adb:19:10: info: loop invariant initialization proved
map_list_incr.adb:19:24: info: precondition proved
```

(continues on next page)

(continued from previous page)

```
map_list_incr.adb:20:12: info: precondition proved
map_list_incr.adb:20:32: info: range check proved
map_list_incr.adb:20:37: info: precondition proved
map_list_incr.adb:20:68: info: range check proved
map_list_incr.adb:21:07: info: precondition proved
map_list_incr.adb:21:31: info: precondition proved
map_list_incr.adb:21:47: info: overflow check proved
map_list_incr.adb:22:07: info: precondition proved
```

Finally, consider a variant of the same mapping loop over a pointer-based list:

```
with Loop_Types; use Loop_Types;

package P with
  SPARK_Mode
is
   function Small_Enough (X : Component_T) return Boolean is
     (X /= Component_T'Last);
   function Bigger_Than_First (X : Component_T) return Boolean is
     (X /= Component_T'First);

   procedure Map_List_Incr (L : access List_Cell) with
     Pre  => For_All_List (L, Small_Enough'Access),
     Post => For_All_List (L, Bigger_Than_First'Access);
end P;
```

```
with Loop_Types; use Loop_Types;

package body P with
  SPARK_Mode
is
   procedure Map_List_Incr (L : access List_Cell) is
      B : access List_Cell := L;
   begin
      while B /= null loop
         pragma Loop_Invariant (For_All_List (B, Small_Enough'Access));
         pragma Loop_Invariant
           (if For_All_List (At_End (B), Bigger_Than_First'Access)
            then For_All_List (At_End (L), Bigger_Than_First'Access));
         B.Value := B.Value + 1;
         B := B.Next;
      end loop;
   end Map_List_Incr;
end P;
```

Like in the other variants, the precondition of `Map_List_Incr` states that all elements of the input list `L` are less than `Component_T'Last` before the call. It uses the `For_All_List` function from `Loop_Types` to quantify over all the elements of the list. The postcondition is weaker than in other variants of the loop. Indeed, referring to the value of a pointer-based datastructure before the call is not allowed in the SPARK language. Therefore we changed the postcondition to state instead that all elements of the list are bigger than `Component_T'First` after the call.

The loop iterates over the list `L` using a local borrower `B` which is a local variable which borrows the ownership of a part of a datastructure for the duration of its scope, see *Borrowing* for more details. The loop invariant is made of two parts.

The first one states that the initial property still holds on the elements of `L` accessible through `B`. The second uses the `At_End` function to express properties about the values of `L` and `B` at the end of the borrow. It states that the elements of `L` at the end of the borrow will have the `Bigger_Than_First` property if the elements of `B` at the end of the borrow have this property. This is provable because we know when verifying the invariant that the already traversed elements currently have the `Bigger_Than_First` property and that they can no longer be changed during the scope of `B`. With this loop invariant, GNATprove is able to prove the postcondition of `Map_List_Incr`:

```
p.adb:10:33: info: loop invariant preservation proved
p.adb:10:33: info: loop invariant initialization proved
p.adb:10:62: info: null exclusion check proved
p.adb:12:13: info: loop invariant initialization proved
p.adb:12:13: info: loop invariant preservation proved
p.adb:12:59: info: null exclusion check proved
p.adb:13:61: info: null exclusion check proved
p.adb:14:11: info: pointer dereference check proved
p.adb:14:22: info: pointer dereference check proved
p.adb:14:29: info: overflow check proved
p.adb:15:16: info: pointer dereference check proved
p.ads:6:13: info: implicit aspect Always_Terminates on "Small_Enough" has been proved,␣
↪subprogram will terminate
p.ads:8:13: info: implicit aspect Always_Terminates on "Bigger_Than_First" has been␣
↪proved, subprogram will terminate
p.ads:12:43: info: null exclusion check proved
p.ads:13:14: info: postcondition proved
p.ads:13:48: info: null exclusion check proved
```

If we want to retain the most precise postcondition relating the elements of the structure before and after the loop, we need to introduce a way to store the values of the list before the call in a separate data structure. In the following example, it is done by declaring a `Copy` function which returns a copy of its input list. In its postcondition, we use the two-valued `For_All_List` function to state that the elements of the new structure are equal to the elements of the input structure. An alternative could be to store the elements in a structure not subject to ownership like an array.

Note: The function `Copy` is marked as `Import` as it is not meant to be executed. It could be implemented in SPARK by returning a deep copy of the argument list, reallocating all cells of the list in the result.

```
1  with Loop_Types; use Loop_Types;
2
3  package P with
4    SPARK_Mode
5  is
6     function Small_Enough (X : Component_T) return Boolean is
7       (X /= Component_T'Last);
8
9     function Equal (X, Y : Component_T) return Boolean is (X = Y);
10
11    function Is_Incr (X, Y : Component_T) return Boolean is
12      (X < Y and then Y = X + 1);
13
14    function Copy (L : access List_Cell) return List_Acc with
15      Ghost,
16      Import,
17      Global   => null,
```

(continues on next page)

(continued from previous page)

```
     Post     => For_All_List (L, Copy'Result, Equal'Access);

   procedure Map_List_Incr (L : access List_Cell) with
     Pre  => For_All_List (L, Small_Enough'Access),
     Post => For_All_List (Copy (L)'Old, L, Is_Incr'Access);
   pragma Annotate (GNATprove, Intentional, "memory leak might occur",
                    "The code will be compiled with assertions disabled");
end P;
```

```
with Loop_Types; use Loop_Types;

package body P with
  SPARK_Mode
is
   procedure Map_List_Incr (L : access List_Cell) is
      L_Old : constant List_Acc := Copy (L) with Ghost;
      pragma Annotate (GNATprove, Intentional, "memory leak might occur",
                       "The code will be compiled with assertions disabled");
      B     : access List_Cell := L;
      B_Old : access constant List_Cell := L_Old with Ghost;
   begin
      while B /= null loop
         pragma Loop_Invariant (For_All_List (B, Small_Enough'Access));
         pragma Loop_Invariant (For_All_List (B, B_Old, Equal'Access));
         pragma Loop_Invariant
           (if For_All_List (B_Old, At_End (B), Is_Incr'Access)
            then For_All_List (L_Old, At_End (L), Is_Incr'Access));
         B.Value := B.Value + 1;
         B := B.Next;
         B_Old := B_Old.Next;
      end loop;
      pragma Assert
        (For_All_List (L_Old, At_End (L), Is_Incr'Access));
   end Map_List_Incr;
end P;
```

The postcondition of `Map_List_Incr` is similar to the postcondition of `Copy`. It uses the two-valued `For_All_List` function to relate the elements of `L` before and after the call. Like in the previous variant, the loop traverses `L` using a local borrower `B`. To be able to speak about the initial value of `L` in the invariant, we introduce a ghost constant `L_Old` storing a copy of this value. As we need to traverse both lists at the same time, we declare a ghost variable `B_Old` as a local observer of `L_Old`.

The loop invariant is made of three parts now. The first one is similar to the one in the previous example. The third loop invariant is a direct adaptation of the second loop invariant of the previous example. It states that if, at the end of the borrow, the values accessible through `B` are related to their equivalent element in `B_Old` through `Is_Incr`, then so are all the elements of `L`. The loop invariant in the middle states that the elements reachable through `B` have not been modified by the loop. GNATprove can verify these loop invariants as well as the postcondition of `Map_List_Incr`:

```
p.adb:7:07: info: absence of resource or memory leak at end of scope justified
p.adb:14:33: info: loop invariant initialization proved
p.adb:14:33: info: loop invariant preservation proved
p.adb:14:62: info: null exclusion check proved
```

(continues on next page)

(continued from previous page)

```
p.adb:15:33: info: loop invariant initialization proved
p.adb:15:33: info: loop invariant preservation proved
p.adb:15:62: info: null exclusion check proved
p.adb:17:13: info: loop invariant initialization proved
p.adb:17:13: info: loop invariant preservation proved
p.adb:17:56: info: null exclusion check proved
p.adb:18:58: info: null exclusion check proved
p.adb:19:11: info: pointer dereference check proved
p.adb:19:22: info: pointer dereference check proved
p.adb:19:29: info: overflow check proved
p.adb:20:16: info: pointer dereference check proved
p.adb:21:24: info: pointer dereference check proved
p.adb:24:10: info: assertion proved
p.adb:24:50: info: null exclusion check proved
p.ads:6:13: info: implicit aspect Always_Terminates on "Small_Enough" has been proved,␣
↪subprogram will terminate
p.ads:9:13: info: implicit aspect Always_Terminates on "Equal" has been proved,␣
↪subprogram will terminate
p.ads:11:13: info: implicit aspect Always_Terminates on "Is_Incr" has been proved,␣
↪subprogram will terminate
p.ads:12:28: info: overflow check proved
p.ads:21:43: info: null exclusion check proved
p.ads:22:14: info: postcondition proved
p.ads:22:28: info: absence of resource or memory leak justified
p.ads:22:52: info: null exclusion check proved
```

Note: The second loop invariant does not subsume the first. Indeed, proving that, if all elements of `L_Old` are small enough, so are all elements of an unknown observer `B_Old` of `L_Old`, is beyond the capacity of GNATprove.

Validation Loops

This kind of loops iterates over a collection to validate that every element of the collection has a valid value. The most common pattern is to exit or return from the loop if an invalid value if encountered:

Loop Pattern	Sequence Validation with Early Exit
Proof Objective	Determine (flag) if there are any invalid elements in a given collection.
Loop Behavior	Loops over the collection and exits/returns if an invalid element is encountered.
Loop Invariant	Every element encountered so far is valid.

Consider a procedure `Validate_Arr_Zero` that checks that all elements of an array `A` have value zero:

```
1  with Loop_Types; use Loop_Types;
2
3  procedure Validate_Arr_Zero (A : Arr_T; Success : out Boolean) with
4    SPARK_Mode,
5    Post => Success = (for all J in A'Range => A(J) = 0)
6  is
7  begin
```

(continues on next page)

(continued from previous page)

```
   for J in A'Range loop
      if A(J) /= 0 then
         Success := False;
         return;
      end if;
      pragma Loop_Invariant (for all K in A'First .. J => A(K) = 0);
   end loop;

   Success := True;
end Validate_Arr_Zero;
```

The loop invariant expresses that all elements up to the current loop index `J` have value zero. With this loop invariant, GNATprove is able to prove the postcondition of `Validate_Arr_Zero`, namely that output parameter `Success` is True if-and-only-if all elements of the array have value zero:

```
validate_arr_zero.adb:3:41: info: initialization of "Success" proved
validate_arr_zero.adb:5:11: info: postcondition proved
validate_arr_zero.adb:13:30: info: loop invariant initialization proved
validate_arr_zero.adb:13:30: info: loop invariant preservation proved
validate_arr_zero.adb:13:61: info: index check proved
```

Consider now a variant of the same validation loop over a vector:

```
with Loop_Types; use Loop_Types; use Loop_Types.Vectors;

procedure Validate_Vec_Zero (V : Vec_T; Success : out Boolean) with
  SPARK_Mode,
  Post => Success = (for all J in First_Index (V) .. Last_Index (V) => Element (V, J) =␣
↪0)
is
begin
   for J in First_Index (V) .. Last_Index (V) loop
      if Element (V, J) /= 0 then
         Success := False;
         return;
      end if;
      pragma Loop_Invariant (for all K in First_Index (V) .. J => Element (V, K) = 0);
   end loop;

   Success := True;
end Validate_Vec_Zero;
```

Like before, the loop invariant expresses that all elements up to the current loop index `J` have the value zero. Since variable `V` is not modified in the loop, no additional loop invariant is needed here for GNATprove to know that its length stays the same (this is different from the case of `Init_Vec_Zero` seen previously). With this loop invariant, GNATprove is able to prove the postcondition of `Validate_Vec_Zero`, namely that output parameter `Success` is True if-and-only-if all elements of the vector have value zero:

```
validate_vec_zero.adb:3:41: info: initialization of "Success" proved
validate_vec_zero.adb:5:11: info: postcondition proved
validate_vec_zero.adb:5:72: info: precondition proved
validate_vec_zero.adb:5:84: info: range check proved
validate_vec_zero.adb:9:10: info: precondition proved
```

(continues on next page)

(continued from previous page)

```
validate_vec_zero.adb:13:30: info: loop invariant initialization proved
validate_vec_zero.adb:13:30: info: loop invariant preservation proved
validate_vec_zero.adb:13:67: info: precondition proved
validate_vec_zero.adb:13:79: info: range check proved
```

Similarly, consider a variant of the same validation loop over a list:

```
with Loop_Types; use Loop_Types; use Loop_Types.Lists;
with Ada.Containers; use Ada.Containers; use Loop_Types.Lists.Formal_Model;

procedure Validate_List_Zero (L : List_T; Success : out Boolean) with
  SPARK_Mode,
  Post => Success = (for all E of L => E = 0)
is
   Cu : Cursor := First (L);
begin
   while Has_Element (L, Cu) loop
      pragma Loop_Invariant (for all I in 1 .. P.Get (Positions (L), Cu) - 1 =>
                               Element (Model (L), I) = 0);
      if Element (L, Cu) /= 0 then
         Success := False;
         return;
      end if;
      Next (L, Cu);
   end loop;

   Success := True;
end Validate_List_Zero;
```

Like in the case of `Init_List_Zero` seen previously, we need to define a cursor here to iterate over the list. The loop invariant expresses that all elements up to the current cursor `Cu` have the value zero. With this loop invariant, GNATprove is able to prove the postcondition of `Validate_List_Zero`, namely that output parameter `Success` is True if-and-only-if all elements of the list have value zero:

```
validate_list_zero.adb:4:43: info: initialization of "Success" proved
validate_list_zero.adb:6:11: info: postcondition proved
validate_list_zero.adb:11:30: info: loop invariant initialization proved
validate_list_zero.adb:11:30: info: loop invariant preservation proved
validate_list_zero.adb:11:49: info: precondition proved
validate_list_zero.adb:12:32: info: precondition proved
validate_list_zero.adb:13:10: info: precondition proved
validate_list_zero.adb:17:07: info: precondition proved
```

The case of sets and maps is similar to the case of lists.

Consider now a variant of the same validation loop over a pointer-based list:

```
with Loop_Types; use Loop_Types;

package P with
  SPARK_Mode
is
   function Is_Zero (X : Component_T) return Boolean is
```

(continues on next page)

(continued from previous page)

```
     (X = 0);

   procedure Validate_List_Zero
     (L       : access constant List_Cell;
      Success : out Boolean)
   with
     Post => Success = For_All_List (L, Is_Zero'Access);
end P;
```

```
with Loop_Types; use Loop_Types;

package body P with
  SPARK_Mode
is
   procedure Validate_List_Zero
     (L       : access constant List_Cell;
      Success : out Boolean)
   is
      C : access constant List_Cell := L;
   begin
      while C /= null loop
         pragma Loop_Invariant
           (For_All_List (L, Is_Zero'Access) = For_All_List (C, Is_Zero'Access));
         if C.Value /= 0 then
            Success := False;
            return;
         end if;
         C := C.Next;
      end loop;

      Success := True;
   end Validate_List_Zero;
end P;
```

The loop is implemented using a local observer (see *Observing*) which borrows a read-only permission over a part of a datastructure until the end of the scope of the observer. In the loop invariant, we cannot, like in the other versions of the algorithm, speak about the value of the elements which have already been traversed to say that they are all `0`. Instead, we state that the list `L` only contains `0` iff `C` only contains `0`. This is true since the loop exits as soon as a non-zero value is encountered. With this invariant, the postcondition can be proved by GNATprove:

```
p.adb:14:13: info: loop invariant initialization proved
p.adb:14:13: info: loop invariant preservation proved
p.adb:14:37: info: null exclusion check proved
p.adb:14:72: info: null exclusion check proved
p.adb:15:14: info: pointer dereference check proved
p.adb:19:16: info: pointer dereference check proved
p.ads:6:13: info: implicit aspect Always_Terminates on "Is_Zero" has been proved,␣
↪subprogram will terminate
p.ads:11:07: info: initialization of "Success" proved
p.ads:13:14: info: postcondition proved
p.ads:13:48: info: null exclusion check proved
```

A variant of the previous validation pattern is to continue validating elements even after an invalid value has been

encountered, which allows for example logging all invalid values:

Loop Pattern	Sequence Validation that Validates Entire Collection
Proof Objective	Determine (flag) if there are any invalid elements in a given collection.
Loop Behavior	Loops over the collection. If an invalid element is encountered, flag this, but keep validating (typically logging every invalidity) for the entire collection.
Loop Invariant	If invalidity is not flagged, every element encountered so far is valid.

Consider a variant of `Validate_Arr_Zero` that keeps validating elements of the array after a non-zero element has been encountered:

```
with Loop_Types; use Loop_Types;

procedure Validate_Full_Arr_Zero (A : Arr_T; Success : out Boolean) with
  SPARK_Mode,
  Post => Success = (for all J in A'Range => A(J) = 0)
is
begin
   Success := True;

   for J in A'Range loop
      if A(J) /= 0 then
         Success := False;
         --  perform some logging here instead of returning
      end if;
      pragma Loop_Invariant (Success = (for all K in A'First .. J => A(K) = 0));
   end loop;
end Validate_Full_Arr_Zero;
```

The loop invariant has been modified to state that all elements up to the current loop index J have value zero if-and-only-if the output parameter Success is True. This in turn requires to move the assignment of `Success` before the loop. With this loop invariant, GNATprove is able to prove the postcondition of `Validate_Full_Arr_Zero`, which is the same as the postcondition of `Validate_Arr_Zero`, namely that output parameter `Success` is True if-and-only-if all elements of the array have value zero:

```
validate_full_arr_zero.adb:3:46: info: initialization of "Success" proved
validate_full_arr_zero.adb:5:11: info: postcondition proved
validate_full_arr_zero.adb:15:30: info: loop invariant initialization proved
validate_full_arr_zero.adb:15:30: info: loop invariant preservation proved
validate_full_arr_zero.adb:15:72: info: index check proved
```

Similarly, variants of `Validate_Vec_Zero` and `Validate_List_Zero` that keep validating elements of the collection after a non-zero element has been encountered would be proved by GNATprove.

Counting Loops

This kind of loops iterates over a collection to count the number of elements of the collection that satisfy a given criterion:

Loop Pattern	Count Elements Satisfying Criterion
Proof Objective	Count elements that satisfy a given criterion.
Loop Behavior	Loops over the collection. Increments a counter each time the value of an element satisfies the criterion.
Loop Invariant	The value of the counter is either 0 when no element encountered so far satisfies the criterion, or a positive number bounded by the current iteration of the loop otherwise.

Consider a procedure `Count_Arr_Zero` that counts elements with value zero in array `A`:

```
with Loop_Types; use Loop_Types;

procedure Count_Arr_Zero (A : Arr_T; Counter : out Natural) with
  SPARK_Mode,
  Post => (Counter in 0 .. A'Length) and then
          ((Counter = 0) = (for all K in A'Range => A(K) /= 0))
is
begin
   Counter := 0;

   for J in A'Range loop
      if A(J) = 0 then
         Counter := Counter + 1;
      end if;
      pragma Loop_Invariant (Counter in 0 .. J);
      pragma Loop_Invariant ((Counter = 0) = (for all K in A'First .. J => A(K) /= 0));
   end loop;
end Count_Arr_Zero;
```

The loop invariant expresses that the value of `Counter` is a natural number bounded by the current loop index `J`, and that `Counter` is equal to zero exactly when all elements up to the current loop index have a non-zero value. With this loop invariant, GNATprove is able to prove the postcondition of `Count_Arr_Zero`, namely that output parameter `Counter` is a natural number bounded by the length of the array `A`, and that `Counter` is equal to zero exactly when all elements in `A` have a non-zero value:

```
count_arr_zero.adb:3:38: info: initialization of "Counter" proved
count_arr_zero.adb:5:11: info: postcondition proved
count_arr_zero.adb:13:29: info: overflow check proved
count_arr_zero.adb:15:30: info: loop invariant initialization proved
count_arr_zero.adb:15:30: info: loop invariant preservation proved
count_arr_zero.adb:16:30: info: loop invariant preservation proved
count_arr_zero.adb:16:30: info: loop invariant initialization proved
count_arr_zero.adb:16:78: info: index check proved
```

Consider now a variant of the same counting loop over a vector:

```
with Loop_Types; use Loop_Types; use Loop_Types.Vectors;

procedure Count_Vec_Zero (V : Vec_T; Counter : out Natural) with
  SPARK_Mode,
  Post => (Counter in 0 .. Natural (Length (V))) and then
          ((Counter = 0) = (for all K in First_Index (V) .. Last_Index (V) => Element (V,
↪ K) /= 0))
is
begin
   Counter := 0;

   for J in First_Index (V) .. Last_Index (V) loop
      if Element (V, J) = 0 then
         Counter := Counter + 1;
      end if;
      pragma Loop_Invariant (Counter in 0 .. J);
      pragma Loop_Invariant ((Counter = 0) = (for all K in First_Index (V) .. J =>␣
↪Element (V, K) /= 0));
   end loop;
end Count_Vec_Zero;
```

Like before, the loop invariant expresses that the value of `Counter` is a natural number bounded by the current loop index `J`, and that `Counter` is equal to zero exactly when all elements up to the current loop index have a non-zero value. With this loop invariant, GNATprove is able to prove the postcondition of `Count_Vec_Zero`, namely that output parameter `Counter` is a natural number bounded by the length of the vector `V`, and that `Counter` is equal to zero exactly when all elements in `V` have a non-zero value:

```
count_vec_zero.adb:3:38: info: initialization of "Counter" proved
count_vec_zero.adb:5:11: info: postcondition proved
count_vec_zero.adb:6:79: info: precondition proved
count_vec_zero.adb:6:91: info: range check proved
count_vec_zero.adb:12:10: info: precondition proved
count_vec_zero.adb:13:29: info: overflow check proved
count_vec_zero.adb:15:30: info: loop invariant initialization proved
count_vec_zero.adb:15:30: info: loop invariant preservation proved
count_vec_zero.adb:16:30: info: loop invariant initialization proved
count_vec_zero.adb:16:30: info: loop invariant preservation proved
count_vec_zero.adb:16:84: info: precondition proved
count_vec_zero.adb:16:96: info: range check proved
```

Search Loops

This kind of loops iterates over a collection to search an element of the collection that meets a given search criterion:

Loop Pattern	Search with Early Exit
Proof Objective	Find an element or position that meets a search criterion.
Loop Behavior	Loops over the collection. Exits when an element that meets the search criterion is found.
Loop Invariant	Every element encountered so far does not meet the search criterion.

Consider a procedure `Search_Arr_Zero` that searches an element with value zero in array `A`:

```
with Loop_Types; use Loop_Types;

procedure Search_Arr_Zero (A : Arr_T; Pos : out Opt_Index_T; Success : out Boolean) with
  SPARK_Mode,
  Post => Success = (for some J in A'Range => A(J) = 0) and then
          (if Success then A (Pos) = 0)
is
begin
   for J in A'Range loop
      if A(J) = 0 then
         Success := True;
         Pos := J;
         return;
      end if;
      pragma Loop_Invariant (for all K in A'First .. J => A(K) /= 0);
   end loop;

   Success := False;
   Pos := 0;
end Search_Arr_Zero;
```

The loop invariant expresses that all elements up to the current loop index `J` have a non-zero value. With this loop invariant, GNATprove is able to prove the postcondition of `Search_Arr_Zero`, namely that output parameter `Success` is True if-and-only-if there is an element of the array that has value zero, and that `Pos` is the index of such an element:

```
search_arr_zero.adb:3:39: info: initialization of "Pos" proved
search_arr_zero.adb:3:62: info: initialization of "Success" proved
search_arr_zero.adb:5:11: info: postcondition proved
search_arr_zero.adb:6:31: info: index check proved
search_arr_zero.adb:15:30: info: loop invariant initialization proved
search_arr_zero.adb:15:30: info: loop invariant preservation proved
search_arr_zero.adb:15:61: info: index check proved
```

Consider now a variant of the same search loop over a vector:

```
with Loop_Types; use Loop_Types; use Loop_Types.Vectors;

procedure Search_Vec_Zero (V : Vec_T; Pos : out Opt_Index_T; Success : out Boolean) with
```

(continues on next page)

(continued from previous page)

```
  SPARK_Mode,
  Post => Success = (for some J in First_Index (V) .. Last_Index (V) => Element (V, J) =
↪0) and then
          (if Success then Element (V, Pos) = 0)
is
begin
   for J in First_Index (V) .. Last_Index (V) loop
      if Element (V, J) = 0 then
         Success := True;
         Pos := J;
         return;
      end if;
      pragma Loop_Invariant (for all K in First_Index (V) .. J => Element (V, K) /= 0);
   end loop;

   Success := False;
   Pos := 0;
end Search_Vec_Zero;
```

Like before, the loop invariant expresses that all elements up to the current loop index `J` have a non-zero value. With this loop invariant, GNATprove is able to prove the postcondition of `Search_Vec_Zero`, namely that output parameter `Success` is True if-and-only-if there is an element of the vector that has value zero, and that `Pos` is the index of such an element:

```
search_vec_zero.adb:3:39: info: initialization of "Pos" proved
search_vec_zero.adb:3:62: info: initialization of "Success" proved
search_vec_zero.adb:5:11: info: postcondition proved
search_vec_zero.adb:5:73: info: precondition proved
search_vec_zero.adb:5:85: info: range check proved
search_vec_zero.adb:6:28: info: precondition proved
search_vec_zero.adb:10:10: info: precondition proved
search_vec_zero.adb:15:30: info: loop invariant preservation proved
search_vec_zero.adb:15:30: info: loop invariant initialization proved
search_vec_zero.adb:15:67: info: precondition proved
search_vec_zero.adb:15:79: info: range check proved
```

Similarly, consider a variant of the same search loop over a list:

```
with Loop_Types; use Loop_Types; use Loop_Types.Lists;
with Ada.Containers; use Ada.Containers; use Loop_Types.Lists.Formal_Model;

procedure Search_List_Zero (L : List_T; Pos : out Cursor; Success : out Boolean) with
  SPARK_Mode,
  Post => Success = (for some E of L => E = 0) and then
          (if Success then Element (L, Pos) = 0)
is
   Cu : Cursor := First (L);
begin
   while Has_Element (L, Cu) loop
      pragma Loop_Invariant (for all I in 1 .. P.Get (Positions (L), Cu) - 1 =>
                                Element (Model (L), I) /= 0);
      if Element (L, Cu) = 0 then
```

(continues on next page)

(continued from previous page)

```
         Success := True;
         Pos := Cu;
         return;
      end if;
      Next (L, Cu);
   end loop;

   Success := False;
   Pos := No_Element;
end Search_List_Zero;
```

The loop invariant expresses that all elements up to the current cursor `Cu` have a non-zero value. With this loop invariant, GNATprove is able to prove the postcondition of `Search_List_Zero`, namely that output parameter `Success` is True if-and-only-if there is an element of the list that has value zero, and that `Pos` is the cursor of such an element:

```
search_list_zero.adb:4:41: info: initialization of "Pos" proved
search_list_zero.adb:4:59: info: initialization of "Success" proved
search_list_zero.adb:6:11: info: postcondition proved
search_list_zero.adb:7:28: info: precondition proved
search_list_zero.adb:12:30: info: loop invariant initialization proved
search_list_zero.adb:12:30: info: loop invariant preservation proved
search_list_zero.adb:12:49: info: precondition proved
search_list_zero.adb:13:32: info: precondition proved
search_list_zero.adb:14:10: info: precondition proved
search_list_zero.adb:19:07: info: precondition proved
```

The case of sets and maps is similar to the case of lists.

Consider a variant of the same search loop over a pointer-based list:

```
with Loop_Types; use Loop_Types;

package P with
  SPARK_Mode
is
   function Is_Non_Zero (X : Component_T) return Boolean is
     (X /= 0);

   function Search_List_Zero (L : access List_Cell) return access List_Cell with
     Post =>
       ((Search_List_Zero'Result = null) = For_All_List (L, Is_Non_Zero'Access)
        and then
            (if Search_List_Zero'Result /= null then Search_List_Zero'Result.Value = 0));
end P;
```

```
with Loop_Types; use Loop_Types;

package body P with
  SPARK_Mode
is
   function Search_List_Zero (L : access List_Cell) return access List_Cell is
      B : access List_Cell := L;
   begin
```

(continues on next page)

(continued from previous page)

```
      while B /= null and then B.Value /= 0 loop
         pragma Loop_Variant (Structural => B);
         pragma Loop_Invariant
           (For_All_List (L, Is_Non_Zero'Access) =
                For_All_List (B, Is_Non_Zero'Access));
         B := B.Next;
      end loop;

      return B;
   end Search_List_Zero;
end P;
```

As our pointer-based lists do not support cursors, the result of the search is a pointer inside the list which can be used to access or even update the corresponding element. Storing such an object inside an OUT parameter would break the ownership model of SPARK by creating an alias. Instead, we use a traversal function (see *Traversal Functions*) to return this pointer as a local borrower of the input list. Since we now have a function, we can no longer have an explicit `Success` flag to encode whether or not the value was found. Instead, we simply return `null` in case of failure.

The loop iterates over the input list `L` using a local borrower `B`. The iteration stops when either `B` is `null` or `B.Value` is zero. In the loop invariant, we cannot speak directly about the elements of `L` that have been traversed to say that they are not `0`. Instead, we write in the invariant that `L` contains only non-zero values iff `B` contains only non-zero values. Thanks to this loop invariant, GNATprove is able to verify the postcondition of `Search_List_Zero`:

```
p.adb:9:33: info: pointer dereference check proved
p.adb:10:10: info: loop variant proved
p.adb:12:13: info: loop invariant initialization proved
p.adb:12:13: info: loop invariant preservation proved
p.adb:12:41: info: null exclusion check proved
p.adb:13:45: info: null exclusion check proved
p.adb:14:16: info: pointer dereference check proved
p.adb:17:14: info: dynamic accessibility check proved
p.ads:6:13: info: implicit aspect Always_Terminates on "Is_Non_Zero" has been proved,␣
↪subprogram will terminate
p.ads:9:13: info: implicit aspect Always_Terminates on "Search_List_Zero" has been␣
↪proved, subprogram will terminate
p.ads:11:08: info: postcondition proved
p.ads:11:72: info: null exclusion check proved
p.ads:13:77: info: pointer dereference check proved
```

For more complex examples of search loops, see the *SPARK Tutorial* as well as the section on *How to Write Loop Invariants*.

Maximize Loops

This kind of loops iterates over a collection to search an element of the collection that maximizes a given optimality criterion:

Loop Pattern	Search Optimum to Criterion
Proof Objective	Find an element or position that maximizes an optimality criterion.
Loop Behavior	Loops over the collection. Records maximum value of criterion so far and possibly index that maximizes this criterion.
Loop Invariant	Exactly one element encountered so far corresponds to the recorded maximum over other elements encountered so far.

Consider a procedure `Search_Arr_Max` that searches an element maximum value in array `A`:

```
1  with Loop_Types; use Loop_Types;
2
3  procedure Search_Arr_Max (A : Arr_T; Pos : out Index_T; Max : out Component_T) with
4    SPARK_Mode,
5    Post => (for all J in A'Range => A(J) <= Max) and then
6            (for some J in A'Range => A(J) = Max) and then
7            A(Pos) = Max
8  is
9  begin
10    Max := 0;
11    Pos := A'First;
12
13    for J in A'Range loop
14       if A(J) > Max then
15          Max := A(J);
16          Pos := J;
17       end if;
18       pragma Loop_Invariant (for all K in A'First .. J => A(K) <= Max);
19       pragma Loop_Invariant (for some K in A'First .. J => A(K) = Max);
20       pragma Loop_Invariant (A(Pos) = Max);
21    end loop;
22 end Search_Arr_Max;
```

The loop invariant expresses that all elements up to the current loop index `J` have a value less than `Max`, and that `Max` is the value of one of these elements. The last loop invariant gives in fact this element, it is `A(Pos)`, but this part of the loop invariant may not be present if the position `Pos` for the optimum is not recorded. With this loop invariant, GNATprove is able to prove the postcondition of `Search_Arr_Max`, namely that output parameter `Max` is the maximum of the elements in the array, and that `Pos` is the index of such an element:

```
search_arr_max.adb:3:38: info: initialization of "Pos" proved
search_arr_max.adb:3:57: info: initialization of "Max" proved
search_arr_max.adb:5:11: info: postcondition proved
search_arr_max.adb:18:30: info: loop invariant initialization proved
search_arr_max.adb:18:30: info: loop invariant preservation proved
search_arr_max.adb:18:61: info: index check proved
```

(continues on next page)

(continued from previous page)

```
search_arr_max.adb:19:30: info: loop invariant preservation proved
search_arr_max.adb:19:30: info: loop invariant initialization proved
search_arr_max.adb:19:62: info: index check proved
search_arr_max.adb:20:30: info: loop invariant initialization proved
search_arr_max.adb:20:30: info: loop invariant preservation proved
```

Consider now a variant of the same search loop over a vector:

```
with Loop_Types; use Loop_Types; use Loop_Types.Vectors;

procedure Search_Vec_Max (V : Vec_T; Pos : out Index_T; Max : out Component_T) with
  SPARK_Mode,
  Pre  => not Is_Empty (V),
  Post => (for all J in First_Index (V) .. Last_Index (V) => Element (V, J) <= Max) and
↪then
          (for some J in First_Index (V) .. Last_Index (V) => Element (V, J) = Max) and
↪then
          Pos in First_Index (V) .. Last_Index (V) and then
          Element (V, Pos) = Max
is
begin
   Max := 0;
   Pos := First_Index (V);

   for J in First_Index (V) .. Last_Index (V) loop
      if Element (V, J) > Max then
         Max := Element (V, J);
         Pos := J;
      end if;
      pragma Loop_Invariant (for all K in First_Index (V) .. J => Element (V, K) <= Max);
      pragma Loop_Invariant (for some K in First_Index (V) .. J => Element (V, K) = Max);
      pragma Loop_Invariant (Pos in First_Index (V) .. J);
      pragma Loop_Invariant (Element (V, Pos) = Max);
   end loop;
end Search_Vec_Max;
```

Like before, the loop invariant expresses that all elements up to the current loop index `J` have a value less than `Max`, and that `Max` is the value of one of these elements, most precisely the value of `Element (V, Pos)` if the position `Pos` for the optimum is recorded. An additional loop invariant is needed here compared to the case of arrays to state that `Pos` remains within the bounds of the vector. With this loop invariant, GNATprove is able to prove the postcondition of `Search_Vec_Max`, namely that output parameter `Max` is the maximum of the elements in the vector, and that `Pos` is the index of such an element:

```
search_vec_max.adb:3:38: info: initialization of "Pos" proved
search_vec_max.adb:3:57: info: initialization of "Max" proved
search_vec_max.adb:6:11: info: postcondition proved
search_vec_max.adb:6:62: info: precondition proved
search_vec_max.adb:6:74: info: range check proved
search_vec_max.adb:7:63: info: precondition proved
search_vec_max.adb:7:75: info: range check proved
search_vec_max.adb:9:11: info: precondition proved
search_vec_max.adb:16:10: info: precondition proved
```

(continues on next page)

(continued from previous page)

```
search_vec_max.adb:17:17: info: precondition proved
search_vec_max.adb:20:30: info: loop invariant initialization proved
search_vec_max.adb:20:30: info: loop invariant preservation proved
search_vec_max.adb:20:67: info: precondition proved
search_vec_max.adb:20:79: info: range check proved
search_vec_max.adb:21:30: info: loop invariant initialization proved
search_vec_max.adb:21:30: info: loop invariant preservation proved
search_vec_max.adb:21:68: info: precondition proved
search_vec_max.adb:21:80: info: range check proved
search_vec_max.adb:22:30: info: loop invariant preservation proved
search_vec_max.adb:22:30: info: loop invariant initialization proved
search_vec_max.adb:23:30: info: precondition proved
search_vec_max.adb:23:30: info: loop invariant preservation proved
search_vec_max.adb:23:30: info: loop invariant initialization proved
```

Similarly, consider a variant of the same search loop over a list:

```
with Loop_Types; use Loop_Types; use Loop_Types.Lists;
with Ada.Containers; use Ada.Containers; use Loop_Types.Lists.Formal_Model;

procedure Search_List_Max (L : List_T; Pos : out Cursor; Max : out Component_T) with
  SPARK_Mode,
  Pre  => not Is_Empty (L),
  Post => (for all E of L => E <= Max) and then
          (for some E of L => E = Max) and then
          Has_Element (L, Pos) and then
          Element (L, Pos) = Max
is
   Cu : Cursor := First (L);
begin
   Max := 0;
   Pos := Cu;

   while Has_Element (L, Cu) loop
      pragma Loop_Invariant (for all I in 1 .. P.Get (Positions (L), Cu) - 1 =>
                               Element (Model (L), I) <= Max);
      pragma Loop_Invariant (Has_Element (L, Pos));
      pragma Loop_Invariant (Max = 0 or else Element (L, Pos) = Max);

      if Element (L, Cu) > Max then
         Max := Element (L, Cu);
         Pos := Cu;
      end if;
      Next (L, Cu);
   end loop;
end Search_List_Max;
```

The loop invariant expresses that all elements up to the current cursor `Cu` have a value less than `Max`, and that `Max` is the value of one of these elements, most precisely the value of `Element (L, Pos)` if the cursor `Pos` for the optimum is recorded. Like for vectors, an additional loop invariant is needed here compared to the case of arrays to state that cursor `Pos` is a valid cursor of the list. A minor difference is that a loop invariant now starts with `Max = 0 or else ..` because the loop invariant is stated at the start of the loop (for convenience with the use of `First_To_Previous`) which requires this modification. With this loop invariant, GNATprove is able to prove the

postcondition of `Search_List_Max`, namely that output parameter `Max` is the maximum of the elements in the list, and that `Pos` is the cursor of such an element:

```
search_list_max.adb:4:40: info: initialization of "Pos" proved
search_list_max.adb:4:58: info: initialization of "Max" proved
search_list_max.adb:7:11: info: postcondition proved
search_list_max.adb:10:11: info: precondition proved
search_list_max.adb:18:30: info: loop invariant preservation proved
search_list_max.adb:18:30: info: loop invariant initialization proved
search_list_max.adb:18:49: info: precondition proved
search_list_max.adb:19:32: info: precondition proved
search_list_max.adb:20:30: info: loop invariant initialization proved
search_list_max.adb:20:30: info: loop invariant preservation proved
search_list_max.adb:21:30: info: loop invariant initialization proved
search_list_max.adb:21:30: info: loop invariant preservation proved
search_list_max.adb:21:46: info: precondition proved
search_list_max.adb:23:10: info: precondition proved
search_list_max.adb:24:17: info: precondition proved
search_list_max.adb:27:07: info: precondition proved
```

The case of sets and maps is similar to the case of lists.

Consider a variant of the same search loop over a pointer-based list:

```
with Loop_Types; use Loop_Types;

package P with
  SPARK_Mode
is
   function All_Smaller_Than_Max
     (L : access constant List_Cell; Max : Component_T) return Boolean
   is (L = null or else
         (L.Value <= Max and then All_Smaller_Than_Max (L.Next, Max)))
   with
     Subprogram_Variant => (Structural => L);

   function Search_List_Max
     (L : not null access List_Cell) return not null access List_Cell
   with
     Post => All_Smaller_Than_Max (L, Search_List_Max'Result.Value);
end P;
```

```
with Loop_Types; use Loop_Types;
with SPARK.Big_Integers; use SPARK.Big_Integers;

package body P with
  SPARK_Mode
is
   function Search_List_Max
     (L : not null access List_Cell) return not null access List_Cell
   is
      B : access List_Cell := L;
   begin
      loop
```

(continues on next page)

(continued from previous page)

```
13          pragma Loop_Invariant (B /= null);
14          pragma Loop_Invariant
15            (for all M in B.Value .. Component_T'Last =>
16               (if All_Smaller_Than_Max (B, M)
17                then All_Smaller_Than_Max (L, M)));
18          pragma Loop_Variant (Decreases => Length (B));
19          declare
20             Prec : access List_Cell := B;
21             Max  : constant Component_T := B.Value;
22          begin
23             loop
24                B := B.Next;
25                exit when B = null or else B.Value > Max;
26                pragma Loop_Invariant (B /= null);
27                pragma Loop_Invariant (B.Value <= Max);
28                pragma Loop_Invariant (Length (B) < Length (B)'Loop_Entry);
29                pragma Loop_Invariant
30                  (for all M in Max .. Component_T'Last =>
31                     (if All_Smaller_Than_Max (B, M)
32                      then All_Smaller_Than_Max (L, M)));
33                pragma Loop_Variant (Decreases => Length (B));
34             end loop;
35             if B = null then
36                return Prec;
37             end if;
38          end;
39       end loop;
40    end Search_List_Max;
41 end P;
```

As our pointer-based lists do not support cursors, the result of the search is a pointer inside the list which can be used to access or even update the corresponding element. Storing such an object inside an OUT parameter would break the ownership model of SPARK by creating an alias. Instead, we use a traversal function (see *Traversal Functions*) to return this pointer as a local borrower of the input list. Since we now have a function, we can no longer explicitely return the value of the maximum. It is not a problem, as it can be accessed easily as the `Value` component of the returned pointer. In the postcondition of `Search_List_Max`, we cannot use `For_All_List` to express that the returned pointer designates the maximum value in the list. Indeed, the property depends on the value of this maximum. Instead, we create a specific recursive function taking the maximum as an additional parameter.

The iteration over the input list `L` uses a local borrower `B`. It is expressed as two nested loops. The inner loop declares a local borrower `Prec` to register the current value of the maximum. Then it iterates through the loop using `B` until a value bigger than the current maximum is found. The outer loop repeats this step as many times as necessary. This split into two loops is necessary as the SPARK language prevents borrowers from jumping into a different part of the data structure. As `B` is not syntactically a path rooted at `Prec`, `Prec` cannot be assigned the current value of `B` when a new maximal value is found. We therefore need to create a new variable to hold the current maximum each time it changes.

In the loop invariant of the outer loop, we cannot speak directly about the elements of `L` that have been traversed to say that they are smaller than the current maximum. Instead, we write in the invariant that the all values of `L` are smaller than any given value bigger than the current maximum iff the values of `B` are. A similar invariant is necessary on the inner loop. Thanks to these loop invariants, GNATprove is able to verify the postcondition of `Search_List_Max`:

```
p.adb:13:33: info: loop invariant preservation proved
```

(continues on next page)

(continued from previous page)

```
p.adb:13:33: info: loop invariant initialization proved
p.adb:15:13: info: loop invariant initialization proved
p.adb:15:13: info: loop invariant preservation proved
p.adb:15:27: info: pointer dereference check proved
p.adb:16:44: info: range check proved
p.adb:17:46: info: range check proved
p.adb:18:31: info: loop variant proved
p.adb:18:44: info: range check proved
p.adb:21:45: info: pointer dereference check proved
p.adb:24:22: info: pointer dereference check proved
p.adb:25:44: info: pointer dereference check proved
p.adb:26:39: info: loop invariant initialization proved
p.adb:26:39: info: loop invariant preservation proved
p.adb:27:39: info: loop invariant initialization proved
p.adb:27:39: info: loop invariant preservation proved
p.adb:27:40: info: pointer dereference check proved
p.adb:28:39: info: predicate check proved
p.adb:28:39: info: loop invariant initialization proved
p.adb:28:39: info: loop invariant preservation proved
p.adb:28:62: info: predicate check proved
p.adb:30:19: info: loop invariant initialization proved
p.adb:30:19: info: loop invariant preservation proved
p.adb:31:50: info: range check proved
p.adb:32:52: info: range check proved
p.adb:33:37: info: loop variant proved
p.adb:33:50: info: range check proved
p.adb:36:23: info: dynamic accessibility check proved
p.adb:36:23: info: null exclusion check proved
p.ads:6:13: info: implicit aspect Always_Terminates on "All_Smaller_Than_Max" has been␣
↪proved, subprogram will terminate
p.ads:9:12: info: pointer dereference check proved
p.ads:9:35: info: subprogram variant proved
p.ads:9:58: info: pointer dereference check proved
p.ads:13:13: info: implicit aspect Always_Terminates on "Search_List_Max" has been␣
↪proved, subprogram will terminate
p.ads:16:14: info: postcondition proved
p.ads:16:61: info: pointer dereference check proved
```

The loop variants state that the length of `B` is strictly decreasing. This is used to prove that the loop terminates.

For more complex examples of search loops, see the *SPARK Tutorial* as well as the section on *How to Write Loop Invariants*.

Update Loops

This kind of loops iterates over a collection to update individual elements based either on their value or on their position. The first pattern we consider is the one that updates elements based on their value:

Loop Pattern	Modification of Elements Based on Value
Proof Objective	Elements of the collection are updated based on their value.
Loop Behavior	Loops over a collection and assigns the elements whose value satisfies a given modification criterion.
Loop Invariant	Every element encountered so far has been assigned according to its value.

Consider a procedure `Update_Arr_Zero` that sets to zero all elements in array `A` that have a value smaller than a given `Threshold`:

```
with Loop_Types; use Loop_Types;

procedure Update_Arr_Zero (A : in out Arr_T; Threshold : Component_T) with
  SPARK_Mode,
  Post => (for all J in A'Range => A(J) = (if A'Old(J) <= Threshold then 0 else A
↪'Old(J)))
is
begin
   for J in A'Range loop
      if A(J) <= Threshold then
         A(J) := 0;
      end if;
      pragma Loop_Invariant (for all K in A'First .. J => A(K) = (if A'Loop_Entry(K) <=␣
↪Threshold then 0 else A'Loop_Entry(K)));
      --  The following loop invariant is generated automatically by GNATprove:
      --  pragma Loop_Invariant (for all K in J + 1 .. A'Last => A(K) = A'Loop_Entry(K));
   end loop;
end Update_Arr_Zero;
```

The loop invariant expresses that all elements up to the current loop index `J` have been zeroed out if initially smaller than `Threshold` (using *Attribute Loop_Entry*). With this loop invariant, GNATprove is able to prove the postcondition of `Update_Arr_Zero`, namely that all elements initially smaller than `Threshold` have been zeroed out, and that other elements have not been modified:

```
update_arr_zero.adb:5:11: info: postcondition proved
update_arr_zero.adb:12:30: info: loop invariant initialization proved
update_arr_zero.adb:12:30: info: loop invariant preservation proved
update_arr_zero.adb:12:61: info: index check proved
update_arr_zero.adb:12:83: info: index check proved
update_arr_zero.adb:12:124: info: index check proved
```

Note that the commented loop invariant expressing that other elements have not been modified is not needed, as it is an example of *Automatically Generated Loop Invariants*.

Consider now a variant of the same update loop over a vector:

```
pragma Unevaluated_Use_Of_Old (Allow);
with Loop_Types; use Loop_Types; use Loop_Types.Vectors;
use Loop_Types.Vectors.Formal_Model;

procedure Update_Vec_Zero (V : in out Vec_T; Threshold : Component_T) with
  SPARK_Mode,
  Post => Last_Index (V) = Last_Index (V)'Old
  and (for all I in 1 .. Last_Index (V) =>
            Element (V, I) =
             (if Element (Model (V)'Old, I) <= Threshold then 0
              else Element (Model (V)'Old, I)))
is
begin
   for J in First_Index (V) .. Last_Index (V) loop
      pragma Loop_Invariant (Last_Index (V) = Last_Index (V)'Loop_Entry);
      pragma Loop_Invariant
        (for all I in 1 .. J - 1 =>
             Element (V, I) =
              (if Element (Model (V)'Loop_Entry, I) <= Threshold then 0
               else Element (Model (V)'Loop_Entry, I)));
      pragma Loop_Invariant
        (for all I in J .. Last_Index (V) =>
             Element (V, I) = Element (Model (V)'Loop_Entry, I));
      if Element (V, J) <= Threshold then
         Replace_Element (V, J, 0);
      end if;
   end loop;
end Update_Vec_Zero;
```

Like for `Map_Vec_Incr`, we need to use the `Model` function over arrays to access elements of the vector before the loop as the vector type is limited. The loop invariant expresses that all elements up to the current loop index `J` have been zeroed out if initially smaller than `Threshold`, that elements that follow the current loop index have not been modified, and that the length of `V` is not modified (like in `Init_Vec_Zero`). With this loop invariant, GNATprove is able to prove the postcondition of `Update_Vec_Zero`:

```
update_vec_zero.adb:7:11: info: postcondition proved
update_vec_zero.adb:9:13: info: precondition proved
update_vec_zero.adb:9:25: info: range check proved
update_vec_zero.adb:10:18: info: precondition proved
update_vec_zero.adb:10:42: info: range check proved
update_vec_zero.adb:11:20: info: precondition proved
update_vec_zero.adb:11:44: info: range check proved
update_vec_zero.adb:15:30: info: loop invariant initialization proved
update_vec_zero.adb:15:30: info: loop invariant preservation proved
update_vec_zero.adb:17:10: info: loop invariant initialization proved
update_vec_zero.adb:17:10: info: loop invariant preservation proved
update_vec_zero.adb:17:30: info: overflow check proved
update_vec_zero.adb:18:14: info: precondition proved
update_vec_zero.adb:18:26: info: range check proved
update_vec_zero.adb:19:19: info: precondition proved
update_vec_zero.adb:19:50: info: range check proved
update_vec_zero.adb:20:21: info: precondition proved
update_vec_zero.adb:20:52: info: range check proved
```

(continues on next page)

(continued from previous page)

```
update_vec_zero.adb:22:10: info: loop invariant initialization proved
update_vec_zero.adb:22:10: info: loop invariant preservation proved
update_vec_zero.adb:23:14: info: precondition proved
update_vec_zero.adb:23:26: info: range check proved
update_vec_zero.adb:23:31: info: precondition proved
update_vec_zero.adb:23:62: info: range check proved
update_vec_zero.adb:24:10: info: precondition proved
update_vec_zero.adb:25:10: info: precondition proved
```

Similarly, consider a variant of the same update loop over a list:

```
with Loop_Types; use Loop_Types; use Loop_Types.Lists;
with Ada.Containers; use Ada.Containers; use Loop_Types.Lists.Formal_Model;

procedure Update_List_Zero (L : in out List_T; Threshold : Component_T) with
  SPARK_Mode,
  Post => Length (L) = Length (L)'Old
  and (for all I in 1 .. Length (L) =>
            Element (Model (L), I) =
             (if Element (Model (L'Old), I) <= Threshold then 0
              else Element (Model (L'Old), I)))
is
   Cu : Cursor := First (L);
begin
   while Has_Element (L, Cu) loop
      pragma Loop_Invariant (Length (L) = Length (L)'Loop_Entry);
      pragma Loop_Invariant
        (for all I in 1 .. P.Get (Positions (L), Cu) - 1 =>
            Element (Model (L), I) =
             (if Element (Model (L'Loop_Entry), I) <= Threshold then 0
              else Element (Model (L'Loop_Entry), I)));
      pragma Loop_Invariant
        (for all I in P.Get (Positions (L), Cu) .. Length (L) =>
            Element (Model (L), I) = Element (Model (L'Loop_Entry), I));
      if Element (L, Cu) <= Threshold then
         Replace_Element (L, Cu, 0);
      end if;
      Next (L, Cu);
   end loop;
end Update_List_Zero;
```

The loop invariant expresses that all elements up to the current cursor `Cu` have been zeroed out if initially smaller than `Threshold` (using function `Model` to access the element stored at a given position in the list and function `Positions` to query the position of the current cursor), and that elements that follow the current loop index have not been modified. Note that it is necessary to state here that the length of the list is not modified during the loop. It is because the length is used to bound the quantification over the elements of the list both in the invariant and in the postcondition.

With this loop invariant, GNATprove is able to prove the postcondition of `Update_List_Zero`, namely that all elements initially smaller than `Threshold` have been zeroed out, and that other elements have not been modified:

```
update_list_zero.adb:6:11: info: postcondition proved
update_list_zero.adb:8:13: info: precondition proved
update_list_zero.adb:9:18: info: precondition proved
```

(continues on next page)

(continued from previous page)

```
update_list_zero.adb:10:20: info: precondition proved
update_list_zero.adb:15:30: info: loop invariant initialization proved
update_list_zero.adb:15:30: info: loop invariant preservation proved
update_list_zero.adb:17:10: info: loop invariant preservation proved
update_list_zero.adb:17:10: info: loop invariant initialization proved
update_list_zero.adb:17:29: info: precondition proved
update_list_zero.adb:18:13: info: precondition proved
update_list_zero.adb:19:18: info: precondition proved
update_list_zero.adb:20:20: info: precondition proved
update_list_zero.adb:22:10: info: loop invariant preservation proved
update_list_zero.adb:22:10: info: loop invariant initialization proved
update_list_zero.adb:22:24: info: precondition proved
update_list_zero.adb:23:13: info: precondition proved
update_list_zero.adb:23:33: info: range check proved
update_list_zero.adb:23:38: info: precondition proved
update_list_zero.adb:23:69: info: range check proved
update_list_zero.adb:24:10: info: precondition proved
update_list_zero.adb:25:10: info: precondition proved
update_list_zero.adb:27:07: info: precondition proved
```

The case of sets and maps is similar to the case of lists.

Consider now a variant of the same update loop over a pointer-based list. To express the postcondition relating the elements of the structure before and after the loop, we need to introduce a way to store the values of the list before the call in a separate data structure. Indeed, the `Old` attribute cannot be used on `L` directly has it would introduce an alias. In this example, it is done by declaring a `Copy` function which returns a copy of its input list. In its postcondition, we use the two-valued `For_All_List` function to state that the elements of the new structure are equal to the elements of its input structure. An alternative could be to store the elements in a structure not subject to ownership like an array.

Note: The function `Copy` is marked as `Import` as it is not meant to be executed. It could be implemented in SPARK by returning a deep copy of the argument list, reallocating all cells of the list in the result.

```
1  with Loop_Types; use Loop_Types;
2
3  package P with
4    SPARK_Mode
5  is
6     function Equal (X, Y : Component_T) return Boolean is (X = Y);
7
8     function Copy (L : access List_Cell) return List_Acc with
9       Ghost,
10      Import,
11      Global   => null,
12      Post     => For_All_List (L, Copy'Result, Equal'Access);
13
14    function Updated_If_Less_Than_Threshold
15      (L1, L2    : access constant List_Cell;
16       Threshold : Component_T) return Boolean
17    is
18      ((L1 = null) = (L2 = null)
19       and then
```

(continues on next page)

(continued from previous page)

```
        (if L1 /= null then
             (if L1.Value <= Threshold then L2.Value = 0
              else L2.Value = L1.Value)
         and then Updated_If_Less_Than_Threshold (L1.Next, L2.Next, Threshold)))
   with
     Subprogram_Variant => (Structural => L1);

   procedure Update_List_Zero (L : access List_Cell; Threshold : Component_T) with
     Post => Updated_If_Less_Than_Threshold (Copy (L)'Old, L, Threshold);
   pragma Annotate (GNATprove, Intentional, "memory leak might occur",
                    "The code will be compiled with assertions disabled");
end P;
```

```
with Loop_Types; use Loop_Types;

package body P with
  SPARK_Mode
is
   procedure Update_List_Zero (L : access List_Cell; Threshold : Component_T) is
      L_Old : constant List_Acc := Copy (L) with Ghost;
      pragma Annotate (GNATprove, Intentional, "memory leak might occur",
                       "The code will be compiled with assertions disabled");
      B     : access List_Cell := L;
      B_Old : access constant List_Cell := L_Old with Ghost;
   begin
      while B /= null loop
         pragma Loop_Invariant (For_All_List (B, B_Old, Equal'Access));
         pragma Loop_Invariant
           (if Updated_If_Less_Than_Threshold (B_Old, At_End (B), Threshold)
            then Updated_If_Less_Than_Threshold (L_Old, At_End (L), Threshold));
         if B.Value <= Threshold then
            B.Value := 0;
         end if;
         B := B.Next;
         B_Old := B_Old.Next;
      end loop;
      pragma Assert
        (Updated_If_Less_Than_Threshold (L_Old, At_End (L), Threshold));
   end Update_List_Zero;
end P;
```

In the postcondition of `Update_List_Zero`, we cannot use `For_All_List` to express the relation between the values of the list before and after the call. Indeed, the relation depends on the value of the input `Threshold`. Instead, we create a specific recursive function taking the threshold as an additional parameter.

The loop traverses `L` using a local borrower `B`. To be able to speak about the initial value of `L` in the invariant, we introduce a ghost constant `L_Old` storing a copy of this value. As we need to traverse both lists at the same time, we declare a ghost variable `B_Old` as a local observer of `L_Old`.

The loop invariant is made of two parts. The first one states that the elements reachable through `B` have not been modified by the loop. In the second loop invariant, we want to use `Updated_If_Less_Than_Threshold` to relate the elements of `L` that were already traversed to the elements of `L_Old`. As we cannot speak specifically about the traversed elements of `L`, the invariant states that, if at the end of the borrow the values accessible through `B` are related

to their equivalent element in `B_Old` through `Updated_If_Less_Than_Threshold`, then so are all the elements of `L`. GNATprove can verify these invariants as well as the postcondition of `Update_List_Zero`:

```
p.adb:7:07: info: absence of resource or memory leak at end of scope justified
p.adb:14:33: info: loop invariant initialization proved
p.adb:14:33: info: loop invariant preservation proved
p.adb:14:62: info: null exclusion check proved
p.adb:16:13: info: loop invariant initialization proved
p.adb:16:13: info: loop invariant preservation proved
p.adb:18:14: info: pointer dereference check proved
p.adb:19:14: info: pointer dereference check proved
p.adb:21:16: info: pointer dereference check proved
p.adb:22:24: info: pointer dereference check proved
p.adb:25:10: info: assertion proved
p.ads:6:13: info: implicit aspect Always_Terminates on "Equal" has been proved,␣
↪subprogram will terminate
p.ads:14:13: info: implicit aspect Always_Terminates on "Updated_If_Less_Than_Threshold"␣
↪has been proved, subprogram will terminate
p.ads:21:20: info: pointer dereference check proved
p.ads:21:47: info: pointer dereference check proved
p.ads:22:22: info: pointer dereference check proved
p.ads:22:33: info: pointer dereference check proved
p.ads:23:19: info: subprogram variant proved
p.ads:23:53: info: pointer dereference check proved
p.ads:23:62: info: pointer dereference check proved
p.ads:28:14: info: postcondition proved
p.ads:28:46: info: absence of resource or memory leak justified
```

The second pattern of update loops that we consider now is the one that updates elements based on their position:

Loop Pattern	Modification of Elements Based on Position
Proof Objective	Elements of the collection are updated based on their position.
Loop Behavior	Loops over a collection and assigns the elements whose position satisfies a given modification criterion.
Loop Invariant	Every element encountered so far has been assigned according to its position.

Consider a procedure `Update_Range_Arr_Zero` that sets to zero all elements in array `A` between indexes `First` and `Last`:

```
1 with Loop_Types; use Loop_Types;
2
3 procedure Update_Range_Arr_Zero (A : in out Arr_T; First, Last : Index_T) with
4   SPARK_Mode,
5   Post => A = (A'Old with delta First .. Last => 0)
6 is
7 begin
8    for J in First .. Last loop
9       A(J) := 0;
```

(continues on next page)

(continued from previous page)

```
      pragma Loop_Invariant (A = (A'Loop_Entry with delta First .. J => 0));
   end loop;
end Update_Range_Arr_Zero;
```

The loop invariant expresses that all elements between `First` and the current loop index `J` have been zeroed out, and that other elements have not been modified (using a combination of *Attribute Loop_Entry* and *Delta Aggregates* to express this concisely). With this loop invariant, GNATprove is able to prove the postcondition of `Update_Range_Arr_Zero`, namely that all elements between `First` and `Last` have been zeroed out, and that other elements have not been modified:

```
update_range_arr_zero.adb:5:11: info: postcondition proved
update_range_arr_zero.adb:10:30: info: loop invariant initialization proved
update_range_arr_zero.adb:10:30: info: loop invariant preservation proved
```

Consider now a variant of the same update loop over a vector:

```
pragma Unevaluated_Use_Of_Old (Allow);
with Loop_Types; use Loop_Types; use Loop_Types.Vectors;
use Loop_Types.Vectors.Formal_Model;

procedure Update_Range_Vec_Zero (V : in out Vec_T; First, Last : Index_T) with
  SPARK_Mode,
  Pre  => Last <= Last_Index (V),
  Post => (for all J in 1 .. Last_Index (V) =>
              (if J in First .. Last then Element (V, J) = 0
               else Element (V, J) = Element (Model (V)'Old, J)))
is
begin
   for J in First .. Last loop
      Replace_Element (V, J, 0);
      pragma Loop_Invariant (Last_Index (V) = Last_Index (V)'Loop_Entry);
      pragma Loop_Invariant
        (for all I in 1 .. Last_Index (V) =>
              (if I in First .. J then Element (V, I) = 0
               else Element (V, I) = Element (Model (V)'Loop_Entry, I)));
   end loop;
end Update_Range_Vec_Zero;
```

Like for `Map_Vec_Incr`, we need to use the `Model` function over arrays to access elements of the vector before the loop as the vector type is limited. The loop invariant expresses that all elements between `First` and current loop index `J` have been zeroed, and that other elements have not been modified. With this loop invariant, GNATprove is able to prove the postcondition of `Update_Range_Vec_Zero`:

```
update_range_vec_zero.adb:8:11: info: postcondition proved
update_range_vec_zero.adb:9:44: info: precondition proved
update_range_vec_zero.adb:9:56: info: range check proved
update_range_vec_zero.adb:10:22: info: precondition proved
update_range_vec_zero.adb:10:34: info: range check proved
update_range_vec_zero.adb:10:39: info: precondition proved
update_range_vec_zero.adb:10:63: info: range check proved
update_range_vec_zero.adb:14:07: info: precondition proved
update_range_vec_zero.adb:15:30: info: loop invariant preservation proved
update_range_vec_zero.adb:15:30: info: loop invariant initialization proved
update_range_vec_zero.adb:17:10: info: loop invariant preservation proved
```

(continues on next page)

(continued from previous page)

```
update_range_vec_zero.adb:17:10: info: loop invariant initialization proved
update_range_vec_zero.adb:18:41: info: precondition proved
update_range_vec_zero.adb:18:53: info: range check proved
update_range_vec_zero.adb:19:22: info: precondition proved
update_range_vec_zero.adb:19:34: info: range check proved
update_range_vec_zero.adb:19:39: info: precondition proved
update_range_vec_zero.adb:19:70: info: range check proved
```

Similarly, consider a variant of the same update loop over a list:

```
with Loop_Types; use Loop_Types; use Loop_Types.Lists;
with Ada.Containers; use Ada.Containers; use Loop_Types.Lists.Formal_Model;

procedure Update_Range_List_Zero (L : in out List_T; First, Last : Cursor) with
  SPARK_Mode,
  Pre  => Has_Element (L, First) and then Has_Element (L, Last)
  and then P.Get (Positions (L), First) <= P.Get (Positions (L), Last),
  Post => Length (L) = Length (L)'Old
  and Positions (L) = Positions (L)'Old
  and (for all I in 1 .. Length (L) =>
            (if I in P.Get (Positions (L), First) .. P.Get (Positions (L), Last) then
               Element (Model (L), I) = 0
             else Element (Model (L), I) = Element (Model (L'Old), I)))
is
   Cu : Cursor := First;
begin
   loop
      pragma Loop_Invariant (Has_Element (L, Cu));
      pragma Loop_Invariant (P.Get (Positions (L), Cu) in P.Get (Positions (L), First) ..
↪ P.Get (Positions (L), Last));
      pragma Loop_Invariant (Length (L) = Length (L)'Loop_Entry);
      pragma Loop_Invariant (Positions (L) = Positions (L)'Loop_Entry);
      pragma Loop_Invariant (for all I in 1 .. Length (L) =>
                               (if I in P.Get (Positions (L), First) .. P.Get (Positions␣
↪(L), Cu) - 1 then
                                  Element (Model (L), I) = 0
                                else Element (Model (L), I) = Element (Model (L'Loop_
↪Entry), I)));
      Replace_Element (L, Cu, 0);
      exit when Cu = Last;
      Next (L, Cu);
   end loop;
end Update_Range_List_Zero;
```

Compared to the vector example, it requires three additional invariants. As the loop is done via a cursor, the first two loop invariants are necessary to know that the current cursor `Cu` stays between `First` and `Last` in the list. The fourth loop invariant states that the position of cursors in `L` is not modified during the loop. It is necessary to know that the two cursors `First` and `Last` keep designating the same range after the loop. With this loop invariant, GNATprove is able to prove the postcondition of `Update_Range_List_Zero`, namely that all elements between `First` and `Last` have been zeroed out, and that other elements have not been modified:

```
update_range_list_zero.adb:7:13: info: precondition proved
```

(continues on next page)

(continued from previous page)

```
update_range_list_zero.adb:7:45: info: precondition proved
update_range_list_zero.adb:8:11: info: postcondition proved
update_range_list_zero.adb:11:23: info: precondition proved
update_range_list_zero.adb:11:55: info: precondition proved
update_range_list_zero.adb:12:16: info: precondition proved
update_range_list_zero.adb:13:19: info: precondition proved
update_range_list_zero.adb:13:44: info: precondition proved
update_range_list_zero.adb:18:30: info: loop invariant initialization proved
update_range_list_zero.adb:18:30: info: loop invariant preservation proved
update_range_list_zero.adb:19:30: info: loop invariant preservation proved
update_range_list_zero.adb:19:30: info: loop invariant initialization proved
update_range_list_zero.adb:19:31: info: precondition proved
update_range_list_zero.adb:19:60: info: precondition proved
update_range_list_zero.adb:19:92: info: precondition proved
update_range_list_zero.adb:20:30: info: loop invariant preservation proved
update_range_list_zero.adb:20:30: info: loop invariant initialization proved
update_range_list_zero.adb:21:30: info: loop invariant preservation proved
update_range_list_zero.adb:21:30: info: loop invariant initialization proved
update_range_list_zero.adb:22:30: info: loop invariant preservation proved
update_range_list_zero.adb:22:30: info: loop invariant initialization proved
update_range_list_zero.adb:23:42: info: precondition proved
update_range_list_zero.adb:23:74: info: precondition proved
update_range_list_zero.adb:24:36: info: precondition proved
update_range_list_zero.adb:25:38: info: precondition proved
update_range_list_zero.adb:25:63: info: precondition proved
update_range_list_zero.adb:26:07: info: precondition proved
update_range_list_zero.adb:28:07: info: precondition proved
```

7.9.3 Manual Proof Examples

The examples in this section contain properties that are difficult to prove automatically and thus require more user interaction to prove completely. The degre of interaction required depends on the difficuly of the proof:

- simple addition of calls to ghost lemmas for arithmetic properties involving multiplication, division and modulo operations, as decribed in *Manual Proof Using SPARK Lemma Library*
- more involved addition of ghost code for universally or existentially quantified properties on data structures and containers, as described in *Manual Proof Using Ghost Code*
- interaction at the level of Verification Condition formulas in the syntax of an interactive prover for arbitrary complex properties, as described in *Manual Proof Using Coq*

Manual Proof Using SPARK Lemma Library

If the property to prove is part of the *SPARK Lemma Library*, then manual proof simply consists in calling the appropriate lemma in your code. For example, consider the following assertion to prove, where `X1`, `X2` and `Y` may be signed or modular positive integers:

```
R1 := X1 / Y;
R2 := X2 / Y;
pragma Assert (R1 <= R2);
```

The property here is the monotonicity of division on positive values. There is a corresponding lemma for both signed and modular integers, for both 32 bits and 64 bits integers:

- for signed 32 bits integers, use `SPARK.Integer_Arithmetic_Lemmas.Lemma_Div_Is_Monotonic`
- for signed 64 bits integers, use `SPARK.Long_Integer_Arithmetic_Lemmas.Lemma_Div_Is_Monotonic`
- for modular 32 bits integers, use `SPARK.Mod32_Arithmetic_Lemmas.Lemma_Div_Is_Monotonic`
- for modular 64 bits integers, use `SPARK.Mod64_Arithmetic_Lemmas.Lemma_Div_Is_Monotonic`

For example, the lemma for signed integers has the following signature:

```
procedure Lemma_Div_Is_Monotonic
  (Val1  : Int;
   Val2  : Int;
   Denom : Pos)
with
  Global => null,
  Pre  => Val1 <= Val2,
  Post => Val1 / Denom <= Val2 / Denom;
```

Assuming the appropriate library unit is with'ed and used in your code (see *SPARK Lemma Library* for details), using the lemma is simply a call to the ghost procedure `Lemma_Div_Is_Monotonic`:

```
R1 := X1 / Y;
R2 := X2 / Y;
Lemma_Div_Is_Monotonic (X1, X2, Y);
--  at this program point, the prover knows that R1 <= R2
--  the following assertion is proved automatically:
pragma Assert (R1 <= R2);
```

Note that the lemma may have a precondition, stating in which contexts the lemma holds, which you will need to prove when calling it. For example, a precondition check is generated in the code above to show that `X1 <= X2`. Similarly, the types of parameters in the lemma may restrict the contexts in which the lemma holds. For example, the type `Pos` for parameter `Denom` of `Lemma_Div_Is_Monotonic` is the type of positive integers. Hence, a range check may be generated in the code above to show that `Y` is positive.

To apply lemmas to signed or modular integers of different types than the ones used in the instances provided in the library, just convert the expressions passed in arguments, as follows:

```
R1 := X1 / Y;
R2 := X2 / Y;
Lemma_Div_Is_Monotonic (Integer(X1), Integer(X2), Integer(Y));
--  at this program point, the prover knows that R1 <= R2
--  the following assertion is proved automatically:
pragma Assert (R1 <= R2);
```

Manual Proof Using User Lemmas

If the property to prove is not part of the *SPARK Lemma Library*, then a user can easily add it as a separate lemma in her program. For example, suppose you need to have a proof that a fix list of numbers are prime numbers. This can be expressed easily in a lemma as follows:

```
function Is_Prime (N : Positive) return Boolean is
  (for all J in Positive range 2 .. N - 1 => N mod J /= 0);

procedure Number_Is_Prime (N : Positive)
with
  Ghost,
  Global => null,
  Pre  => N in 15486209 | 15487001 | 15487469,
  Post => Is_Prime (N);
```

Using the lemma is simply a call to the ghost procedure `Number_Is_Prime`:

```
Number_Is_Prime (15486209);
--  at this program point, the prover knows that 15486209 is prime, so
--  the following assertion is proved automatically:
pragma Assert (Is_Prime (15486209));
```

Note that the lemma here has a precondition, which you will need to prove when calling it. For example, the following incorrect call to the lemma will be detected as a precondition check failure:

```
Number_Is_Prime (10);  --  check message issued here
```

Then, the lemma procedure can be either implemented as a null procedure, in which case GNATprove will issue a check message about the unproved postcondition, which can be justified (see *Justifying Check Messages*) or proved with Coq (see *Manual Proof Using Coq*):

```
procedure Number_Is_Prime (N : Positive) is null;
```

Or it can be implemented as a normal procedure body with a single assumption:

```
procedure Number_Is_Prime (N : Positive) is
begin
   pragma Assume (Is_Prime (N));
end Number_Is_Prime;
```

Or it can be implemented in some cases as a normal procedure body with ghost code to achieve fully automatic proof, see *Manual Proof Using Ghost Code*.

Manual Proof Using Ghost Code

Guiding automatic solvers by adding intermediate assertions is a commonly used technique. More generally, whole pieces of *Ghost Code* can be added to enhance automated reasoning.

Proving Existential Quantification

Existentially quantified properties are difficult to verify for automatic solvers. Indeed, it requires coming up with a concrete value for which the property holds and solvers are not good at guessing. As an example, consider the following program:

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

pragma Assume (A (A'First) = 0 and then A (A'Last) > 0);

pragma Assert
  (for some I in A'Range =>
     I < A'Last and then A (I) = 0 and then A (I + 1) > 0);
```

Here we assume that the first element of an array `A` of type `Nat_Array` is 0, whereas is last element is positive. In such a case, we are sure that there is an index `I` in the array such `A (I)` is 0 but not `A (I + 1)`. Indeed, we know that `A` starts with a non-empty sequence of zeros. The last element of this sequence has the expected property. However, automatic solvers are unable to prove such a property automatically because they cannot guess which index they should consider. To help them, we can define a ghost function returning a value for which the property holds, and call it from an assertion:

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

function Find_Pos (A : Nat_Array) return Positive with Ghost,
  Pre  => A (A'First) = 0 and then A (A'Last) > 0,
  Post => Find_Pos'Result in A'First .. A'Last - 1 and then
     A (Find_Pos'Result) = 0 and then A (Find_Pos'Result + 1) > 0;

pragma Assume (A (A'First) = 0 and then A (A'Last) > 0);
pragma Assert (Find_Pos (A) in A'Range);
pragma Assert
  (for some I in A'Range =>
     I < A'Last and then A (I) = 0 and then A (I + 1) > 0);
```

Automatic solvers are now able to discharge the proof.

Performing Induction

Another difficult point for automated solvers is proof by induction. Though some automatic solvers do have heuristics allowing them to perform the most simple inductive proofs, they generally are lost when the induction is less straightforward. For example, in the example below, we state that the array `A` is sorted in two different ways, first by saying that each element is bigger than the one just before, and then by saying that each element is bigger than all the ones before:

```
pragma Assume
  (for all I in A'Range =>
    (if I > A'First then A (I) > A (I - 1)));
pragma Assert
  (for all I in A'Range =>
    (for all J in A'Range => (if I > J then A (I) > A (J))));
```

The second assertion is provable from the first one by induction over the number of elements separating `I` and `J`, but automatic solvers are unable to verify this code. To help them, we can use a ghost loop. In the loop invariant, we say that the property holds for all indexes `I` and `J` separated by less than `K` elements:

```
procedure Prove_Sorted (A : Nat_Array) with Ghost is
begin
  for K in 0 .. A'Length loop
    pragma Loop_Invariant
      (for all I in A'Range => (for all J in A'Range =>
          (if I > J and then I - J <= K then A (I) > A (J))));
  end loop;
end Prove_Sorted;
```

GNATprove will verify that the invariant holds in two steps, first it will show that the property holds at the first iteration, and then that, if it holds at a given iteration, then it also holds at the next (see *Loop Invariants*). Both proofs are straightforward using the assumption.

Note that we have introduced a ghost subprogram above to contain the loop. This will allow the compiler to recognize that this loop is ghost, so that it can be entirely removed when assertions are disabled.

If `Prove_Sorted` is declared locally to the subprogram that we want to verify, it is not necessary to supply a contract for it, as local subprograms with no contracts are inlined (see *Contextual Analysis of Subprograms Without Contracts*). We can still choose to provide such a contract to turn `Prove_Sorted` into a lemma (see *Manual Proof Using User Lemmas*).

A Concrete Example: a Sort Algorithm

We show how to prove the correctness of a sorting procedure on arrays using ghost code. In particular, we want to show that the sorted array is a permutation of the input array. A common way to define permutations is to encode the number of occurrences of each element in the array as a multiset, constructed inductively over the size of its array parameter (but it is not the only one, see *Ghost Variables*). The *Functional Containers Library* of SPARK provides an implementation of multisets that we use here:

```
pragma SPARK_Mode (On);

with SPARK.Containers.Functional.Multisets;

package Nat_Multisets is new SPARK.Containers.Functional.Multisets (Natural, "=");
```

```
package Sort_Types with SPARK_Mode is
   subtype Index is Integer range 1 .. 100;
   type Nat_Array is array (Index range <>) of Natural;
end Sort_Types;
```

```
with SPARK.Big_Integers;   use SPARK.Big_Integers;
with Nat_Multisets;        use Nat_Multisets;
with Sort_Types;           use Sort_Types;

package Perm with SPARK_Mode, Ghost is
   use Nat_Multisets;

   function Occurrences (Values : Nat_Array; Lst : Integer) return Multiset is
     (if Lst < Values'First then Empty_Multiset
      else Add (Occurrences (Values, Lst - 1), Values (Lst)))
   with Subprogram_Variant => (Decreases => Lst),
     Pre => Lst <= Values'Last;
```

(continues on next page)

(continued from previous page)

```
13
14    function Occurrences (Values : Nat_Array) return Multiset is
15      (Occurrences (Values, Values'Last));
16
17    function Occ (Values : Nat_Array; N : Natural) return Big_Natural is
18      (Nb_Occurence (Occurrences (Values), N));
19
20    function Is_Perm (Left, Right : Nat_Array) return Boolean is
21      (Occurrences (Left) = Occurrences (Right));
22
23 end Perm;
```

The only property of the function `Occurrences` required to prove that swapping two elements of an array is in fact a permutation, is the way `Occurrences` is modified when updating a value of the array.

There is no native construction for axioms in SPARK. As a workaround, a ghost subprogram, named "lemma subprogram", can be introduced with the desired property as a postcondition. An instance of the axiom will then be available whenever the subprogram is called. Notice that an explicit call to the lemma subprogram with the proper arguments is required whenever an instance of the axiom is needed, like in manual proofs in an interactive theorem prover. Here is how a lemma subprogram can be defined for the desired property of `Occurrences`:

```
package Perm.Lemma_Subprograms with
  SPARK_Mode,
  Ghost,
  Always_Terminates
is

   function Is_Set (A : Nat_Array; I : Index; V : Natural; R : Nat_Array)
                    return Boolean
   is (R'First = A'First and then R'Last = A'Last
       and then R (I) = V
       and then (for all J in A'Range =>
                   (if I /= J then R (J) = A (J)))) with
     Pre  => I in A'Range;

   procedure Occ_Set (A : Nat_Array; I : Index; V : Natural; R : Nat_Array)
   with
     Global => null,
     Pre    => I in A'Range and then Is_Set (A, I, V, R),
     Post   =>
       (if V = A (I) then Occurrences (R) = Occurrences (A)
        else Occ (R, V) = Occ (A, V) + 1
          and then Occ (R, A (I)) = Occ (A, A (I)) - 1
          and then
            (for all E of Union (Occurrences (R), Occurrences (A)) =>
               (if E not in V | A (I) then Occ (R, E) = Occ (A, E))));

end Perm.Lemma_Subprograms;
```

This "axiom" can then be used to prove an implementation of the selection sort algorithm. The lemma subprogram needs to be explicitely called when needed:

```
with Nat_Multisets;          use Nat_Multisets;
```

(continues on next page)

(continued from previous page)

```
with Perm.Lemma_Subprograms; use Perm.Lemma_Subprograms;

package body Sort
   with SPARK_Mode
is

   ----------------------------------------------------------------------------

   procedure Swap (Values : in out Nat_Array;
                   X      : in     Positive;
                   Y      : in     Positive)
     with
       Pre  => (X in Values'Range and then
                Y in Values'Range and then
                  X /= Y),

       Post => Is_Perm (Values'Old, Values)
     and Values (X) = Values'Old (Y)
     and Values (Y) = Values'Old (X)
     and (for all Z in Values'Range =>
            (if Z /= X and Z /= Y then Values (Z) = Values'Old (Z)))
   is
      Temp : Integer;

      --  Ghost variables
      Init   : constant Nat_Array (Values'Range) := Values with Ghost;
      Interm : Nat_Array (Values'Range) with Ghost;

      --  Ghost procedure
      procedure Prove_Perm with Ghost,
        Pre  => X in Values'Range and then Y in Values'Range and then
        Is_Set (Init, X, Init (Y), Interm)
        and then Is_Set (Interm, Y, Init (X), Values),
        Post => Is_Perm (Init, Values)
      is
      begin
         Occ_Set (Init, X, Init (Y), Interm);
         Occ_Set (Interm, Y, Init (X), Values);
         pragma Assert
           (for all F of Union (Occurrences (Init), Occurrences (Values)) =>
                Occ (Values, F) = Occ (Init, F));
      end Prove_Perm;

   begin
      Temp       := Values (X);
      Values (X) := Values (Y);

      --  Ghost code
      pragma Assert (Is_Set (Init, X, Init (Y), Values));
      Interm := Values;

      Values (Y) := Temp;
```

(continues on next page)

(continued from previous page)

```
   --  Ghost code
   pragma Assert (Is_Set (Interm, Y, Init (X), Values));
   Prove_Perm;
end Swap;

-- Finds the index of the smallest element in the array
function Index_Of_Minimum (Values : in Nat_Array)
                           return Positive
  with
    Pre  => Values'Length > 0,
    Post => Index_Of_Minimum'Result in Values'Range and then
    (for all I in Values'Range =>
       Values (Index_Of_Minimum'Result) <= Values (I))
is
   Min : Positive;
begin
   Min := Values'First;
   for Index in Values'Range loop
      if Values (Index) < Values (Min) then
         Min := Index;
      end if;
      pragma Loop_Invariant
        (Min in Values'Range and then
           (for all I in Values'First .. Index =>
                Values (Min) <= Values (I)));
   end loop;
   return Min;
end Index_Of_Minimum;

procedure Selection_Sort (Values : in out Nat_Array) is
   Smallest : Positive;  -- Index of the smallest value in the unsorted part
begin
   if Values'Length = 0 then
      return;
   end if;

   for Current in Values'First .. Values'Last - 1 loop
      Smallest := Index_Of_Minimum (Values (Current .. Values'Last));

      if Smallest /= Current then
         Swap (Values => Values,
               X      => Current,
               Y      => Smallest);
      end if;

      pragma Loop_Invariant
        (for all I in Values'First .. Current =>
           (for all J in I + 1 .. Values'Last =>
                Values (I) <= Values (J)));
      pragma Loop_Invariant (Is_Perm (Values'Loop_Entry, Values));
   end loop;
```

(continues on next page)

(continued from previous page)

```
   end Selection_Sort;

end Sort;
```

```
with SPARK.Big_Integers;   use SPARK.Big_Integers;
with Sort_Types;           use Sort_Types;
with Perm;                 use Perm;

package Sort with SPARK_Mode is

   -- Sorts the elements in the array Values in ascending order
   procedure Selection_Sort (Values : in out Nat_Array)
     with
       Post => Is_Perm (Values'Old, Values) and then
     (if Values'Length > 0 then
        (for all I in Values'First .. Values'Last - 1 =>
             Values (I) <= Values (I + 1)));
end Sort;
```

The procedure `Selection_Sort` can be verified using GNATprove at level 2.

```
sort.adb:18:16: info: postcondition proved
sort.adb:19:18: info: index check proved
sort.adb:19:35: info: index check proved
sort.adb:20:18: info: index check proved
sort.adb:20:35: info: index check proved
sort.adb:22:48: info: index check proved
sort.adb:22:65: info: index check proved
sort.adb:24:07: info: initialization of "Temp" proved
sort.adb:27:07: info: range check proved
sort.adb:27:53: info: length check proved
sort.adb:28:07: info: initialization of "Interm" proved
sort.adb:28:07: info: range check proved
sort.adb:33:09: info: precondition proved
sort.adb:33:23: info: range check proved
sort.adb:33:32: info: index check proved
sort.adb:34:18: info: precondition proved
sort.adb:34:34: info: range check proved
sort.adb:34:43: info: index check proved
sort.adb:35:17: info: postcondition proved
sort.adb:38:10: info: precondition proved
sort.adb:38:25: info: range check proved
sort.adb:38:34: info: index check proved
sort.adb:39:10: info: precondition proved
sort.adb:39:27: info: range check proved
sort.adb:39:36: info: index check proved
sort.adb:41:13: info: assertion proved
sort.adb:42:17: info: predicate check proved
sort.adb:42:35: info: predicate check proved
sort.adb:46:29: info: index check proved
sort.adb:47:15: info: index check proved
```

(continues on next page)

(continued from previous page)

```
sort.adb:47:29: info: index check proved
sort.adb:50:22: info: precondition proved
sort.adb:50:22: info: assertion proved
sort.adb:50:36: info: range check proved
sort.adb:50:45: info: index check proved
sort.adb:51:14: info: length check proved
sort.adb:51:17: info: length check proved
sort.adb:53:15: info: index check proved
sort.adb:53:21: info: range check proved
sort.adb:56:22: info: precondition proved
sort.adb:56:22: info: assertion proved
sort.adb:56:38: info: range check proved
sort.adb:56:47: info: index check proved
sort.adb:57:07: info: precondition proved
sort.adb:61:13: info: implicit aspect Always_Terminates on "Index_Of_Minimum" has been␣
↪proved, subprogram will terminate
sort.adb:65:16: info: postcondition proved
sort.adb:67:35: info: index check proved
sort.adb:67:55: info: index check proved
sort.adb:69:07: info: initialization of "Min" proved
sort.adb:71:20: info: range check proved
sort.adb:73:38: info: index check proved
sort.adb:74:20: info: range check proved
sort.adb:77:13: info: loop invariant initialization proved
sort.adb:77:13: info: loop invariant preservation proved
sort.adb:79:28: info: index check proved
sort.adb:79:44: info: index check proved
sort.adb:85:07: info: initialization of "Smallest" proved
sort.adb:91:50: info: overflow check proved
sort.adb:92:22: info: precondition proved
sort.adb:92:40: info: range check proved
sort.adb:95:13: info: precondition proved
sort.adb:96:29: info: range check proved
sort.adb:101:13: info: loop invariant preservation proved
sort.adb:101:13: info: loop invariant initialization proved
sort.adb:102:31: info: overflow check proved
sort.adb:103:28: info: index check proved
sort.adb:103:42: info: index check proved
sort.adb:104:33: info: loop invariant initialization proved
sort.adb:104:33: info: loop invariant preservation proved
sort.ads:10:16: info: postcondition proved
sort.ads:12:51: info: overflow check proved
sort.ads:13:22: info: index check proved
sort.ads:13:38: info: overflow check proved
sort.ads:13:38: info: index check proved
```

To complete the verification of our selection sort, the only remaining issue is the correctness of the axiom for `Occurrences`. It can be discharged using its definition. Since this definition is recursive, the proof requires induction, which is not normally in the reach of an automated prover. For GNATprove to verify it, it must be implemented using recursive calls on itself to assert the induction hypothesis. Note that the proof of the lemma is then conditioned to the termination of the lemma functions, which can be verified by GNATprove using a *Subprogram Variant*.

```
package body Perm.Lemma_Subprograms with SPARK_Mode is

   procedure Occ_Eq (A, B : Nat_Array; LA, LB : Integer) with
     Subprogram_Variant => (Decreases => LA),
     Pre  => LA <= A'Last and then LB <= B'Last
     and then A (A'First .. LA) = B (B'First .. LB),
     Post => Occurrences (A, LA) = Occurrences (B, LB);

   procedure Occ_Eq (A, B : Nat_Array; LA, LB : Integer) is
   begin
      if LA < A'First then
         return;
      end if;

      Occ_Eq (A, B, LA - 1, LB - 1);
   end Occ_Eq;

   procedure Occ_Set_Rec
     (A : Nat_Array; I : Index; V : Natural; R : Nat_Array; L : Integer)
   with
     Subprogram_Variant => (Decreases => L),
     Pre  => I in A'Range and then Is_Set (A, I, V, R) and then L <= A'Last,
     Post =>
       (if V = A (I) or else I > L then Occurrences (R, L) = Occurrences (A, L)
        else Nb_Occurence (Occurrences (R, L), V) =
          Nb_Occurence (Occurrences (A, L), V) + 1
        and then Nb_Occurence (Occurrences (R, L), A (I)) =
          Nb_Occurence (Occurrences (A, L), A (I)) - 1
        and then
          (for all E of Occurrences (R, L) =>
             (if E not in V | A (I)
              then Nb_Occurence (Occurrences (R, L), E) =
                Nb_Occurence (Occurrences (A, L), E)))
        and then
          (for all E of Occurrences (A, L) =>
             (if E not in V | A (I)
              then Nb_Occurence (Occurrences (R, L), E) =
                Nb_Occurence (Occurrences (A, L), E))));

   procedure Occ_Set_Rec
     (A : Nat_Array; I : Index; V : Natural; R : Nat_Array; L : Integer)
   is
   begin
      if L < A'First then
         return;
      end if;

      if I = L then
         Occ_Eq (A, R, L - 1, L - 1);
      else
         Occ_Set_Rec (A, I, V, R, L - 1);
      end if;
   end Occ_Set_Rec;
```

(continues on next page)

(continued from previous page)

```
   procedure Occ_Set (A : Nat_Array; I : Index; V : Natural; R : Nat_Array)
   is
   begin
      Occ_Set_Rec (A, I, V, R, A'Last);
   end Occ_Set;

end Perm.Lemma_Subprograms;
```

GNATprove proves automatically all checks on the final program at level 2.

```
perm.ads:8:13: info: implicit aspect Always_Terminates on "Occurrences" has been proved,␣
↪subprogram will terminate
perm.ads:10:17: info: precondition proved
perm.ads:10:17: info: subprogram variant proved
perm.ads:10:42: info: overflow check proved
perm.ads:10:56: info: index check proved
perm.ads:14:13: info: implicit aspect Always_Terminates on "Occurrences" has been proved,
↪ subprogram will terminate
perm.ads:15:07: info: precondition proved
perm.ads:17:13: info: implicit aspect Always_Terminates on "Occ" has been proved,␣
↪subprogram will terminate
perm.ads:20:13: info: implicit aspect Always_Terminates on "Is_Perm" has been proved,␣
↪subprogram will terminate
perm-lemma_subprograms.adb:3:14: info: aspect Always_Terminates on "Occ_Eq" has been␣
↪proved, subprogram will terminate
perm-lemma_subprograms.adb:6:15: info: range check proved
perm-lemma_subprograms.adb:6:35: info: range check proved
perm-lemma_subprograms.adb:7:14: info: precondition proved
perm-lemma_subprograms.adb:7:14: info: postcondition proved
perm-lemma_subprograms.adb:7:36: info: precondition proved
perm-lemma_subprograms.adb:15:07: info: precondition proved
perm-lemma_subprograms.adb:15:07: info: subprogram variant proved
perm-lemma_subprograms.adb:15:24: info: overflow check proved
perm-lemma_subprograms.adb:15:32: info: overflow check proved
perm-lemma_subprograms.adb:18:14: info: aspect Always_Terminates on "Occ_Set_Rec" has␣
↪been proved, subprogram will terminate
perm-lemma_subprograms.adb:22:36: info: precondition proved
perm-lemma_subprograms.adb:24:08: info: postcondition proved
perm-lemma_subprograms.adb:24:19: info: index check proved
perm-lemma_subprograms.adb:24:41: info: precondition proved
perm-lemma_subprograms.adb:24:62: info: precondition proved
perm-lemma_subprograms.adb:25:14: info: predicate check proved
perm-lemma_subprograms.adb:25:28: info: precondition proved
perm-lemma_subprograms.adb:26:11: info: predicate check proved
perm-lemma_subprograms.adb:26:25: info: precondition proved
perm-lemma_subprograms.adb:26:50: info: predicate check proved
perm-lemma_subprograms.adb:27:18: info: predicate check proved
perm-lemma_subprograms.adb:27:32: info: precondition proved
perm-lemma_subprograms.adb:27:55: info: index check proved
perm-lemma_subprograms.adb:28:11: info: predicate check proved
perm-lemma_subprograms.adb:28:25: info: precondition proved
```

(continues on next page)

(continued from previous page)

```
perm-lemma_subprograms.adb:28:48: info: index check proved
perm-lemma_subprograms.adb:28:54: info: predicate check proved
perm-lemma_subprograms.adb:30:25: info: precondition proved
perm-lemma_subprograms.adb:31:34: info: index check proved
perm-lemma_subprograms.adb:32:20: info: predicate check proved
perm-lemma_subprograms.adb:32:34: info: precondition proved
perm-lemma_subprograms.adb:33:17: info: predicate check proved
perm-lemma_subprograms.adb:33:31: info: precondition proved
perm-lemma_subprograms.adb:35:25: info: precondition proved
perm-lemma_subprograms.adb:36:34: info: index check proved
perm-lemma_subprograms.adb:37:20: info: predicate check proved
perm-lemma_subprograms.adb:37:34: info: precondition proved
perm-lemma_subprograms.adb:38:17: info: predicate check proved
perm-lemma_subprograms.adb:38:31: info: precondition proved
perm-lemma_subprograms.adb:49:10: info: precondition proved
perm-lemma_subprograms.adb:49:26: info: overflow check proved
perm-lemma_subprograms.adb:49:33: info: overflow check proved
perm-lemma_subprograms.adb:51:10: info: precondition proved
perm-lemma_subprograms.adb:51:10: info: subprogram variant proved
perm-lemma_subprograms.adb:51:37: info: overflow check proved
perm-lemma_subprograms.adb:58:07: info: precondition proved
perm-lemma_subprograms.ads:7:13: info: implicit aspect Always_Terminates on "Is_Set" has␣
↪been proved, subprogram will terminate
perm-lemma_subprograms.ads:10:20: info: index check proved
perm-lemma_subprograms.ads:12:39: info: index check proved
perm-lemma_subprograms.ads:12:47: info: index check proved
perm-lemma_subprograms.ads:15:14: info: aspect Always_Terminates on "Occ_Set" has been␣
↪proved, subprogram will terminate
perm-lemma_subprograms.ads:17:06: info: data dependencies proved
perm-lemma_subprograms.ads:18:38: info: precondition proved
perm-lemma_subprograms.ads:20:08: info: postcondition proved
perm-lemma_subprograms.ads:20:19: info: index check proved
perm-lemma_subprograms.ads:21:14: info: predicate check proved
perm-lemma_subprograms.ads:21:27: info: predicate check proved
perm-lemma_subprograms.ads:21:40: info: predicate check proved
perm-lemma_subprograms.ads:22:20: info: predicate check proved
perm-lemma_subprograms.ads:22:31: info: index check proved
perm-lemma_subprograms.ads:22:37: info: predicate check proved
perm-lemma_subprograms.ads:22:48: info: index check proved
perm-lemma_subprograms.ads:22:54: info: predicate check proved
perm-lemma_subprograms.ads:25:36: info: index check proved
perm-lemma_subprograms.ads:25:44: info: predicate check proved
perm-lemma_subprograms.ads:25:57: info: predicate check proved
```

Manual Proof Using Coq

This section presents a simple example of how to prove interactively a check with an interactive prover like Coq when GNATprove fails to prove it automatically (for installation of Coq, see also: *Coq*). Here is a simple SPARK procedure:

```
procedure Nonlinear (X, Y, Z : Positive; R1, R2 : out Natural) with
  SPARK_Mode,
  Pre  => Y > Z,
  Post => R1 <= R2
is
begin
   R1 := X / Y;
   R2 := X / Z;
end Nonlinear;
```

When only the Alt-Ergo prover is used, GNATprove does not prove automatically the postcondition of the procedure, even when increasing the value of the timeout:

```
nonlinear.adb:4:11: medium: postcondition might fail
    4 |  Post => R1 <= R2
      |          ^~~~~~~~
```

This is expected, as the automatic prover Alt-Ergo has only a simple support for non-linear integer arithmetic. More generally, it is a known difficulty for all automatic provers, although, in the case above, using prover cvc5 is enough to prove automatically the postcondition of procedure `Nonlinear`. We will use this case to demonstrate the use of a manual prover, as an example of what can be done when automatic provers fail to prove a check. We will use Coq here.

The Coq input file associated to this postcondition can be produced by either selecting *SPARK → Prove Check* and specifying `Coq` as alternate prover in GNAT Studio or by executing on the command-line:

```
gnatprove -P <prj_file>.gpr --limit-line=nonlinear.adb:4:11:VC_POSTCONDITION
--prover=Coq
```

The generated file contains many definitions and axioms that can be used in the proof, in addition to the ones in Coq standard library. The property we want to prove is at the end of the file:

```
Theorem def'vc :
  forall (r1:Numbers.BinNums.Z) (r2:Numbers.BinNums.Z),
  dynamic_invariant1 x Init.Datatypes.true Init.Datatypes.false
  Init.Datatypes.true Init.Datatypes.true ->
  dynamic_invariant1 y Init.Datatypes.true Init.Datatypes.false
  Init.Datatypes.true Init.Datatypes.true ->
  dynamic_invariant1 z Init.Datatypes.true Init.Datatypes.false
  Init.Datatypes.true Init.Datatypes.true ->
  dynamic_invariant r1 Init.Datatypes.false Init.Datatypes.false
  Init.Datatypes.true Init.Datatypes.true ->
  dynamic_invariant r2 Init.Datatypes.false Init.Datatypes.false
  Init.Datatypes.true Init.Datatypes.true -> (z < y)%Z ->
  forall (r11:Numbers.BinNums.Z), (r11 = (ZArith.BinInt.Z.quot x y)) ->
  forall (r21:Numbers.BinNums.Z), (r21 = (ZArith.BinInt.Z.quot x z)) ->
  (r11 <= r21)%Z.
Proof.
intros r1 r2 h1 h2 h3 h4 h5 h6 r11 h7 r21 h8.

Qed.
```

From the `forall` to the first `.` we can see the expression of what must be proved, also called the goal. The proof starts right after the dot and ends with the `Qed` keyword. Proofs in Coq are done with the help of different tactics which will change the state of the current goal. The first tactic (automatically added) here is `intros`, which allows to "extract" variables and hypotheses from the current goal and add them to the current environment. Each parameter to the `intros` tactic is the name that the extracted element will have in the new environment. The `intros` tactic here puts all universally quantified variables and all hypotheses in the environment. The goal is reduced to a simple inequality, with all potentially useful information in the environment.

Here is the state of the proof as displayed in a suitable IDE for Coq:

```
1 subgoal
r1, r2 : int
h1 : dynamic_invariant1 x true false true true
h2 : dynamic_invariant1 y true false true true
h3 : dynamic_invariant1 z true false true true
h4 : dynamic_invariant r1 false false true true
h5 : dynamic_invariant r2 false false true true
h6 : (z < y)%Z
r11 : int
h7 : r11 = (x ÷ y)%Z
r21 : int
h8 : r21 = (x ÷ z)%Z
______________________________________(1/1)
(r11 <= r21)%Z
```

Some expresions are enclosed in `()%Z`, which means that they are dealing with relative integers. This is necessarily in order to use the operators (e.g. < or +) on relative integers instead of using the associated Coq function or to declare a relative integer constant (e.g. `0%Z`).

Next, we can use the `subst` tactic to automaticaly replace variables by terms to which they are equal (as stated by the hypotheses in the current environment) and clean the environment of replaced variables. Here, we can get rid of many variables at once with `subst.` (note the presence of the `.` at the end of each tactic). The new state is:

```
1 subgoal
r1, r2 : int
h1 : dynamic_invariant1 x true false true true
h2 : dynamic_invariant1 y true false true true
h3 : dynamic_invariant1 z true false true true
h4 : dynamic_invariant r1 false false true true
h5 : dynamic_invariant r2 false false true true
h6 : (z < y)%Z
______________________________________(1/1)
(x ÷ y <= x ÷ z)%Z
```

At this state, the hypotheses alone are not enough to prove the goal without proving properties about ÷ and < operators. It is necessary to use theorems from the Coq standard library. Coq provides a command `SearchAbout` to find theorems and definition concerning its argument. For instance, to find the theorems referring to the operator ÷, we use `SearchAbout Z.quot.`, where `Z.quot` is the underlying function for the ÷ operator. Among the theorems displayed, the conclusion (the rightmost term separated by -> operator) of one of them seems to match our current goal:

```
Z.quot_le_compat_l:
  forall p q r : int, (0 <= p)%Z -> (0 < q <= r)%Z -> (p ÷ r <= p ÷ q)%Z
```

The tactic `apply` allows the use of a theorem or an hypothesis on the current goal. Here we use: `apply Z.quot_le_compat_l.`. This tactic will try to match the different variables of the theorem with the terms present in the goal. If it succeeds, one subgoal per hypothesis in the theorem will be generated to verify that the terms matched with

the theorem variables satisfy the hypotheses on those variables required by the theorem. In this case, `p` is matched with `x`, `q` with `z` and `r` with `y` and the new state is:

```
2 subgoals
r1, r2 : int
h1 : dynamic_invariant1 x true false true true
h2 : dynamic_invariant1 y true false true true
h3 : dynamic_invariant1 z true false true true
h4 : dynamic_invariant r1 false false true true
h5 : dynamic_invariant r2 false false true true
h6 : (z < y)%Z
______________________________________(1/2)
(0 <= x)%Z
______________________________________(2/2)
(0 < z <= y)%Z
```

As expected, there are two subgoals, one per hypothesis of the theorem. Once the first subgoal is proved, the rest of the script will automatically apply to the second one. Now, if we look back at the SPARK code, `X` is of type `Positive` so `X` is greater than 0 and `dynamic_invariantN` (where N is a number) are predicates generated by SPARK to state the range of a value from a ranged subtype interpreted as a relative integer in Coq. Here, the predicate `dynamic_invariant1` provides the property needed to prove the first subgoal which is that "All elements of subtype positive have their integer interpretation in the range 1 .. (2^{31} - 1)". To be able to see the definition of `dynamic_invariant1` in `h1`, we can use the `unfold` tactic of Coq. We need to supply the name of the predicate to unfold: `unfold dynamic_invariant1 in h1`. After this unfolding, we get a new predicate `in_range1` that we can unfold too so that `h1` is now `true = true \/ (1 <= 2147483647)%Z -> (1 <= x <= 2147483647)%Z`.

We see now that the goal does not match exactly the hypothesis, because one is a comparison with 0, while the other is a comparison with 1. Transitivity on "lesser or equal" relation is needed to prove this goal, of course this is provided in Coq's standard library:

```
Lemma Z.le_trans : forall n m p:Z, (n <= m)%Z -> (m <= p)%Z -> (n <= p)%Z.
```

Since the lemma's conclusion contains only two variables while it uses three, using tactic `apply Z.le_trans.` will generate an error stating that Coq was not able to find a term for the variable `m`. In this case, `m` needs to be instantiated explicitly, here with the value 1: `apply Zle_trans with (m:= 1%Z).` There are two new subgoals, one to prove that `0 <= 1` and the other that `1 <= x`:

```
3 subgoals
r1, r2 : int
h1 : true = true \/ (1 <= 2147483647)%Z -> (1 <= x <= 2147483647)%Z
h2 : dynamic_invariant1 y true false true true
h3 : dynamic_invariant1 z true false true true
h4 : dynamic_invariant r1 false false true true
h5 : dynamic_invariant r2 false false true true
h6 : (z < y)%Z
______________________________________(1/3)
(0 <= 1)%Z
______________________________________(2/3)
(1 <= x)%Z
______________________________________(3/3)
(0 < z <= y)%Z
```

To prove that `0 <= 1`, the theorem `Lemma Z.le_0_1 : (0 <= 1)%Z.` is used. `apply Z.le_0_1` will not generate any new subgoals since it does not contain implications. Coq passes to the next subgoal:

```
2 subgoals
r1, r2 : int
h1 : true = true \/ (1 <= 2147483647)%Z -> (1 <= x <= 2147483647)%Z
h2 : dynamic_invariant1 y true false true true
h3 : dynamic_invariant1 z true false true true
h4 : dynamic_invariant r1 false false true true
h5 : dynamic_invariant r2 false false true true
h6 : (z < y)%Z
______________________________________(1/2)
(1 <= x)%Z
______________________________________(2/2)
(0 < z <= y)%Z
```

This goal is now adapted to the range information in hypothesis `h1`. It introduces a subgoal which is the disjunction in the hypothesis of `h1`. To prove this disjunction, we need to tell Coq which operand we want to prove. Here, both are obviously true. Let's choose the left one using the tactic `left.`. We are left with only the equality to prove:

```
2 subgoals
r1, r2 : int
h1 : true = true \/ (1 <= 2147483647)%Z -> (1 <= x <= 2147483647)%Z
h2 : dynamic_invariant1 y true false true true
h3 : dynamic_invariant1 z true false true true
h4 : dynamic_invariant r1 false false true true
h5 : dynamic_invariant r2 false false true true
h6 : (z < y)%Z
______________________________________(1/2)
true = true
______________________________________(2/2)
(0 < z <= y)%Z
```

This can be discharged using `apply eq_refl.` to apply the reflexivity axiom of equality. Now the subgoal 1 is fully proved, and all that remains is subgoal 2:

```
1 subgoal
r1, r2 : int
h1 : dynamic_invariant1 x true false true true
h2 : dynamic_invariant1 y true false true true
h3 : dynamic_invariant1 z true false true true
h4 : dynamic_invariant r1 false false true true
h5 : dynamic_invariant r2 false false true true
h6 : (z < y)%Z
______________________________________(1/1)
(0 < z <= y)%Z
```

Transitivity is needed again, as well as the definition of `dynamic_invariant1`. In the previous subgoal, every step was detailed in order to show how the tactic `apply` worked. Now, let's see that proof doesn't have to be this detailed. The first thing to do is to add the fact that `1 <= z` to the current environment: `unfold dynamic_invariant1, in_range1 in h3.` will add the range of `z` as an hypothesis in the environment:

```
1 subgoal
r1, r2 : int
h1 : dynamic_invariant1 x true false true true
h2 : dynamic_invariant1 y true false true true
```

(continues on next page)

(continued from previous page)

```
h3 : true = true \/ (1 <= 2147483647)%Z -> (1 <= z <= 2147483647)%Z
h4 : dynamic_invariant r1 false false true true
h5 : dynamic_invariant r2 false false true true
h6 : (z < y)%Z
______________________________________(1/1)
(0 < z <= y)%Z
```

At this point, the goal can be solved simply using the `intuition.` tactic. `intuition` is an automatic tactic of Coq implementing a decision procedure for some simple goals. It either solves the goal or, if it fails, it does not generate any subgoals. The benefit of the latter way is that there are less steps than with the previous subgoal for a more complicated goal (there are two inequalities in the second subgoal) and we do not have to find the different theorems we need to solve the goal without `intuition`.

Finally, here is the final version of the proof script for the postcondition:

```
Theorem def'vc :
  forall (r1:Numbers.BinNums.Z) (r2:Numbers.BinNums.Z),
  dynamic_invariant1 x Init.Datatypes.true Init.Datatypes.false
  Init.Datatypes.true Init.Datatypes.true ->
  dynamic_invariant1 y Init.Datatypes.true Init.Datatypes.false
  Init.Datatypes.true Init.Datatypes.true ->
  dynamic_invariant1 z Init.Datatypes.true Init.Datatypes.false
  Init.Datatypes.true Init.Datatypes.true ->
  dynamic_invariant r1 Init.Datatypes.false Init.Datatypes.false
  Init.Datatypes.true Init.Datatypes.true ->
  dynamic_invariant r2 Init.Datatypes.false Init.Datatypes.false
  Init.Datatypes.true Init.Datatypes.true -> (z < y)%Z ->
  forall (r11:Numbers.BinNums.Z), (r11 = (ZArith.BinInt.Z.quot x y)) ->
  forall (r21:Numbers.BinNums.Z), (r21 = (ZArith.BinInt.Z.quot x z)) ->
  (r11 <= r21)%Z.
Proof.
intros r1 r2 h1 h2 h3 h4 h5 h6 r11 h7 r21 h8.
subst.
apply Z.quot_le_compat_l.
  apply Z.le_trans with (m:=1%Z).
    (* 0 <= x *)
  - apply Z.le_0_1.
    (* 1 <= x *)
  - unfold dynamic_invariant1, in_range1 in h1.
    apply h1. left. apply eq_refl.
  (* 0 < z <= y *)
  - unfold dynamic_invariant1, in_range1 in h3.
    intuition.
Qed.
```

To check and save the proof:

```
gnatprove -P <prj_file>.gpr --limit-line=nonlinear.adb:4:11:VC_POSTCONDITION
--prover=Coq --report=all
```

Now running GNATprove on the project should confirm that all checks are proved:

```
nonlinear.adb:4:11: info: postcondition proved
nonlinear.adb:7:12: info: range check proved
```

(continues on next page)

(continued from previous page)

```
nonlinear.adb:7:12: info: division check proved
nonlinear.adb:8:12: info: range check proved
nonlinear.adb:8:12: info: division check proved
```

CHAPTER

EIGHT

APPLYING SPARK IN PRACTICE

SPARK tools offer different levels of analysis, which are relevant in different contexts. This section starts with a description of the five *Levels of Software Assurance* that can be achieved with SPARK. It continues with a description of the main *Objectives of Using SPARK*. This list gathers the most commonly found reasons for adopting SPARK in industrial projects, but it is not intended to be an exhaustive list.

Whatever the objective(s) of using SPARK, any project fits in one of four possible *Project Scenarios*:

- the *brown field* scenario: *Maintenance and Evolution of Existing Ada Software*
- the *green field* scenario: *New Developments in SPARK*
- the *migration* scenario: *Conversion of Existing SPARK Software to SPARK 2014*
- the *frozen* scenario: *Analysis of Frozen Ada Software*

The end of this section examines each of these scenarios in turn and describes how SPARK can be applied in each case.

8.1 Levels of Software Assurance

SPARK analysis can give strong guarantees that a program:

- does not read uninitialized data,
- accesses global data only as intended,
- does not contain concurrency errors (deadlocks and data races),
- does not contain run-time errors (e.g., division by zero or buffer overflow), except for `Storage_Error`, which is not covered by SPARK analysis (see also section *Dealing with Storage_Error* below)
- respects key integrity properties (e.g., interaction between components or global invariants),
- is a correct implementation of software requirements expressed as contracts.

SPARK can analyze either a complete program or *those parts that are marked as being subject to analysis*, but it can only be applied to code that follows *some restrictions designed to facilitate formal verification*. In particular, tasking is restricted to the Ravenscar or Jorvik profiles and use of pointers should follow a strict ownership policy aiming at preventing aliasing of allocated data. Pointers and tasking are both features that, if supported completely, make formal verification, as done by SPARK, infeasible, either because of limitations of state-of-the-art technology or because of the disproportionate effort required from users to apply formal verification in such situations. The large subset of Ada that is analyzed by SPARK is also called the SPARK language subset.

SPARK builds on the strengths of Ada to provide even more guarantees statically rather than dynamically. As summarized in the following table, Ada provides strict syntax and strong typing at compile time plus dynamic checking of run-time errors and program contracts. SPARK allows such checking to be performed statically. In addition, it enforces the use of a safer language subset and detects data flow errors statically.

	Ada	SPARK
Contract programming	dynamic	dynamic / static
Run-time errors	dynamic	dynamic / static
Data flow errors	–	static
Strong typing	static	static
Safer language subset	–	static
Strict clear syntax	static	static

The main benefit of formal program verification as performed by SPARK is that it allows verifying properties that are difficult or very costly to verify by other methods, such as testing or reviews. That difficulty may stem from the complexity of the software, the complexity of the requirements, and/or the unknown capabilities of attackers. Formal verification allows giving guarantees that some properties are always verified, however complex the context. The latest versions of international certification standards for avionics (DO-178C / ED-12C) and railway systems (CENELEC EN 50128:2011) have recognized these benefits by increasing the role that formal methods can play in the development and verification of critical software.

8.1.1 Levels of SPARK Use

The scope and level of SPARK analysis depend on the objectives being pursued by the adoption of SPARK. The scope of analysis may be the totality of a project, only some units, or only parts of units. The level of analysis may range from simple guarantees provided by flow analysis to complex properties being proved. These can be divided into five easily remembered levels:

1. *Stone level* - valid SPARK
2. *Bronze level* - initialization and correct data flow
3. *Silver level* - absence of run-time errors (AoRTE)
4. *Gold level* - proof of key integrity properties
5. *Platinum level* - full functional proof of requirements

Platinum level is defined here for completeness, but it is seldom applicable due to the high cost of achieving it. Each level builds on the previous one, so that the code subject to the Gold level should be a subset of the code subject to Silver level, which itself is a subset of the code subject to Bronze level, which is in general the same as the code subject to Stone level. We advise using:

- Stone level only as an intermediate level during adoption,
- Bronze level for as large a part of the code as possible,
- Silver level as the default target for critical software (subject to costs and limitations),
- Gold level only for a subset of the code subject to specific key integrity (safety/security) properties,
- Platinum level only for those parts of the code with the highest integrity (safety/security) constraints.

Our starting point is a program in Ada, which could be thought of as the Brick level: thanks to the use of Ada programming language, this level already provides some confidence: it is the highest level in The Three Little Pigs fable! And indeed languages with weaker semantics could be thought of as Straw and Sticks levels. However, the adoption of SPARK allows us to get stronger guarantees, should the wolf in the fable adopt more aggressive means of attack than simply blowing.

A pitfall when using tools for automating human tasks is to end up "pleasing the tools" rather than working around the tool limitations. Both flow analysis and proof, the two technologies used in SPARK, have known limitations. Users should refrain from changing the program for the benefit of only getting fewer messages from the tools. When

relevant, users should justify tool messages through appropriate pragmas. See the sections on *Suppressing Warnings* and *Justifying Check Messages* for more details.

GNATprove can be run at the different levels mentioned in this document, either through the Integrated Development Environments (IDE) *GNAT Studio*, *Visual Studio Code* or *Eclipse*, or *on the command line*. Use of the command-line interface at a given level is facilitated by convenient synonyms:

- use switch `--mode=stone` for Stone level (synonym of `--mode=check_all`)
- use switch `--mode=bronze` for Bronze level (synonym of `--mode=flow`)
- use switch `--mode=silver` for Silver level (synonym of `--mode=all`)
- use switch `--mode=gold` for Gold level (synonym of `--mode=all`)

Note that levels Silver and Gold are activated with the same switches. Indeed, the difference between these levels is not on how GNATprove is run, but on the objectives of verification. This is explained in the section on *Gold Level*. Platinum level is not given a separate switch value, as it would be the same.

Sections *Stone Level* to *Platinum Level* present the details of the five levels of software assurance. Each section consists in a short description of three key aspects of adopting SPARK at that level:

- *Benefits* - What is gained from adopting SPARK?
- *Impact on Process* - How should the process (i.e., the software life cycle development and verification activities) be adapted to use SPARK?
- *Costs and Limitations* - What are the main costs and limitations for adopting SPARK?

Additionally, the index of this document contains entries for all levels (from *Stone level* to *Platinum level*) which point to parts of the User's Guide relevant for reaching a specific level.

8.1.2 Stone Level - Valid SPARK

The goal of reaching this level is to identify as much code as possible as belonging to the SPARK subset. The user is responsible for identifying candidate SPARK code by applying the marker `SPARK_Mode` to flag SPARK code to GNATprove, which is responsible for checking that the code marked with `SPARK_Mode` is indeed valid SPARK code. Note that valid SPARK code may still be incorrect in many ways, such as raising run-time exceptions. Being valid merely means that the code respects the legality rules that define the SPARK subset in the SPARK Reference Manual (see http://docs.adacore.com/spark2014-docs/html/lrm/). The number of lines of SPARK code in a program can be computed (along with other metrics such as the total number of lines of code) by the metrics computation tool GNATmetric.

Benefits

The stricter SPARK rules are enforced on a (hopefully) large part of the program, which leads to higher quality and maintainability, as error-prone features such as side effects in regular functions are avoided, and others, such as pointers, are restricted to avoid common mistakes. Individual and peer review processes can be reduced on the SPARK parts of the program, since analysis automatically eliminates some categories of defects. The parts of the program that don't respect the SPARK rules are carefully isolated so they can be more thoroughly reviewed and tested.

Impact on Process

After the initial pass of applying the SPARK rules to the program, ongoing maintenance of SPARK code is similar to ongoing maintenance of Ada code, with a few additional rules, such as the need to avoid side effects in functions. These additional rules are checked automatically by running GNATprove on the modified program, which can be done either by the developer before committing changes or by an automatic system (continuous builder, regression testsuite, etc.)

Costs and Limitations

Pointer-heavy code needs to be rewritten to follow the ownership policy or to hide pointers from SPARK analysis, which may be difficult. The initial pass may require large, but shallow, rewrites in order to transform the code, for example to annotate functions with side effects with aspect `Side_Effects` and move calls to such functions to the right-hand side of assignments.

8.1.3 Bronze Level - Initialization and Correct Data Flow

The goal of reaching this level is to make sure that no uninitialized data can ever be read and, optionally, to prevent unintended access to global variables. This also ensures no possible interference between parameters and global variables; i.e., the same variable isn't passed multiple times to a subprogram, either as a parameter or global variable. Finally, it ensures that functions must return in the absence of run-time error.

Benefits

The SPARK code is guaranteed to be free from a number of defects: no reads of uninitialized variables, no possible interference between parameters and global variables, no unintended access to global variables, no infinite loop or recursion in functions.

When `Global` contracts are used to specify which global variables are read and/or written by subprograms, maintenance is facilitated by a clear documentation of intent. This is checked automatically by GNATprove, so that any mismatch between the implementation and the specification is reported.

Impact on Process

An initial pass is required where flow analysis is enabled and the resulting messages are resolved either by rewriting code or justifying any false alarms. Once this is complete, ongoing maintenance can preserve the same guarantees at a low cost. A few simple idioms can be used to avoid most false alarms, and the remaining false alarms can be easily justified.

Costs and Limitations

The guarantees offered at Bronze level do not extend to subprograms with the annotation `Skip_Flow_And_Proof`, which are only analyzed at Stone level. These guarantees also do not extend to code in the following constructs:

- branches ending in an error-signaling statement such as `pragma Assert (False)`, which flow analysis treats as dead code.
- exception handlers and any code that can be executed after catching an exception, which flow analysis could treat as dead code if no callee has a corresponding exceptional contract.

Analysis by proof at *Silver Level* and above is required in these two cases.

The property that no uninitialized data can be read can only be guaranteed when the following SPARK feature is not used, as its purpose is precisely to allow more complex initialization patterns that can only be analyzed by proof at *Silver Level* and above:

- relaxed initialization of types and variables using aspect `Relaxed_Initialization`.

The property that functions must return in the absence of run-time errors can only be guaranteed when the following SPARK features are not used, as their purpose is precisely to allow more complex termination conditions that can only be analyzed by proof at *Silver Level* and above:

- specification of termination with aspect `Always_Terminate` with a non-static expression;
- specification of subprogram variants with aspect `Subprogram_Variant`;
- specification of loop variants with pragma `Loop_Variant`;
- specification of exceptional contracts with aspect `Exceptional_Cases`.

The initial pass may require a substantial effort to deal with the false alarms, depending on the coding style adopted up to that point. The analysis may take a long time, up to an hour on large programs, but it is guaranteed to terminate. Flow analysis is, by construction, limited to local understanding of the code, with no knowledge of values (only code paths) and handling of composite variables is only through calls, rather than component by component, which may lead to false alarms.

8.1.4 Silver Level - Absence of Run-time Errors (AoRTE)

The goal of this level is to ensure that the program does not raise an unexpected exception at run time. Among other things, this guarantees that the control flow of the program cannot be circumvented by exploiting a buffer overflow, or integer overflow. This also ensures that the program cannot crash or behave erratically when compiled without support for run-time checking (compiler switch `-gnatp`) because of operations that would have triggered a run-time exception.

GNATprove can be used to prove the complete absence of possible run-time errors corresponding to the explicit raising of unexpected exceptions in the program, raising the exception `Constraint_Error` at run time, and failures of assertions (corresponding to raising exception `Assertion_Error` at run time).

A special kind of run-time error that can be proved at this level is the absence of exceptions from defensive code. This requires users to add subprogram preconditions (see section *Preconditions* for details) that correspond to the conditions checked in defensive code. For example, defensive code that checks the range of inputs is modeled by a precondition of the form `Input_X in Low_Bound .. High_Bound`. These conditions are then checked by GNATprove at each call.

Benefits

The SPARK code is guaranteed to be free from run-time errors (Absence of Run Time Errors - AoRTE) plus all the defects already detected at Bronze level: no reads of uninitialized variables, no possible interference between parameters and/or global variables, no unintended access to global variables, and no infinite loop or recursion in functions. These guarantees extend to code using features that require proof for ensuring correct initialization and termination, as described in the limitations for *Bronze Level*. Thus, the quality of the program can be guaranteed to achieve higher levels of integrity than would be possible in other programming languages.

All the messages about possible run-time errors can be carefully reviewed and justified (for example by relying on external system constraints such as the maximum time between resets) and these justifications can be later reviewed as part of quality inspections.

The proof of AoRTE can be used to compile the final executable without run-time exceptions (compiler switch `-gnatp`), which results in very efficient code comparable to what can be achieved in C or assembly.

The proof of AoRTE can be used to comply with the objectives of certification standards in various domains (DO-178B/C in avionics, EN 50128 in railway, IEC 61508 in many safety-related industries, ECSS-Q-ST-80C in space, IEC 60880 in nuclear, IEC 62304 in medical, ISO 26262 in automotive). To date, the use of SPARK has been qualified in an EN 50128 context. Qualification plans for DO-178 have been developed by AdaCore. Qualification material in any context can be developed by AdaCore as part of a contract.

Impact on Process

An initial pass is required where proof of AoRTE is applied to the program, and the resulting messages are resolved by either rewriting code or justifying any false alarms. Once this is complete, as for the Bronze level, ongoing maintenance can retain the same guarantees at reasonable cost. Using precise types and simple subprogram contracts (preconditions and postconditions) is sufficient to avoid most false alarms, and any remaining false alarms can be easily justified.

Special treatment is required for loops, which may need the addition of loop invariants to prove AoRTE inside and after the loop. See the relevant sections of the SPARK User's Guide for a description of the detailed process for adding loop contracts, as well as examples of common patterns of loops and their corresponding loop invariants.

Costs and Limitations

The guarantees offered at Silver level and above do not extend to subprograms with the annotations `Skip_Flow_And_Proof` or `Skip_Proof`, which are only analyzed at Stone or Bronze level respectively.

The initial pass may require a substantial effort to resolve all false alarms, depending on the coding style adopted previously. The analysis may take a long time, up to a few hours, on large programs but is guaranteed to terminate. Proof is, by construction, limited to local understanding of the code, which requires using sufficiently precise types of variables, and some preconditions and postconditions on subprograms to communicate relevant properties to their callers.

Even if a property is provable, automatic provers may nevertheless not be able to prove it, due to limitations of the heuristic techniques used in automatic provers. In practice, these limitations mostly show up on non-linear integer arithmetic (such as division and modulo) and floating-point arithmetic.

8.1.5 Gold Level - Proof of Key Integrity Properties

The goal of the Gold level is to ensure key integrity properties such as maintaining critical data invariants throughout execution and guaranteeing that transitions between states follow a specified safety automaton. Typically these properties derive from software requirements. Together with the Silver level, these goals ensure program integrity, that is, the program executes within safe boundaries: the control flow of the program is correctly programmed and cannot be circumvented through run-time errors and data cannot be corrupted.

SPARK has a number of useful features for specifying both data invariants and control flow constraints:

- Type predicates reflect properties that should always be true of any object of the type.
- Preconditions reflect properties that should always hold on subprogram entry.
- Postconditions reflect properties that should always hold on subprogram exit.

These features can be verified statically by running GNATprove in proof mode, similarly to what was done at the Silver level. At every point where a violation of the property may occur, GNATprove issues either an 'info' message, verifying that the property always holds, or a 'check' message about a possible violation. Of course, a benefit of proving properties is that they don't need to be tested, which can be used to reduce or completely eliminate unit testing.

These features can also be used to augment integration testing with dynamic verification of key integrity properties. To enable this additional verification during execution, you can use either the compilation switch `-gnata` (which enables verification of all invariants and contracts at run time) or `pragma Assertion_Policy` (which enables a subset of

the verification) either inside the code (so that it applies to the code that follows in the current unit) or in a pragma configuration file (so that it applies to the entire program).

Benefits

The SPARK code is guaranteed to respect key integrity properties as well as being free from all the defects already detected at the Bronze and Silver levels: no reads of uninitialized variables, no possible interference between parameters and global variables, no unintended access to global variables, no infinite loop or recursion in functions, and no run-time errors. This is a unique feature of SPARK that is not found in other programming languages. In particular, such guarantees may be used in a safety case to make reliability claims.

The effort in achieving this level of confidence based on proof is relatively low compared to the effort required to achieve the same level based on testing. Indeed, confidence based on testing has to rely on an extensive testing strategy. Certification standards define criteria for approaching comprehensive testing, such as Modified Condition / Decision Coverage (MC/DC), which are expensive to achieve. Some certification standards allow the use of proof as a replacement for certain forms of testing, in particular DO-178C in avionics, EN 50128 in railway and IEC 61508 for functional safety. Obtaining proofs, as done in SPARK, can thus be used as a cost-effective alternative to unit testing.

Impact on Process

In a high-DAL certification context where proof replaces testing and independence is required between certain development/verification activities, one person can define the architecture and low-level requirements (package specs) and another person can develop the corresponding bodies and use GNATprove for verification. Using a common syntax/semantics – Ada 2012 contracts – for both the specs/requirements and the code facilitates communication between the two activities and makes it easier for the same person(s) to play different roles at different times.

Depending on the complexity of the property being proven, it may be more or less costly to add the necessary contracts on types and subprograms and to achieve complete automatic proof by interacting with the tool. This typically requires some experience with the tool, which can be gained by training and practice. Thus not all developers should be tasked with developing such contracts and proofs, but instead a few developers should be designated for this task.

As with the proof of AoRTE at Silver level, special treatment is required for loops, such as the addition of loop invariants to prove properties inside and after the loop. Details are presented in the SPARK User's Guide, together with examples of loops and their corresponding loop invariants.

Costs and Limitations

The analysis may take a long time, up to a few hours, on large programs, but it is guaranteed to terminate. It may also take more or less time depending on the proof strategy adopted (as indicated by the switches passed to GNATprove). Proof is, by construction, limited to local understanding of the code, which requires using sufficiently precise types of variables and some preconditions and postconditions on subprograms to communicate relevant properties to their callers.

Even if a property is provable, automatic provers may fail to prove it due to limitations of the heuristic techniques they employ. In practice, these limitations are mostly visible on non-linear integer arithmetic (such as division and modulo) and on floating-point arithmetic.

Some properties might not be easily expressible in the form of data invariants and subprogram contracts, for example properties of execution traces or temporal properties. Other properties may require the use of non-intrusive instrumentation in the form of ghost code.

8.1.6 Platinum Level - Full Functional Correctness

Platinum level is achieved when contracts fully cover the functional requirements. Achieving the Platinum level is rare in itself, and usually done for small parts of an application.

Benefits

The SPARK code is guaranteed to correctly implement its specification, including being free from all the defects already detected at the Bronze, Silver and Gold levels. These strong guarantees can be used as arguments in a safety/security case for the overall software system, providing steps are taken for *Managing Assumptions*.

Impact on Process

The impact on process is mostly the same as for Gold level. When manual proof is used, which is very likely at this level, there is an associated activity to maintain these proofs as the code evolves. Typically a dedicated verification engineer with enough experience of formal program verification in SPARK should be tasked with this activity.

Costs and Limitations

These are the same as for Gold level, plus the cost of applying manual proof more systematically. Depending on the manual proof technique used and the complexity of the proof, this might be more or less costly initially and during maintenance:

- *Manual Proof Using SPARK Lemma Library* is the least costly of all, only requiring to use the right lemma from the library.
- *Manual Proof Using Ghost Code* is more costly, as it requires expertise and interactions with the tool to guide automatic provers.
- *Manual Proof Using Coq* is the most costly, as it require expertise in interactive proof as well as knowledge of the syntax of the Coq interactive prover.

While the use of manual proof allows to prove any provable property in principle, a balance needs to be found between the higher cost of manual proof techniques and the benefits they bring compared to testing or manual justification.

8.2 Objectives of Using SPARK

8.2.1 Safe Coding Standard for Critical Software

SPARK is a subset of Ada meant for formal verification, by excluding features that are difficult or impossible to analyze automatically. This means that SPARK can also be used as a coding standard to restrict the set of features used in critical software. As a safe coding standard checker, SPARK allows both to prevent the introduction of errors by excluding unsafe Ada features, and it facilitates their early detection with GNATprove's flow analysis.

Exclusion of Unsafe Ada Features

Once the simple task of *Identifying SPARK Code* has been completed, one can use GNATprove in `check` mode to verify that SPARK restrictions are respected in SPARK code. Here we list some of the most error-prone Ada features that are excluded from SPARK (see *Excluded Ada Features* for the complete list).

- All expressions, including function calls, are free of side-effects. Expressions with side-effects are problematic because they hide interactions that occur in the code, in the sense that a computation will not only produce a value but also modify some hidden state in the program. In the worst case, they may even introduce interferences between subexpressions of a common expression, which results in different executions depending on the order of evaluation of subexpressions chosen by the compiler.
- The use of access types and allocators is restricted to pool specific access types and subject to an ownership policy ensuring that a mutable memory cell has a single owner. In general, pointers can introduce aliasing, that is, they can allow the same object to be visible through different names at the same program point. This makes it difficult to reason about a program as modifying the object under one of the names will also modify the other names. What is more, access types come with their own load of common mistakes, like double frees and dangling pointers.
- SPARK also prevents dependencies on the elaboration order by ensuring that no package can write into variables declared in other packages during its elaboration. The use of controlled types is also forbidden as they lead to insertions of implicit calls by the compiler. Finally, backward goto statements are not permitted as they obfuscate the control flow.

Early Detection of Errors

GNATprove's flow analysis will find all the occurrences of the following errors:

- uses of uninitialized variables (see *Data Initialization Policy*)
- aliasing of parameters that can cause interferences, which are often not accounted for by programmers (see *Absence of Interferences*)

It will also warn systematically about the following suspicious behaviors:

- wrong parameter modes (can hurt readability and maintainability or even be the sign of a bug, for example if the programmer forgot to update a parameter, to read the value of an out parameter, or to use the initial value of a parameter)
- unused variables or statements (again, can hurt readability and maintainability or even be the sign of a bug)

8.2.2 Prove Absence of Run-Time Errors (AoRTE)

With Proof Only

GNATprove can be used to prove the complete absence of possible run-time errors corresponding to:

- all possible explicit raising of unexpected exceptions in the program,
- raising exception `Constraint_Error` at run time, and
- all possible failures of assertions corresponding to raising exception `Assert_Error` at run time.

AoRTE is important for ensuring safety in all possible operational conditions for safety-critical software (including boundary conditions, or abnormal conditions) or for ensuring availability of a service (absence of DOS attack that can crash the software).

When run-time checks are enabled during execution, Ada programs are not vulnerable to the kind of attacks like buffer overflows that plague programs in C and C++, which allow attackers to gain control over the system. But in the case

where run-time checks are disabled (in general for efficiency, but it could be for other reasons), proving their absence with GNATprove also prevents such attacks. This is specially important for ensuring security when some inputs may have been crafted by an attacker.

Few subprogram contracts (*Preconditions* and *Postconditions*) are needed in general to prove AoRTE, far fewer than for proving functional properties. Even fewer subprogram contracts are needed if types are suitably constrained with *Type Contracts*. Typically, 95% to 98% of run-time checks can be proved automatically, and the remaining checks can be either verified with manual provers or justified by manual analysis.

GNATprove supports this type of combination of results in the summary table of *The Analysis Results Summary File*. Multiple columns display the number of checks automatically verified, while the column *Justified* displays the number of checks manually justified. The column *Unproved* should be empty for all checks to be verified.

With a Combination of Proof and Test

It is not always possible to achieve 100% proof of AoRTE, for multiple reasons:

1. Formal verification is only applicable to the part of the program that is in SPARK. If the program includes parts in Ada that are not in SPARK, for example, then it is not possible to prove AoRTE on those parts.
2. Some run-time checks may not be proved automatically due to prover shortcomings (see *Investigating Prover Shortcomings* for details).
3. `Storage_Error` exceptions are not covered by SPARK analysis.
4. It may not be cost-effective to add the required contracts for proving AoRTE in a less critical part of the code, compared to using testing as a means of verification.

For all these reasons, it is important to be able to combine the results of formal verification and testing on different parts of a codebase. Formal verification works by making some assumptions, and these assumptions should be shown to hold even when formal verification and testing are combined. Certainly, formal verification cannot guarantee the same properties when part of a program is only tested, as when all of a program is proved. The goal then, when combining formal verification and testing, is to reach a level of confidence as good as the level reached by testing alone.

At the Level of Individual Run-Time Checks

One way to get confidence that unproved run-time checks cannot fail during execution is to exercise them during testing. Test coverage information allows guaranteeing a set of run-time checks have been executed successfully during a test run. This coverage information may be gathered from the execution of a unit testing campaign, an integration testing campaign, or the execution of a dedicated testsuite focussing on exercising the run-time checks (for example on boundary values or random ones).

This strategy is already applied in other static analysis tools, for example in the integration between the CodePeer static analyzer and the VectorCAST testing tool for Ada programs.

Between Proof and Integration Testing

Contracts can also be exercised dynamically during integration testing. In cases where unit testing is not required (either because proof has been applied to all subprograms, or because the verification context allows it), exercising contracts during integration testing can complement proof results, by giving the assurance that the actual compiled program behaves as expected.

This strategy has been applied at Capgemini Engineering on UK military projects submitted to Def Stan 00-56 certification: AoRTE was proved on all the code, and contracts were exercised during integration testing, which allowed to scrap unit testing.

Between Proof and Unit Testing

Contracts on subprograms provide a natural boundary for combining proof and test:

- If proof is used to demonstrate that a subprogram is free of run-time errors and respects its contract, this proof depends on these properties being respected at the call site:
  - the precondition of the subprogram;
  - all inputs (including global variables) of the subprogram contain valid data for their types;
  - the Anti-Aliasing rules in SPARK RM 6.4.2 are respected.

  This verification can be achieved by proving the caller too, or, in the case of the precondition, by checking it dynamically during unit testing of the caller.
- If proof is used to demonstrate that a subprogram is free of run-time errors and respects its contract, and this subprogram calls other subprograms, this proof depends on the postconditions of the called subprogram being respected at call sites. This verification can be achieved by proving the callees too, or by checking dynamically the postcondition of the called subprograms during their unit testing.

Thus, it is possible to combine freely subprograms that are proved and subprograms that are unit tested, provided subprogram contracts (*Preconditions* and *Postconditions*) are exercised during unit testing. This can be achieved by compiling the program with assertions for testing (for example with switch `-gnata` in GNAT), or by using GNATtest to create the test harness (see section 7.10.12 of GNAT User's Guide on *Testing with Contracts*).

When combining proof and test on individual subprograms, one should make sure that the assumptions made for proof are justified at the boundary between proved subprograms and tested subprograms (see section on *Managing Assumptions*). To help with this verification, special switches are defined in GNAT to add run-time checks that verify dynamically the assumptions made during proof:

- `-gnateA` adds checks that parameters are not aliased
- `-gnateV` adds checks that parameters are valid, including parameters of composite types (arrays, records)
- `-gnatVa` adds checks that objects are valid at more places than -gnateV, but only for scalar objects

This strategy is particularly well suited in the context of the DO-178C certification standard in avionics, which explicitly allows proof or test to be used as verification means on each module.

8.2.3 Prove Correct Integration Between Components

Correct Integration In New Developments

GNATprove can be used to prove correct integration between components, where a component could be a subprogram, a unit or a set of units. Indeed, even if components are verified individually (for example by proof or test or a combination thereof), their combination may still fail because of unforeseen interactions or design problems.

SPARK is ideally equipped to support such analysis, with its detailed *Subprogram Contracts*:

- With *Data Dependencies*, a user can specify exactly the input and output data of a subprogram, which goes a long way towards uncovering unforeseen interactions.
- With functional contracts (*Preconditions* and *Postconditions*), a user can specify precisely properties about the behavior of the subprogram that are relevant for component integration. In general, simple contracts are needed for component integration, which means that they are easy to write and to verify automatically. See section on *Writing Contracts for Program Integrity* for examples of such contracts.

When using data dependencies, GNATprove's flow analysis is sufficient to check correct integration between components. When using functional contracts, GNATprove's proof should also be applied.

In Replacement of Comments

It is good practice to specify properties of a subprogram that are important for integration in the comments that are attached to the subprogram declaration.

Comments can be advantageously replaced by contracts:

- Comments about the domain of the subprogram can be replaced by *Preconditions*.
- Comments about the effects of the subprogram can be replaced by *Postconditions* and *Data Dependencies*.
- Comments about the result of functions can be replaced by *Postconditions*.
- GNATprove can use the contracts to prove correct integration between components, as in new developments.

Contracts are less ambiguous than comments, and can be accompanied by (or interspersed with) higher level comments that need not be focused on the finer grain details of which variables must have which values, as these are already specified concisely and precisely in the contracts.

In Replacement of Defensive Coding

In existing Ada code that is migrated to SPARK, defensive coding is typically used to verify the correct integration between components: checks are made at the start of a subprogram that inputs (parameters and global variables) satisfy expected properties, and an exception is raised or the program halted if an unexpected situation is found.

Defensive code can be advantageously replaced by preconditions:

- The dynamic checks performed by defensive code at run time can be performed equally by preconditions, and they can be enabled at a much finer grain thanks to *Pragma Assertion_Policy*.
- GNATprove can use the preconditions to prove correct integration between components, as in new developments.

8.2.4 Prove Functional Correctness

Functional Correctness In New Developments

GNATprove can be used to prove functional correctness of an implementation against its specification. This strongest level of verification can be applied either to specific subprograms, or specific units, or the complete program. For those subprograms whose functional correctness is to be checked, the user should:

1. express the specification of the subprogram as a subprogram contract (see *Preconditions* and *Postconditions*);
2. use GNATprove to prove automatically that most checks (including contracts) always hold; and
3. address the remaining unproved checks with manual justifications or testing, as already discussed in the section on how to *Prove Absence of Run-Time Errors (AoRTE)*.

As more complex contracts are required in general, it is expected that achieving that strongest level of verification is also more costly than proving absence of run-time errors. Typically, SPARK features like *Quantified Expressions* and *Expression Functions* are needed to express the specification, and features like *Loop Invariants* are needed to achieve automatic proof. See section on *Writing Contracts for Functional Correctness* for examples of such contracts, and section on *How to Write Loop Invariants* for examples of the required loop invariants.

When the functional specification is expressed as a set of disjoint cases, the SPARK feature of *Contract Cases* can be used to increase readability and to provide an automatic means to verify that cases indeed define a partitioning of the possible operational contexts.

In Replacement of Unit Testing

In existing Ada code that is migrated to SPARK, unit testing is typically used to verify functional correctness: actual outputs obtained when calling the subprogram are compared to expected outputs for given inputs. A *test case* defines an expected behavior to verify; a *test procedure* implements a *test case* with specific given inputs and expected outputs.

Test cases can be used as a basis for functional contracts, as they define in general a behavior for a set of similar inputs. Thus, a set of test cases can be transformed into *Contract Cases*, where each case corresponds to a test case: the test input constraint becomes the guard of the corresponding case, while the test output constraint becomes the consequence of the corresponding case.

GNATprove can be used to prove this initial functional contract, as in new developments. Then, cases can be progressively generalized (by relaxing the conditions in the guards), or new cases added to the contract, until the full functional behavior of the subprogram is specified and proved.

8.2.5 Ensure Correct Behavior of Parameterized Software

In some domains (railway, space), it is common to develop software which depends on parameterization data, which changes from mission to mission. For example, the layout of railroads or the characteristics of the payload for a spacecraft are mission specific, but in general do not require developing completely new software for the mission. Instead, the software may either depend on data definition units which are subject to changes between missions, or the software may load at starting time (possibly during *elaboration* in Ada) the data which defines the characteristics of the mission. Then, the issue is that a verification performed on a specific version of the software (for a given parameterization) is not necessarily valid for all versions of the software. In general, this means that verification has to be performed again for each new version of the software, which can be costly.

SPARK provides a better solution to ensure correct behavior of the software for all possible parameterizations. It requires defining a getter function for every variable or constant in the program that represents an element of parameterization, and calling this getter function instead of reading the variable or constant directly. Because GNATprove performs an analysis based on contracts, all that is known at analysis time about the value returned by a getter function is what is available from its signature and contract. Typically, one may want to use *Scalar Ranges* or *Predicates* to constrain the return subtype of such getter functions, to reflect the operational constraints respected by all parameterizations.

This technique ensures that the results of applying GNATprove are valid not only for the version of the software analyzed, but for any other version that satisfies the same operational constraints. This is valid whatever the objective(s) pursued with the use of SPARK: *Prove Absence of Run-Time Errors (AoRTE)*, *Prove Correct Integration Between Components*, *Prove Functional Correctness*, etc.

It may be the case that changing constants into functions makes the code illegal because the constants were used in representation clauses that require static values. In that case, compilation switch `-gnatI` should be specified when analyzing the modified code with GNATprove, so that representation clauses are ignored. As representation clauses have no effect on GNATprove's analysis, and their validity is checked by GNAT when compiling the original code, the formal verification results are valid for the original code.

For constants of a non-scalar type (for example, constants of record or array type), an alternative way to obtain a similar result as the getter function is to define the constant as a deferred constant, whose initial declaration in the visible part of a package spec does not specify the value of the constant. Then, the private part of the package spec which defines the completion of the deferred constant must be marked `SPARK_Mode => Off`, so that clients of the package only see the visible constant declaration without value. In such a case, the analysis of client units with GNATprove is valid for all possible values of the constant.

8.2.6 Safe Optimization of Run-Time Checks

Enabling run-time checks in a program usually increases the running time by around 10%. This may not fit the timing schedule in some highly constrained applications. In some cases where a piece of code is called a large number of times (for example in a loop), enabling run-time checks on that piece of code may increase the running time by far more than 10%. Thus, it may be tempting to remove run-time checking in the complete program (with compilation switch `-gnatp`) or a selected piece of code (with pragma `Suppress`), for the purpose of decreasing running time. The problem with that approach is that the program is not protected anymore against programming mistakes (for safety) or attackers (for security).

GNATprove provides a better solution, by allowing users to prove the absence of all run-time errors (or run-time errors of a specific kind, for example overflow checks) in a piece of code, provided the assumptions on which their proof relies are respected. This includes in particular the fact that the precondition of the enclosing subprogram is respected. Then, all run-time checks (or run-time errors of a specific kind) can be suppressed in that piece of code using pragma `Suppress`, knowing that they will never fail at run time, provided the corresponding assumptions are checked. For example, this can be done for the precondition of the enclosing subprogram by using *Pragma Assertion_Policy*. For more details, see *Choosing Which Run-time Checking to Keep*. By replacing many checks with a few checks, we can decrease the running time of the application by doing safe and controlled optimization of run-time checks.

8.2.7 Address Data and Control Coupling

As defined in the avionics standard DO-178, data coupling is *"The dependence of a software component on data not exclusively under the control of that software component"* and control coupling is *"The manner or degree by which one software component influences the execution of another software component"*, where a software component could be a subprogram, a unit or a set of units.

Although analysis of data and control coupling are not performed at the same level of details in non-critical domains, knowledge of data and control coupling is important to assess impact of code changes. In particular, it may be critical for security that some secret data does not leak publicly, which can be rephrased as saying that only the specified data dependencies are allowed. SPARK is ideally equipped to support such analysis, with its detailed *Subprogram Contracts*:

- With *Data Dependencies*, a user can specify exactly the input and output data of a subprogram, which identifies the *"data not exclusively under the control of that software component"*:
  - When taking the subprogram as component, any variable in the data dependencies is in general not exclusively under the control of that software component.
  - When taking the unit (or sets of units) as component, any variable in the data dependencies that is not defined in the unit itself (or the set of units) is in general not exclusively under the control of that software component.
- With *Flow Dependencies*, a user can specify the nature of the *"dependence of a software component on data not exclusively under the control of that software component"*, by identifying how that data may influence specific outputs of a subprogram.
- With *Flow Dependencies*, a user can also specify how *"one software component influences the execution of another software component"*, by identifying the shared data potentially written by the subprogram.
- With functional contracts (*Preconditions* and *Postconditions*), a user can specify very precisely the behavior of the subprogram, which defines how it *"influences the execution of another software component"*. These contracts need not be complete, for example they could describe the precedence order rules for calling various subprograms.

When using data and flow dependencies, GNATprove's flow analysis is sufficient to check that the program implements its specifications. When using functional contracts, GNATprove's proof should also be applied.

8.2.8 Ensure Portability of Programs

Using SPARK enhances portability of programs by excluding language features that are known to cause portability problems, and by making it possible to obtain guarantees that specific portability problems cannot occur. In particular, analyses of SPARK code can prove the absence of run-time errors in the program, and that specified functional properties always hold.

Still, porting a SPARK program written for a given compiler and target to another compiler and/or target may require changes in the program. As SPARK is a subset of Ada, and because in general only some parts of a complete program are in SPARK, we need to consider first the issue of portability in the context of Ada, and then specialize it in the context of SPARK.

Note that we consider here portability in its strictest sense, whereby a program is portable if its observable behavior is exactly the same across a change of compiler and/or target. In the more common sense of the word, a program is portable if it can be reused without modification on a different target, or when changing compiler. That is consistent with the definition of portability in WikiPedia: "Portability in high-level computer programming is the usability of the same software in different environments". As an example of a difference between both interpretations, many algorithms which use trigonometry are portable in the more common sense, not in the strictest sense.

Portability of Ada Programs

Programs with errors cause additional portability issues not seen in programs without errors, which is why we consider them separately.

Portability of Programs Without Errors

The Ada Reference Manual defines precisely which features of the language depend on choices made by the compiler (see Ada RM 1.1.3 "Conformity of an Implementation with the Standard"):

- *Implementation defined behavior* - The set of possible behaviors is specified in the language, and the particular behavior chosen in a compiler should be documented. An example of implementation defined behavior is the size of predefined integer types (like `Integer`). All implementation defined behaviors are listed in Ada RM M.2, and GNAT documents its implementation for each of these points in section 7 "Implementation Defined Characteristics" of the GNAT Reference Manual.
- *Unspecified behavior* - The set of possible behaviors is specified in the language, but the particular behavior chosen in a compiler need not be documented. An example of unspecified behavior is the order of evaluation of arguments in a subprogram call.

Changes of compiler and/or target may lead to different implementation defined and unspecified behavior, which may or not have a visible effect. For example, changing the order of evaluation of arguments in a subprogram call only has a visible effect if the evaluation of arguments itself has some side-effects.

Section 18.4 "Implementation-dependent characteristics" of the GNAT Reference Manual gives some advice on how to address implementation defined behavior for portability.

A particular issue is that the Ada Reference Manual gives much implementation freedom to the compiler in the implementation of operations of fixed-point and floating-point types:

- The small of a fixed-point type is implementation defined (Ada RM 3.5.9(8/2)) unless specified explicitly.
- The base type of a fixed-point type is implementation defined (Ada RM 3.5.9(12-16)), which has an impact on possible overflows.
- The rounded result of an ordinary fixed-point multiplication or division is implementation defined (Ada RM G.2.3(10)).

- For some combinations of types of operands and results for fixed-point multiplication and division, the value of the result belongs to an implementation defined set of values (Ada RM G.2.3(5)).
- The semantics of operations on floating-point types is implementation defined (Ada RM G.2). It may or may not follow the IEEE 754 floating point standard.
- The precision of elementary functions (exponential and trigonometric functions) is implementation defined (Ada RM G.2.4).

Section 18.1 "Writing Portable Fixed-Point Declarations" of the GNAT Reference Manual gives some advice on how to reduce implementation defined behavior for fixed-point types. Use of IEEE 754 floating-point arithmetic can be enforced in GNAT by using the compilation switches "-msse2 -mfpmath=sse", as documented in section 6.3.1.6 "Floating Point Operations" of the GNAT User's Guide.

Note that a number of restrictions can be used to prevent some features leading to implementation defined or unspecified behavior:

- Restriction `No_Fixed_Point` forbids the use of fixed-point types.
- Restriction `No_Floating_Point` forbids the use of floating-point types.
- Restriction `No_Implementation_Aspect_Specifications` forbids the use of implementation defined aspects.
- Restriction `No_Implementation_Attributes` forbids the use of implementation defined attributes.
- Restriction `No_Implementation_Pragmas` forbids the use of implementation defined pragmas.

Note: SPARK defines a few constructs (aspects, pragmas and attributes) that are not defined in Ada. While GNAT supports these constructs, care should be exercised to use these constructs with other compilers, or older versions of GNAT. This issue is detailed in section *Portability Issues*.

Portability of Programs With Errors

In addition to the portability issues discussed so far, programs with errors cause specific portability issues related to whether errors are detected and how they are reported. The Ada Reference Manual distinguishes between four types of errors (see Ada RM 1.1.5 "Classification of Errors"):

- *Compile-time errors* - These errors make a program illegal, and should be detected by any Ada compiler. They do not cause any portability issue, as they must be fixed before compilation.
- *Run-time errors* - These errors are signaled by raising an exception at run time. They might be a cause of portability problems, as a change of compiler and/or target may lead to new run-time errors. For example, a new compiler may cause the program to use more stack space, leading to an exception `Storage_Error`, and a new target may change the size of standard integer types, leading to an exception `Constraint_Error`.
- *Bounded errors* - These errors need not be detected either at compiler time or at run time, but their effects should be bounded. For example, reading an uninitialized value may result in any value of the type to be used, or to `Program_Error` being raised. Like for run-time errors, they might be a cause of portability problems, as a change of compiler and/or target may lead to new bounded errors.
- *Erroneous execution* - For the remaining errors, a program exhibits erroneous execution, which means that the error need not be detected, and its effects are not bounded by the language rules. These errors might be a cause of portability problems.

Portability issues may arise in a number of cases related to errors:

- The original program has an error that is not detected (a run-time error, bounded error or erroneous execution). Changing the compiler and/or target causes the error to be detected (an exception is raised) or to trigger a different

behavior. Typically, reads of uninitialized data or illegal accesses to memory that are not detected in the original program may result in errors when changing the compiler and/or the target.

- The original program has no error, but changing the compiler and/or target causes an error to appear, which may or not be detected. Typically, uses of low-level constructs like `Unchecked_Conversion` which depend on the exact representation of values in bits may lead to errors when changing the compiler and/or the target. Some run-time errors like overflow errors or storage errors are also particularly sensitive to compiler and target changes.

To avoid portability issues, errors should be avoided by using suitable analyses and reviews in the context of the original and the new compiler and/or target. Whenever possible, these analyses and reviews should be automated by tools to guarantee that all possible errors of a given kind have been reported.

Benefits of Using SPARK for Portability

The *Language Restrictions* in SPARK favor portability by excluding problematic language features (see *Excluded Ada Features*):

- By excluding side-effects in expressions, SPARK programs cannot suffer from effects occurring in different orders depending on the order of evaluation of expressions chosen by the compiler.
- By excluding aliasing, the behavior of SPARK programs does not depend on the parameter passing mechanism (by copy or by reference) or the order of assignment to out and in-out parameters passed by copy after the call, which are both chosen by the compiler.
- By excluding controlled types, SPARK programs cannot suffer from the presence and ordering of effects taking place as part of the initialization, assignment and finalization of controlled objects, which depend on choices made by the compiler.

As permitted by the SPARK language rules (see section 1.4.1 "Further Details on Formal Verification" of the SPARK Reference Manual), GNATprove rejects with an error programs which may implicitly raise a `Program_Error` in parts of code that are in SPARK. For example, all static execution paths in a SPARK function should end with a return statement, a raise statement, or a `pragma Assert (False)`. GNATprove's analysis can be further used to ensure that dynamic executions can only end in a return.

GNATprove reduces portability issues related to the use of fixed-point and floating-point values:

- GNATprove supports a subset of fixed-point types and operations that ensures that the result of an operation always belongs to the *perfect result set* as defined in Ada RM G.2.3. Note that the perfect result set still contains in general two values (the two model fixed-point values above and below the perfect mathematical result), which means that two compilers may give two different results for multiplication and division. Users should thus avoid multiplication and division of fixed-point values for maximal portability. See *GNATprove Limitations*.
- GNATprove assumes IEEE 754 standard semantics for basic operations of floating-point types (addition, subtraction, multiplication, division). With GNAT, this is achieved by using compilation switches "-msse2 -mfpmath=sse". Users should still avoid elementary functions (exponential and trigonometric functions) for maximal portability. See *Semantics of Floating Point Operations*.

Additionally, GNATprove can detect all occurrences of specific portability issues in SPARK code (that is, parts of the program for which `SPARK_Mode=On` is specified, see section on *Identifying SPARK Code*) when run in specific modes (see *Effect of Mode on Output* for a description of the different modes):

- In all modes (including mode `check`), when switch `--pedantic` is set, GNATprove issues a warning for every arithmetic operation which could be re-ordered by the compiler, thus leading to a possible overflow with one compiler and not another. For example, arithmetic operation `A + B + C` can be interpreted as `(A + B) + C` by one compiler, and `A + (B + C)` (after re-ordering) by another compiler. Note that GNAT always uses the former version without re-ordering. See *Parenthesized Arithmetic Operations*.

- In modes `flow`, `prove` and `all`, GNATprove issues high check messages on possible parameter aliasing, when such an aliasing may lead to interferences. This includes all cases where the choice of parameter passing mechanism in a compiler (by copy or by reference) might influence the behavior of the subprogram. See *Absence of Interferences*.
- In modes `flow`, `prove` and `all`, GNATprove issues check messages on possible reads of uninitialized data. These messages should be reviewed with respect to the stricter *Data Initialization Policy* in SPARK rather than in Ada. Hence, it is possible when the program does not conform to the stricter SPARK rules to manually validate them, see section *Justifying Check Messages*.
- In modes `prove` and `all`, GNATprove issues check messages on all possible run-time errors corresponding to raising exception `Constraint_Error` at run time, all possible failures of assertions corresponding to raising exception `Assert_Error` at run time, and all possible explicit raising of unexpected exceptions in the program.

The analysis of GNATprove can take into account characteristics of the target (size and alignment of standard scalar types, endianness) by specifying a *Target Parameterization*.

How to Use SPARK for Portability

GNATprove's analysis may be used to enhance the portability of programs. Note that the guarantees provided by this analysis only hold for the source program. To ensure that these guarantees extend to the executable object code, one should independently provide assurance that the object code correctly implements the semantics of the source code.

Avoiding Non-Portable Features

As much as possible, uses of non-portable language features should be avoided, or at least isolated in specific parts of the program to facilitate analyses and reviews when changing the compiler and/or the target.

This includes in particular language features that deal with machine addresses, data representations, interfacing with assembler code, and similar issues (for example, language attribute `Size`). When changing the compiler and/or the target, the program logic should be carefully reviewed for possible dependences on the original compiler behavior and/or original target characteristics. See also the section 18.4.5 "Target-specific aspects" of the GNAT Reference Manual.

In particular, features that bypass the type system of Ada for reinterpreting values (`Unchecked_Conversion`) and memory locations (`Address` clause overlays, in which multiple objects are defined to share the same address, something that can also be achieved by sharing the same `Link_Name` or `External_Name`) have no impact on SPARK analysis, yet they may lead to portability issues.

By using the following restrictions (or a subset thereof), one can ensure that the corresponding non-portable features are not used in the program:

```
pragma No_Dependence (Ada.Unchecked_Conversion);
pragma No_Dependence (System.Machine_code);
```

Similarly, the program logic should be carefully reviewed for possible dependency on target characteristics (for example, the size of standard integer types). GNATprove's analysis may help here as it can take into account the characteristics of the target. Hence, proofs of functional properties with GNATprove ensure that these properties will always hold on the target.

In the specific case that the target is changing, it might be useful to run GNATprove's analysis on the program in `proof` mode, even if it cannot prove completely the absence of run-time errors and that the specified functional properties (if any) hold. Indeed, by running GNATprove twice, once with the original target and once with the new target, comparing the results obtained in both cases might point to parts of the code that are impacted by the change of target, which may require more detailed manual reviews.

Apart from non-portable language features and target characteristics, non-portability in SPARK may come from a small list of causes:

- Possible re-ordering of non-parenthesized arithmetic operations. These can be detected by running GNATprove (see *Benefits of Using SPARK for Portability*). Then, either these operations may not be re-ordered by the compiler (for example, GNAT ensures this property), or re-ordering may not lead to an intermediate overflow (for example, if the base type is large enough), or the user may introduce parentheses to prevent re-ordering.
- Possible aliasing between parameters (or parameters and global variables) of a call causing interferences. These can be detected by running GNATprove (see *Benefits of Using SPARK for Portability*). Then, either aliasing is not possible in reality, or aliasing may not cause different behaviors depending on the parameter passing mechanism chosen in the compiler, or the user may change the code to avoid aliasing. When SPARK subprograms are called from non-SPARK code (for example Ada or C code), manual reviews should be performed to ensure that these calls cannot introduce aliasing between parameters, or between parameters and global variables.
- Possible different choices of base type for user-defined integer types (contrary to derived types or subtypes, which inherit their base type from their parent type). GNATprove follows GNAT in choosing as base type the smallest multiple-words-size integer type that contains the type bounds (see *Base Type of User-Defined Integer Types* for more information).
- Issues related to errors. See section *Avoiding Errors to Enhance Portability*.
- Issues related to the use of fixed-point or floating-point operations. See section *Portability of Fixed-Point and Floating-Point Computations* below.

Avoiding Errors to Enhance Portability

Because errors in a program make portability particularly challenging (see *Portability of Programs With Errors*), it is important to ensure that a program is error-free for portability. GNATprove's analysis can help by ensuring that the SPARK parts of a program are free from broad kinds of errors:

- all possible reads of uninitialized data
- all possible explicit raise of unexpected exceptions in the program
- all possible run-time errors except raising exception `Storage_Error`, corresponding to raising exception `Program_Error`, `Constraint_Error` or `Tasking_Error` at run time
- all possible failures of assertions corresponding to raising exception `Assert_Error` at run time

When parts of the program are not in SPARK (for example, in Ada or C), the results of GNATprove's analysis depend on assumptions on the correct behavior of the non-SPARK code. For example, callers of a SPARK subprogram should only pass initialized input values, and non-SPARK subprograms called from SPARK code should respect their postcondition. See section *Managing Assumptions* for more details on assumptions.

In particular, when changing the target characteristics, GNATprove's analysis can be used to show that no possible overflow can occur as a result of changing the size of standard integer types.

GNATprove's analysis does not detect possible run-time errors corresponding to raising exception `Storage_Error` at run time, which should be independently assessed.

Portability of Fixed-Point and Floating-Point Computations

Portability issues related to the use of fixed-point or floating-point operations can be avoided altogether by ensuring that the program does not use fixed-point or floating-point values, using:

```
pragma Restrictions (No_Fixed_Point);
pragma Restrictions (No_Floating_Point);
```

When fixed-point values are used, the value of the small and size in bits for the type should be specified explicitly, as documented in section 18.1 "Writing Portable Fixed-Point Declarations" of the GNAT Reference Manual:

```
My_Small : constant := 2.0**(-15);
My_First : constant := -1.0;
My_Last  : constant := +1.0 - My_Small;

type F2 is delta My_Small range My_First .. My_Last;
for F2'Small use my_Small;
for F2'Size  use 16;
```

The program should also avoid multiplication and division of fixed-point values to ensure that the result of arithmetic operations is exactly defined.

When floating-point values are used, use of IEEE 754 standard semantics for basic operations of floating-point types (addition, subtraction, multiplication, division) should be enforced. With GNAT, this is achieved by using compilation switches "-msse2 -mfpmath=sse".

The program should also avoid elementary functions (exponential and trigonometric functions), which can be ensured with a restriction:

```
pragma No_Dependence (Ada.Numerics);
```

If elementary functions are used, subject to reviews for ensuring portability, GNATprove's proof results may depend on the fact that elementary functions can be modeled as mathematical functions of their inputs that always return the same result when taking the same values in arguments. GNAT compiler was modified to ensure this property (see https://blog.adacore.com/how-our-compiler-learnt-from-our-analyzers), which may not hold for other Ada compilers.

8.3 Project Scenarios

The workflow for using SPARK depends not only on the chosen *Objectives of Using SPARK*, but also on the context in which SPARK is used: Is it for a new development? Or an evolution of an existing codebase? Is the existing codebase in Ada or in a version of SPARK prior to SPARK 2014? We examine all these project scenarios in this section.

8.3.1 Maintenance and Evolution of Existing Ada Software

Although SPARK is a large subset of Ada, it contains a number of *Language Restrictions* which prevent in general direct application of GNATprove to an existing Ada codebase without any modifications. The suggested workflow is to:

1. Identify violations of SPARK restrictions.
2. For each violation, either rewrite the code in SPARK or mark it `SPARK_Mode => Off` (see section on *Identifying SPARK Code*).

3. Perform the required analyses to achieve the desired objectives (see section on *Formal Verification with GNATprove*), a process which likely involved writing contracts (see in particular section on *How to Write Subprogram Contracts*).
4. Make sure that the assumptions made for formal verification are justified at the boundary between SPARK and full Ada code (see section on *Managing Assumptions*).

Identifying Violations of SPARK Restrictions

A simple way to identify violations of SPARK restrictions is by *Setting the Default SPARK_Mode* to `SPARK_Mode => On`, and then running GNATprove either in `check` mode (to report basic violations) or in `flow` mode (to report violations whose detection requires flow analysis).

If only a subset of the project files should be analyzed, one should create a project file for *Specifying Files To Analyze* or *Excluding Files From Analysis*.

Finally, one may prefer to work her way through the project one unit at a time by *Using SPARK_Mode in Code*, and running GNATprove on the current unit only.

Rewriting the Code in SPARK

Depending on the violation, it may be more or less easy to rewrite the code in SPARK:

- Unsupported types should in general be rewritten as private types of a package whose public part is marked `SPARK_Mode => On` and whose private part is marked `SPARK_Mode => Off`. Thus, the body of that package cannot be analyzed by GNATprove, but clients of the package can be analyzed.
- Functions with side-effects should be rewritten as procedures, by adding an additional out parameter for the result of the function.
- Aliasing should be either explicitly signed off by *Justifying Check Messages* or removed by introducing a copy of the object to pass as argument to the call.
- Controlled types cannot be rewritten easily.

Using `SPARK_Mode` to Select or Exclude Code

Depending on the number and location of remaining violations, `SPARK_Mode` can be used in different ways:

- If most of the codebase is in SPARK, *Setting the Default SPARK_Mode* to `SPARK_Mode => On` is best. Violations should be isolated in parts of the code marked `SPARK_Mode => Off` by either *Excluding Selected Unit Bodies* or *Excluding Selected Parts of a Unit*.
- Otherwise, `SPARK_Mode => On` should be applied selectively for *Verifying Selected Subprograms* or *Verifying Selected Units*. Violations are allowed outside the parts of the code marked `SPARK_Mode => On`.
- Even when most of the code is in SPARK, it may be more cost effective to apply `SPARK_Mode => On` selectively rather than by default. This is the case in particular when some units have non-SPARK declarations in the public part of their package spec. Rewriting the code of these units to isolate the non-SPARK declarations in a part that can be marked `SPARK_Mode => Off` may be more costly than specifying no `SPARK_Mode` for these units, which allows SPARK code elsewhere in the program to refer to the SPARK entities in these units.

When analyzing a unit for the first time, it may help to gradually mark the code `SPARK_Mode => On`:

1. Start with the unit spec marked `SPARK_Mode => On` and the unit body marked `SPARK_Mode => Off`. First run GNATprove in `flow` mode, then in `proof` mode, until all errors are resolved (some unproved checks may remain, as errors and checks are different *Categories of Messages*).

2. Continue with the both the unit spec and body marked `SPARK_Mode => On`. First run GNATprove in `flow` mode, then in `proof` mode, until all errors are resolved.
3. Now that GNATprove can analyze the unit without any errors, continue with whatever analysis is required to achieve the desired objectives.

Choosing Which Run-time Checking to Keep

Inside proven SPARK code, no run-time errors of the kinds that GNATprove targets can be raised (see *Avoiding Errors to Enhance Portability* for details), provided the analysis assumptions are respected. See section *Managing Assumptions* for more details on assumptions. In such proven code, it is possible to remove run-time checking as described in section *Safe Optimization of Run-Time Checks*.

Note that GNATprove's analysis does not detect possible run-time errors corresponding to raising exception `Storage_Error` at run time. As described in "GNAT User's Guide for Native Platforms", section 6.6.1 on "Stack Overflow Checking", `gcc` option `-fstack-check` can be used to activate stack checking.

An important use case is the one of unproven code calling proven code, typically when rewriting core components of the application in SPARK. In that case, the guarantees provided by proof on SPARK code rely on the following main assumptions:

- The preconditions of proven SPARK subprograms should be respected. If these subprograms can be called from subprograms that are not proved, it is recommended to activate their preconditions at run time with *Pragma Assertion_Policy*, as shown in *Writing Contracts for Program Integrity*.
- All inputs of proven SPARK subprograms should have valid values for their types. This is enforced by the combination of flow analysis and proof in SPARK code, both for parameters and global variables that are read in the subprogram. It can be partially verified (for parameters but not global variables) during testing for calls from unproven subprograms by compiling the program with special switches to add run-time checks related to validity, as described in section *Between Proof and Unit Testing*.
- Inputs and outputs that may interfere should not be aliased. See section *Absence of Interferences* for details. Similar to validity, it can be partially verified (for parameters but not global variables) during testing for calls from unproven subprograms by compiling the program with special switch `-gnateA`, as described in section *Between Proof and Unit Testing*.

Inside unproven code, users may opt for keeping run-time checking and/or assertion checking in the executable or not, depending on their overall error detection and recovery policy. At the level of a compilation unit, this choice can be made through compilation switch `-gnatp` (for suppressing run-time checking) and `-gnata` (for activating assertion checking). These choices can be reversed for a selected piece of code with pragmas `Suppress` and `Unsuppress` (for all checks) and `Assertion_Policy` (for assertions only).

Additional compilation switches that activate validity checking are best kept for verification, as described in section *Between Proof and Unit Testing*. Activating them in the final executable may lead to large increases in running time, with some checks being inserted at unexpected/extra places, as these validity checks do not follow a formal definition like the one found in Ada Reference Manual for other run-time checks.

8.3.2 New Developments in SPARK

In this scenario, a significant part of a software (possibly a module, possibly the whole software) is developed in SPARK. Typically, SPARK is used for the most critical parts of the software, with less critical parts programmed in Ada, C or Java (for example the graphical interface). A typical development process for this scenario might be:

1. Produce the high level (architectural) design in terms of package specifications. Determine which packages will be in SPARK, to be marked `SPARK_Mode => On`.
2. Alternatively, if the majority of packages are to be SPARK, *Setting the Default SPARK_Mode* to `SPARK_Mode => On` is best. Those few units that are not SPARK should be marked `SPARK_Mode => Off`.
3. Add *Package Contracts* to SPARK packages and, depending on the desired objectives, add relevant *Subprogram Contracts* to the subprograms declared in these packages. The package contracts should identify the key elements of *State Abstraction* which might also be referred to in *Data Dependencies* and *Flow Dependencies*.
4. Begin implementing the package bodies. One typical method of doing this is to use a process of top-down decomposition, starting with a top-level subprogram specification and implementing the body by breaking it down into further (nested) subprograms which are themselves specified but not yet implemented, and to iterate until a level is reached where it is appropriate to start writing executable code. However the exact process is not mandated and will depend on other factors such as the design methodology being employed. Provided unimplemented subprograms are stubbed (that is, they are given dummy bodies), GNATprove can be used at any point to analyze the program.
5. As each subprogram is implemented, GNATprove can be used (in mode `flow` or `proof` depending on the objectives) to verify it (against its contract, and/or to show absence of run-time errors).

8.3.3 Conversion of Existing SPARK Software to SPARK 2014

If an existing piece of software has been developed in a previous version of SPARK and is still undergoing active development/maintenance then it may be advantageous to upgrade to using SPARK 2014 in order to make use of the larger language subset and the new tools and environment. This requires more efforts than previous upgrades between versions of SPARK (SPARK 83, SPARK 95 and SPARK 2005) because the new version SPARK 2014 of SPARK is incompatible with those previous versions of the language. While the programming language itself in those previous versions of SPARK is a strict subset of SPARK 2014, the contracts and assertions in previous versions of SPARK are expressed as stylized comments that are ignored by GNATprove. Instead, those contracts and assertions should be expressed as executable Ada constructs, as presented in the *Overview of SPARK Language*.

The SPARK Language Reference Manual has an appendix containing a *SPARK 2005 to SPARK 2014 Mapping Specification* which can be used to guide the conversion process. Various options can be considered for the conversion process:

1. *Only convert annotations into contracts and assertions, with minimal changes to the executable code* - Note that some changes to the code may be required when converting annotations, for example adding with-clauses in a unit to give visibility over entities used in contracts in this unit but defined in another units (which was performed in previous versions of SPARK with `inherit` annotations). This conversion should be relatively straightforward by following the mapping of features between the two languages.

   The SPARK tools should be used to analyze the work in progress throughout the conversion process (which implies that a bottom-up approach may work best) and any errors corrected as they are found. This may also be an occasion to dramatically simplify annotations, as GNATprove requires far fewer of them. See the README of the SPARKSkein program distributed with SPARK.

   Once the conversion is complete, development and maintenance can continue in SPARK.

2. *In addition to converting annotations, benefit from the larger language and more powerful tools to simplify code and contracts* - SPARK 2014 is far less constraining than previous versions of SPARK in terms of dependencies between units (which can form a graph instead of a tree), control structures (for example arbitrary return

statements and exit statements are allowed), data structures (for example scalar types with dynamic bounds are allowed), expressions (for example local variables can be initialized with non-static expressions at declaration). In addition, useful new language constructs are available:

- *Contract Cases* can be used to replace complex postconditions with implications.
- *Predicates* can be used to state invariant properties of subtypes, so that they need not be repeated in preconditions, postconditions, loop invariants, etc.
- *Expression Functions* can be used to replace simple query functions and their postcondition.
- *Ghost Code* can be used to mark code only used for verification.
- *Loop Variants* can be used to prove the termination of loops.

Changing the code to use these new features may favor readability and maintenance. These changes can be performed either while converting annotations, or as a second stage after all annotations have been converted (the case discussed above). Like in the previous case, the SPARK tools should be used to analyze the work in progress throughout the conversion process (which implies that a bottom-up approach may work best) and any errors corrected as they are found. Once the conversion is complete, development and maintenance can continue in SPARK.

3. *Gradually convert annotations and code* - It is possible to keep annotations in comments for the previous versions of SPARK while gradually adding contracts and assertions in SPARK 2014. The latest version of the SPARK 2005 toolset facilitates this gradual migration by ignoring SPARK pragmas. Thus, new contracts (for example *Preconditions* and *Postconditions*) should be expressed as pragmas rather than aspects in that case.

   Typically, annotations and code would be converted when it needs to be changed. The granularity of how much code needs to be converted when a module is touched should be considered, and is likely to be at the level of the whole package.

   The latest version of the SPARK 2005 toolset can be used to continue analyzing the parts of the program that do not use the new features of SPARK 2014, including units which have the two versions of contracts in parallel. GNATprove can be used to analyze parts of the program that have contracts in SPARK 2014 syntax, including units which have the two versions of contracts in parallel.

Note that some users may wish to take advantage of the new SPARK contracts and tools whilst retaining the more restrictive nature of SPARK 2005. (Many of the restrictions from SPARK 2005 have been lifted in SPARK because improvements in the tools mean that sound analysis can be performed without them, but some projects may need to operate in a more constrained environment.) This can be achieved using `pragma Restrictions (SPARK_05)`. For further details of this restriction please see the GNAT Reference Manual.

8.3.4 Analysis of Frozen Ada Software

In some very specific cases, users may be interested in the results of GNATprove's analysis on an unmodified code. This may be the case for example if the only objective is to *Ensure Portability of Programs* for existing Ada programs that cannot be modified (due to some certification or legal constraints).

In such a case, the suggested workflow is very similar to the one described for *Maintenance and Evolution of Existing Ada Software*, except the code cannot be rewritten when a violation of SPARK restrictions is encountered, and instead that part of the code should be marked `SPARK_Mode => Off`. To minimize the parts of the code that need to be marked `SPARK_Mode => Off`, it is in general preferable to apply `SPARK_Mode => On` selectively rather than by default, so that units that have non-SPARK declarations in the public part of their package spec need not be marked `SPARK_Mode => Off`. See *Using SPARK_Mode to Select or Exclude Code* for details.

8.3.5 Dealing with Storage_Error

As mentioned, SPARK analysis doesn't cover the possible exhaustion of data storage, either by exhausting the stack (this can happen by placing too much data on the stack, or via a too deep recursion) or by exhausting the heap (this can happen by allocating too much data using `new`).

To protect against stack exhaustion, we recommend using GNATstack.

As GNATstack doesn't analyze the secondary stack, if protection against exhaustion of the secondary stack is desired, we recommend using `pragma Restrictions (No_Secondary_Stack);`.

To protect against heap exhaustion, a possible way is to encapsulate allocations in a wrapper that handles the possible `Storage_Error` exception and signals the failure of the allocation to the calling environment via a return type. The verification of this wrapper cannot be effectively done with SPARK as the handler would be considered unreachable. The following example, inspired by this Stackoverflow post shows such a wrapper, that returns an "invalid" pointer that can't be dereferenced in case of memory exhaustion:

```
package Storage with SPARK_Mode is

   type Int_Ptr is access Integer;

   type Weak_Int_Ptr (Valid : Boolean := False) is record
      case Valid is
         when False => null;
         when True  => Ptr : Int_Ptr;
      end case;
   end record;

   function New_Integer (N : Integer) return Weak_Int_Ptr
     with Post => (if New_Integer'Result.Valid then New_Integer'Result.Ptr /= null);

   procedure Free (P : in out Weak_Int_Ptr)
     with
       Pre  => not P'Constrained,
       Post => P.Valid = False;

end Storage;
```

```
with Ada.Unchecked_Deallocation;
package body Storage with SPARK_Mode is

   function New_Integer (N : Integer) return Weak_Int_Ptr is
      pragma SPARK_Mode (Off);
   begin
      return Weak_Int_Ptr'(Valid => True, Ptr => new Integer'(N));

   exception
      when Storage_Error =>
         return Weak_Int_Ptr'(Valid => False);

   end New_Integer;

   procedure Free (P : in out Weak_Int_Ptr) is

      procedure Local_Free is new Ada.Unchecked_Deallocation
```

(continues on next page)

(continued from previous page)

```
        (Object => Integer, Name => Int_Ptr);

   begin
      if P.Valid then
         Local_Free (P.Ptr);
         P := Weak_Int_Ptr'(Valid => False);
      end if;
   end Free;
end Storage;
```

APPENDIX

A

COMMAND LINE INVOCATION

The following is an overview over the GNATprove commandline options. See also the section *Running GNATprove from the Command Line* for more detailed information for many of the switches.

```
Usage: gnatprove -Pproj [switches] [-cargs switches]

proj is a GNAT project file
-cargs switches are passed to gcc

All main units in proj are analyzed by default. Switches to change this:
 -u [files]            Analyze only the given files
    [files]            Analyze given files and all dependencies
 -U                    Analyze all files (including unused) of all projects
                       (can also be used with "--limit-*" switches to analyze
                       all instances of a generic)

gnatprove basic switches:
 -aP=p                 Add path p to project path
     --assumptions     Output assumptions information
     --clean           Remove GNATprove intermediate files, and exit
     --cwe             Include CWE ids in message output
     --explain=Ennnn   Output explanation for explain code Ennnn
                       associated to a message (error, warning or check)
 -f                    Force recompilation/analysis of all units
 -h, --help            Display this usage information
     --info            Output info messages about the analysis
 -j N                  Use N parallel processes (default: 1; N=0 will use
                       all cores of the machine)
 -k                    Do not stop analysis at the first error
     --level=n         Set the level of proof (0 = faster to 4 = more powerful)
     --list-categories Output a list of all message categories and exit
 -m                    Minimal reanalysis
     --mode=m          Set the mode of GNATprove (m=check, check_all, flow,
                       prove, all*, stone, bronze, silver, gold)
     --no-subprojects  Do not analyze subprojects, only the root project
     --output-msg-only Do not run any provers, output current flow and proof
                       results
 -q, --quiet           Be quiet/terse
     --replay          Replay proofs, do not attempt new proofs
     --report=r        Set the report mode of GNATprove (r=fail*, all,
                       provers, statistics)
```

(continues on next page)

(continued from previous page)

```
     --subdirs=p       Create all artifacts in this subdir
 -v, --verbose         Output extra verbose information
     --version         Output version of the tool and exit
     --warnings=w      Set the warning mode of GNATprove
                       (w=off, continue*, error)

 * Main mode values
   . check               - Fast partial check for SPARK violations
   . check_all, stone    - Full check for SPARK violations
   . flow, bronze        - Prove correct initialization and data flow
   . prove               - Prove absence of run-time errors and contracts
   . all, silver, gold   - Activates all modes (default)

 * Report mode values
   . fail          - Report failures to prove checks (default)
   . all           - Report all results of proving checks
   . provers       - Same as all, plus prover usage information
   . statistics    - Same as provers, plus timing and steps information

 * Warning mode values
   . off           - Do not issue warnings
   . continue      - Issue warnings and continue (default)
   . error         - Treat warnings as errors

gnatprove advanced switches:
 --check-counterexamples=c
                       Enable or disable checking of counterexample (c=on*,off)
 --checks-as-errors=c Treat unproved check messages as errors (c=on,off*)
 --ce-steps=nnn       Set the maximum number of proof steps for counterexamples.
                       This replaces the use of timeout for counterexamples.
 --counterexamples=c  Enable or disable counterexamples (c=on,off*)
 -d, --debug          Debug mode
 --debug-save-vcs     Do not delete intermediate files for provers
 --debug-exec-rac     Only execute runtime assertion checking (RAC) and exit
 --flow-debug         Extra debugging for flow analysis (requires graphviz)
 --function-sandboxing=c
                       Enable or disable the generation of guards for axioms
                       providing contracts of functions (c=on*,off)
 --limit-line=f:l     Limit analysis to given file and line
 --limit-line=f:l:c:k Limit analysis to given file, line, column and kind of
                       check
 --limit-name=s       Limit analysis to subprogram with the specified name
 --limit-region=f:l:l Limit analysis to given file and range of lines
 --limit-subp=f:l     Limit analysis to subprogram declared at the specified
                       location
 --memcached-server=host:portnumber
                       Specify a memcached instance that will be used for
                       caching of proof results.
 --memcached-server=file:directory
                       Use the fixed string "file" for part before the colon.
                       The cache will be stored in the directory after the
                       colon. Best for CI integration.
```

(continues on next page)

(continued from previous page)

```
--memlimit=nnn       Set the prover memory limit in MB. Use value 0 for
                     no limit (default when no level set)
--no-global-generation
                     Do not generate Global and Initializes contracts from
                     code, instead assume "null". Note that this option also
                     implies --no-inlining.
--no-inlining        Do not inline calls to local subprograms for proof
--no-loop-unrolling  Do not unroll loops with static bounds and no
                     (in)variant for proof
--output=o           Set the output mode of GNATprove (o=brief, oneline,
                     pretty*)
--output-header      Add a header with extra information in the generated
                     output file
--pedantic           Use a strict interpretation of the Ada standard
--proof-warnings     Issue warnings by proof
--proof-warnings-timeout
                     Set the timeout for proof warnings
--proof=g[:l]        Set the proof modes for generation of formulas
                     (g=per_check*, per_path, progressive) (l=lazy*, all)
--prover=s[,s]*      Use given provers (s=altergo, cvc5*, z3, ..., or s=all
                     for using all built-in provers)
--RTS=dir            Specify the Ada runtime name/location
--steps=nnn          Set the maximum number of proof steps (prover-specific)
                     Use value 0 for no steps limit.
--timeout=nnn        Set the prover timeout in seconds. Use value 0 for
                     no timeout (default when no level set)
--why3-conf=f        Specify a configuration file for why3

* Output mode values
  . brief         - Output minimal check message on a single line
  . oneline       - Output rich check message on a single line
  . pretty        - Pretty-print check messages for command-line use (default)

* Proof mode values for generation
  . per_check     - Generate one formula per check (default when no level set)
  . per_path      - Generate one formula per path for each check
  . progressive   - Start with one formula per check, then split into
                    paths when needed

* Proof mode values for laziness
  . lazy          - Stop at first unproved formula for each check
                    (most suited for fully automatic proof) (default)
  . all           - Attempt to prove all formulas
                    (most suited for combination of automatic and manual proof)

* Prover name values
  (Default prover is cvc5.)
  . altergo       - Use Alt-Ergo
  . colibri       - Use Colibri
  . cvc5          - Use CVC5
  . z3            - Use Z3
  . ...           - Any other prover configured in your .why3.conf file
```

APPENDIX

B

ALTERNATIVE PROVERS

B.1 Installed with SPARK Pro

The provers Alt-Ergo, Colibri, cvc5 and Z3 are installed with the SPARK tool. By default, GNATprove uses prover cvc5 only. Switch `--level` changes the default to use one or more provers depending on the chosen level (see *Running GNATprove from the Command Line*). Switch `--prover` allows to use another prover, or a list of provers. Prover names `altergo`, `colibri`, `cvc5` and `z3` are used to refer to the versions of provers Alt-Ergo, Colibri, cvc5 and Z3 that are installed with the SPARK toolset. The string `alt-ergo` can also be used to refer to Alt-Ergo. Using the switch `--prover=all`, one can select all four built-in provers, in the order `cvc5, z3, colibri, altergo`. More information on Alt-Ergo, cvc5 and Z3 can be found on their respective websites:

- Alt-Ergo: https://alt-ergo.ocamlpro.com
- cvc5: https://cvc5.github.io/
- Z3: https://github.com/Z3Prover/z3

B.2 Installed with SPARK Discovery

In this case, only prover Alt-Ergo is installed with the SPARK tool. Hence, by default GNATprove only uses prover Alt-Ergo. In particular, switch `--level` has no impact on the use of different provers, and `--prover=all` will only select Alt-Ergo.

B.3 Installed with SPARK Community

The provers Alt-Ergo, cvc5 and Z3 are installed with the SPARK tool.

B.4 Other Automatic or Manual Provers

B.4.1 Updating the Why3 Configuration File

GNATprove can call other provers, as long as they are supported by the Why3 platform (see complete list on Why3 webpage). To use another prover, it must be listed in your Why3 configuration file.

To create or update automatically a Why3 configuration file, call the command `<spark2014-install>/libexec/spark/bin/why3config --detect-provers`. It searches your `PATH` for any supported provers and adds them to the default configuration file `.why3.conf` in your `HOME`, or a configuration file given in argument with switch `-C <file>`. This file consists of a few general settings and a section for each prover which is supported.

Note that GNATprove never reads the default configuration file `.why3.conf` in your `HOME`. You need to pass the configuration file explicitly with switch `--why3-conf=<file>`. Any prover name configured in this configuration file can be used as an argument to switch `--prover`.

Note that using this mechanism, you cannot replace the definitions provided with the SPARK tools for the provers `altergo`, `colibri`, `cvc5` and `z3`.

If more than one prover is specified, the provers are tried in order on each VC, until one of them succeeds or all fail. Interactive provers cannot be combined with other provers, so must appear on their own.

B.4.2 Sharing Libraries of Theorems

When GNATprove is used with a manual prover, the user can provide libraries of theorems to use during the proof process.

To do so, the user will need to set a proof directory (see *Project Attributes* for more details on this directory). The user needs to create a folder with the same name as the chosen manual prover (the casing of the name is the same as the one passed to the switch `--prover`) and put the library sources inside this folder.

Finally, some additional fields need to be added to the prover configuration in the Why3 configuration file (a basic example of prover configuration can be found in the section on *Coq*):

- `configure_build`: this field allows you to specify a command to configure the compilation of the library of theorems. This command will be called each time a source file is added to the library.
- `build_commands`: this field allows you to specify a set of command which will be called sequentially to build your library. These commands will be called each time GNATprove runs the corresponding manual prover. (In order to define multiple commands for this field, just set the field multiple times with different values, each time the field is set it adds a new element to the set of `build_commands`).

Inside these commands, pattern `%f` refers to the name of the library file considered, and `%o` to the name of the main `gnatprove` repository generated by GNATprove. This allows referring to the path of the compiled library of theorems inside these commands with `%o/user/<prover_name>`.

B.5 Coq

`gnatprove` has support for the Coq interactive prover, even though Coq is not part of the SPARK distribution. If you want to use Coq with SPARK, you need to install it yourself on your system and put it in your `PATH` environment variable. Then, you can simply provide `--prover=coq` to `gnatprove`. Note that the only supported version currently is Coq 8.11.

APPENDIX

C

PROJECT ATTRIBUTES

GNATprove reads the package `Prove` in the given project file. This package is allowed to contain the following attributes:

- `Proof_Switches`, which defines additional command line switches that are used for the invokation of GNATprove. This attribute can be used in two different settings:
  - to define switches that should apply to all files in the project. As an example, the following package in the project file sets the default report mode of GNATprove to `all`:

```
package Prove is
   for Proof_Switches ("Ada") use ("--report=all");
end Prove;
```

  - to define switches that should apply only to one file. The following example sets timeout for provers run by GNATprove to 10 seconds for `file.adb`:

```
package Prove is
   for Proof_Switches ("file.adb") use ("--timeout=10");
end Prove;
```

    Note that, if a unit has both a body and specification file, the body file should be used for this attribute.

  Switches given on the command line have priority over switches given in the project file, and file-specific switches have priority over switches that apply to all files. A special case is the `--level` switch: the values for `--timeout` etc implied by the `--level` switch are always overridden by more specific switches, regardless of where they appear. For example, the timeout for the analysis of `file.adb` is set to 10 seconds below, despite the `--level=0` switch (which implies a lower timeout) specified for this file:

```
package Prove is
   for Proof_Switches ("Ada") use ("--timeout=10");
   for Proof_Switches ("file.adb") use ("--level=0");
end Prove;
```

  The following switches cannot be used inside project files: `-P`, `-aP`, `--subdirs`, `--clean`, `--list-categories`, `--version`.

  Only the following switches are allowed for file-specific switches: `--steps`, `--timeout`, `--memlimit`, `--proof`, `--prover`, `--level`, `--mode`, `--counterexamples`, `--no-inlining`, `--no-loop-unrolling`.

- `Switches`. This deprecated attribute is the same as `Proof_Switches ("Ada")`.
- `Proof_Dir`, which defines the directory where are stored the files concerning the state of the proof of a project. This directory contains a sub-directory `sessions` with one directory per source package analyzed for proof. Each of these package directories contains a Why3 session file. If a manual prover is used to prove some VCs, then a sub-directory called by the name of the prover is created next to `sessions`, with the same organization of

sub-directories. Each of these package directories contains manual proof files. Common proof files to be used across various proofs can be stored at the toplevel of the prover-specific directory.

APPENDIX

D

IMPLEMENTATION DEFINED ASPECTS AND PRAGMAS

This appendix lists all the aspects or pragmas specific to GNATprove.

D.1 Aspect and Pragma `SPARK_Mode`

SPARK_Mode is a three-valued aspect. At least until we get to the next paragraph, a SPARK_Mode of On, Off, or Auto is associated with each Ada construct. Roughly, the meaning of the three values is the following:

- a value of On means that the construct is required to be in SPARK, and the construct will be analyzed by GNATprove.
- a value of Off means that the construct will not be analyzed by GNATprove, and does not need to obey the SPARK restrictions. The construct also cannot be referenced from other parts that are required to be in SPARK.
- a value of Auto means that the construct will not be analyzed, and GNATprove will infer whether this construct can be used in other SPARK parts or not.

As generic units are not directly analyzed by GNATprove, but only generic instances are, the meaning of the three values is slightly different for generic units:

- a value of On means that instances can be in SPARK, depending on the parameters passed for the instantiation.
- a value of Off means that instances cannot be in SPARK, independent of the parameters passed for the instantiation.
- a value of Auto provides no information on whether instances can be in SPARK or not.

We now explain in more detail how the SPARK_Mode pragma works.

Some Ada constructs are said to have more than one "section". For example, a declaration which requires a completion will have (at least) two sections: the initial declaration and the completion. The SPARK_Modes of the different sections of one entity may differ. In other words, SPARK_Mode is not an aspect of an entity but rather of a section of an entity.

For example, if a subprogram declaration has a SPARK_Mode of On while its body has a SPARK_Mode of Off, then an error would be generated if the subprogram took a parameter of a general access type but not if the subprogram declared a local variable of a general access type (recall that general access types are not in SPARK).

A package is defined to have 4 sections: its visible part, its private part, its body declarations, and its body statements. A protected or task unit has 3 sections: its visible part, its private part, and its body. Other declarations which require a completion have two sections, as noted above; all other entities and constructs have only one section.

If the SPARK_Mode of a section of an entity is Off, then the SPARK_Mode of a later section of that entity shall not be On. [For example, a subprogram can have a SPARK declaration and a non-SPARK body, but not vice versa.]

If the SPARK_Mode of a section of an entity is Auto, then the SPARK_Mode of a later section of that entity shall not be On, and it shall not be Off unless that entity is a generic entity, or an instance of such a generic. [This makes it

possible to mark a later section of a generic unit as Off, in cases where its initial section is Auto to allow instantiations to have any value of SPARK_Mode.]

The SPARK_Mode aspect can be specified either via a pragma or via an aspect_specification. In some contexts, only a pragma can be used because of syntactic limitations. In those contexts where an aspect_specification can be used, it has the same effect as a corresponding pragma.

The form of a pragma SPARK_Mode is as follows:

```
pragma SPARK_Mode [ (On | Off) ]
```

The form for the aspect_definition of a SPARK_Mode aspect_specification is as follows:

```
[ On | Off ]
```

For example:

```
package P
   with SPARK_Mode => On
is
```

The pragma can be used as a configuration pragma. The effect of such a configuration pragma is described below in the rules for determining the SPARK_Mode aspect value for an arbitrary section of an arbitrary Ada entity or construct.

Pragma `SPARK_Mode` shall be used as a local pragma in only the following contexts and has the described semantics:

Pragma placement	Affected construct	Alternative aspect form
Start of the visible declarations (preceded only by other pragmas) of a package declaration	Visible part of the package	As part of the package_specification
Start of the visible declarations (preceded only by other pragmas) of a task or protected unit	Visible part of the unit	As part of the declaration
Start of the private declarations of a package, a protected unit, or a task unit (only other pragmas can appear between the `private` keyword and the `SPARK_Mode` pragma)	Private part	None
Immediately at the start of the declarations of a package body (preceded only by other pragmas)	Body declarations of the package	As part of the package_body
Start of the elaboration statements of a package body (only other pragmas can appear between the `begin` keyword and the `SPARK_Mode` pragma)	Body statements of the package	None
Start of the declarations of a protected or task body (preceded only by other pragmas)	Body	As part of the protected or task body
After a subprogram declaration (with only other pragmas intervening). [This does not include the case of a subprogram whose initial declaration is via a subprogram_body_stub. Such a subprogram has only one section because a subunit is not a completion.]	Subprogram's specification	As part of the subprogram_declaration
Start of the declarations of a subprogram body (preceded only by other pragmas)	Subprogram's body	As part of the subprogram_body

A default argument of On is assumed for any SPARK_Mode pragma or aspect_specification for which no argument is explicitly specified.

A SPARK_Mode of Auto can only be explicitly specified for a configuration pragma; the cases in which a SPARK_Mode of Auto is implicitly specified are described below. Roughly speaking, Auto indicates that it is left up to the formal verification tools to determine whether or not a given construct is in SPARK.

A SPARK_Mode pragma or aspect specification shall only apply to a (section of a) package, generic package, subprogram, or generic subprogram.

A SPARK_Mode of On shall only apply to a (section of a) library-level entity, except for the case of SPARK_Mode specifications occurring within generic instances. A SPARK_Mode of On applying to a non-library-level entity within a generic instance has no effect.

The SPARK_Mode aspect value of an arbitrary section of an arbitrary Ada entity or construct is then defined to be the following value (except if this yields a result of Auto for a non-package; see below):

- If SPARK_Mode has been specified for the given section of the given entity or construct, then the specified value;
- else for the instance of a generic unit, follow the rules as for a declaration that would not be a generic instantiation; take the resulting value of SPARK_Mode if it is Auto or Off; otherwise, take the value of SPARK_Mode specified for the generic unit if any; otherwise the value is On.
- else for the private part of a public child unit whose parent unit's private part has a SPARK_Mode of Off, the SPARK_Mode is Off;
- else for the private part of a package or a protected or task unit, the SPARK_Mode of the visible part;
- else for a package body's statements, the SPARK_Mode of the package body's declarations;
- else for the first section (in the case of a package, the visible part) of a public child unit, the SPARK_Mode of the visible part of the parent unit;
- else for the first section (in the case of a package, the visible part) of a private child unit, the SPARK_Mode of the private part of the parent unit;
- else for any of the visible part or body declarations of a library unit package or either section of a library unit subprogram, if there is an applicable SPARK_Mode configuration pragma then the value specified by the pragma; if no such configuration pragma applies, then an implicit specification of Auto is assumed;
- else the SPARK_Mode of the enclosing section of the nearest enclosing package or subprogram;
- Corner case: the SPARK_Mode of the visible declarations of the limited view of a package is always Auto.

If the above computation yields a result of Auto for any construct other than one of the four sections of a package, then a result of On or Off is determined instead based on the legality (with respect to the rules of SPARK) of the construct. The construct's SPARK_Mode is On if and only if the construct is in SPARK. [A SPARK_Mode of Auto is therefore only possible for (sections of) a package.]

In code where SPARK_Mode is On (also called "SPARK code"), the rules of SPARK are enforced. In particular, such code shall not reference non-SPARK entities, although such code may reference a SPARK declaration with one or more non-SPARK subsequent sections (e.g., a package whose visible part has a SPARK_Mode of On but whose private part has a SPARK_Mode of Off; a package whose visible part has a SPARK_Mode of Auto may also be referenced).

Code where SPARK_Mode is Off shall not enclose code where Spark_Mode is On. However, if an instance of a generic unit is enclosed by code where SPARK_Mode is Off and if any SPARK_Mode specifications occur within the generic unit, then the corresponding SPARK_Mode specifications occurring within the instance have no semantic effect. [In particular, such an ignored SPARK_Mode specification could not violate the preceding "Off shall not enclose On" rule because the SPARK_Mode of the entire instance is Off. Similarly, such an ignored SPARK_Mode specification could not violate the preceding rule that a SPARK_Mode specification shall only apply to a (section of a) library-level entity.]

For purposes of the "Off shall not enclose On" rule just described, the initial section of a child unit is considered to occur immediately within either the visible part (for a public child unit) or the private part (for a private child unit) of the parent unit. In addition, the private part of a public child package is considered to occur immediately within the private part of the parent unit. [This follows Ada's visibility rules for child units. This means, for example, that if a parent unit's private part has a SPARK_Mode of Off, then the private part of a public child package shall not have a SPARK_Node of On. Note also that a SPARK_Mode configuration pragma which applies only to the specification (not the body) of a child unit is always ineffective; this is a consequence of the rules given above for determining the SPARK_Mode of the first section of a child unit.]

The rules for a protected unit follow from the rules given for other constructs after notionally rewriting the protected unit as a package.

A protected unit declaration such as

```
protected type Prot
  with SPARK_Mode => On
is
   procedure Op1 (X : in out Integer);
   procedure Op2;
   function Non_SPARK_Profile (X : in out Integer) return Boolean
     with SPARK_Mode => Off;
private
   Aaa, Bbb : Integer := 0;
end Prot;
```

can be thought of, for purposes of SPARK_Mode rules, as being a lot like

```
package Pkg
  with SPARK_Mode => On
is
   type Prot is limited private;
   procedure Op1 (Obj : in out Prot; X : in out Integer);
   procedure Op2 (Obj : in out Prot);
   function Non_SPARK_Profile (Obj : Prot; Ptr : in out Integer) return Boolean
     with SPARK_Mode => Off;
private
   type Prot is
     limited record
        Aaa, Bbb : Integer := 0;
     end record;
end Pkg;
```

which is legal. The point is that a protected type which is in SPARK can have protected operation whose declaration is not in SPARK.

SPARK_Mode is an implementation-defined Ada aspect; it is not (strictly speaking) part of the SPARK language. It is used to notionally transform programs which would otherwise not be in SPARK so that they can be viewed (at least in part) as SPARK programs.

Note that if you would like to mark all your code in SPARK_Mode, the simplest solution is to specify in your project file:

```
package Builder is
   for Global_Configuration_Pragmas use "spark.adc";
end Builder;
```

and provide a file *spark.adc* which contains:

```
pragma SPARK_Mode;
```

D.2 Aspect and Pragma `Iterable`

In SPARK, it is possible to allow quantification over any container type using the `Iterable` aspect. This aspect provides the primitives of a container type that will be used to iterate over its content. For example, if we write:

```
type Container is private with
  Iterable => (First       => First,
               Next        => Next,
               Has_Element => Has_Element);
```

where

```
function First (S : Set) return Cursor;
function Has_Element (S : Set; C : Cursor) return Boolean;
function Next (S : Set; C : Cursor) return Cursor;
```

then quantification over containers can be done using the type `Cursor`. For example, we could state:

```
(for all C in S => P (Element (S, C)))
```

to say that `S` only contains elements for which a property `P` holds. For execution, this expression is translated as a loop using the provided `First`, `Has_Element`, and `Next` primitives. For proof, it is translated as a logic quantification over every element of type `Cursor`. To restrict the property to cursors that are actually valid in the container, the provided function `Has_Element` is used. For example, the property stated above becomes:

```
(for all C : Cursor => (if Has_Element (S, C) then P (Element (S, C))))
```

Like for the standard Ada iteration mechanism, it is possible to allow quantification directly over the elements of the container by providing in addition an `Element` primitive to the `Iterable` aspect. For example, if we write:

```
type Container is private with
  Iterable => (First       => First,
               Next        => Next,
               Has_Element => Has_Element
               Element     => Element);
```

where

```
function Element (S : Set; C : Cursor) return Element_Type;
```

then quantification over containers can be done directly on its elements. For example, we could rewrite the above property into:

```
(for all E of S => P (E))
```

For execution, quantification over elements of a container is translated as a loop over its cursors. In the same way, for proof, quantification over elements of a container is no more than syntactic sugar for quantification over its cursors. For example, the above property is translated using quantification over cursors :

```
(for all C : Cursor => (if Has_Element (S, C) then P (Element (S, C))))
```

Depending on the application, this translation may be too low-level and introduce an unnecessary burden on the automatic provers. As an example, let us consider a package for functional sets:

```
package Sets with SPARK_Mode is

  type Cursor is private;
  type Set (<>) is private with
    Iterable => (First       => First,
                 Next        => Next,
                 Has_Element => Has_Element,
                 Element     => Element);

  function Mem (S : Set; E : Element_Type) return Boolean with
    Post => Mem'Result = (for some F of S => F = E);

  function Intersection (S1, S2 : Set) return Set with
    Post => (for all E of Intersection'Result => Mem (S1, E) and Mem (S2, E))
      and (for all E of S1 =>
               (if Mem (S2, E) then Mem (Intersection'Result, E)));
```

Sets contain elements of type `Element_Type`. The most basic operation on sets is membership test, here provided by the `Mem` subprogram. Every other operation, such as intersection here, is then specified in terms of members. Iteration primitives `First`, `Next`, `Has_Element`, and `Element`, that take elements of a private type `Cursor` as an argument, are only provided for the sake of quantification.

Following the scheme described previously, the postcondition of `Intersection` is translated for proof as:

```
(for all C : Cursor =>
    (if Has_Element (Intersection'Result, C) then
           Mem (S1, Element (Intersection'Result, C))
       and Mem (S2, Element (Intersection'Result, C))))
and
(for all C1 : Cursor =>
    (if Has_Element (S1, C1) then
           (if Mem (S2, Element (S1, C1)) then
                 Mem (Intersection'Result, Element (S1, C1)))))
```

Using the postcondition of `Mem`, this can be refined further into:

```
(for all C : Cursor =>
    (if Has_Element (Intersection'Result, C) then
           (for some C1 : Cursor =>
               Has_Element (S1, C1) and Element (Intersection'Result, C) = Element (S1,␣
↪C1))
       and (for some C2 : Cursor =>
               Has_Element (S2, C2) and Element (Intersection'Result, C) = Element (S2,␣
↪C2))))
and
(for all C1 : Cursor =>
    (if Has_Element (S1, C1) then
           (if (for some C2 : Cursor =>
               Has_Element (S2, C2) and Element (S1, C1) = Element (S2, C2)))
    then (for some C : Cursor => Has_Element (Intersection'Result, C)
             and Element (Intersection'Result, C) = Element (S1, C1))))
```

APPENDIX

E

IMPLEMENTATION DEFINED ANNOTATIONS

This appendix lists all the uses of aspect or pragma `Annotate` for GNATprove. Aspect or pragma `Annotate` can also be used to control other AdaCore tools. The uses of such annotations are explained in the User's Guide of each tool.

Annotations in GNATprove are useful in two cases:

1. for justifying check messages using *Direct Justification with Pragma Annotate*, typically using a pragma rather than an aspect, as the justification is generally associated to a statement or declaration.
2. for influencing the generation of proof obligations, typically using an aspect rather than a pragma, as the annotation is generally associated to an entity in that case. Some of these uses can be seen in *SPARK Libraries* for example. Some of these annotations introduce additional assumptions which are not verified by the GNATprove tool, and thus should be used with care.

When the annotation is associated to an entity, both the pragma and aspect form can be used and are equivalent, for example on a subprogram:

```
function Func (X : T) return T
  with Annotate => (GNATprove, <annotation name>);
```

or

```
function Func (X : T) return T;
pragma Annotate (GNATprove, <annotation name>, Func);
```

In the following, we use the aspect form whenever possible.

E.1 Annotation for Justifying Check Messages

You can use annotations of the form

```
pragma Annotate (GNATprove, False_Positive,
                 "message to be justified", "reason");
```

to justify an unproved check message that cannot be proved by other means. See the section *Direct Justification with Pragma Annotate* for more details about this use of pragma `Annotate`.

E.2 Annotation for Skipping Parts of the Analysis for an Entity

Subprograms, packages, tasks, entries and protected subprograms can be annotated to skip flow analysis, and to skip generating proof obligations for their implementation, and the implementations of all such entities defined inside.

```
procedure P
  with Annotate => (GNATprove, Skip_Proof);

procedure P
  with Annotate => (GNATprove, Skip_Flow_And_Proof);
```

If an entity is annotated with `Skip_Proof`, no messages related to possible run-time errors and functional contracts are issued for this entity and any contained entities. This is similar to specifying `--mode=flow` on the command line (see *Effect of Mode on Output*), except that the effect is limited to this entity (and enclosed entities).

If an entity is annotated with `Skip_Flow_And_Proof`, in addition, no messages related to global contracts, initialization and dependency relations are issues for this entity and any contained entities. This is similar to specifying `--mode=check-all` on the command line, expect that the effect is limited to this entity (and enclosed entities).

Note that the `Skip_Proof` annotation cannot be used if an enclosing subprogram already has the `Skip_Flow_And_Proof` annotation.

E.3 Annotation for Overflow Checking on Modular Types

The standard semantics of arithmetic on modular types is that operations wrap around, hence GNATprove issues no overflow checks on such operations. You can instruct it to issue such checks (hence detecting possible wrap-around) using annotations of the form:

```
type T is mod 2**32
  with Annotate => (GNATprove, No_Wrap_Around);
```

or on a derived type:

```
type T is new U
  with Annotate => (GNATprove, No_Wrap_Around);
```

This annotation is inherited by derived types. It must be specified on a type declaration (and cannot be specified on a subtype declaration). All four binary arithmetic operations + - * ** are checked for possible overflows. Division cannot lead to overflow. Unary negation is checked for possible non-nullity of its argument, which leads to overflow. The predecessor attribute `'Pred` and successor attribute `'Succ` are also checked for possible overflows.

E.4 Annotation for Simplifying Iteration for Proof

The translation presented in :ref:`Aspect and Pragma Iterable` may produce complicated proofs, especially when verifying complex properties over sets. The |GNATprove| annotation ``Iterable_For_Proof`` can be used to change the way `for ... of` quantification is translated. More precisely, it allows to provide GNATprove with a *Contains* function, that will be used for quantification. For example, on our sets, we could write:

```
function Mem (S : Set; E : Element_Type) return Boolean;
pragma Annotate (GNATprove, Iterable_For_Proof, "Contains", Mem);
```

With this annotation, the postcondition of `Intersection` is translated in a simpler way, using logic quantification directly over elements:

```
(for all E : Element_Type =>
     (if Mem (Intersection'Result, E) then Mem (S1, E) and Mem (S2, E)))
and (for all E : Element_Type =>
     (if Mem (S1, E) then
            (if Mem (S2, E) then Mem (Intersection'Result, E))))
```

Note that care should be taken to provide an appropriate function contains, which returns true if and only if the element `E` is present in `S`. This assumption will not be verified by GNATprove.

The annotation `Iterable_For_Proof` can also be used in another case. Operations over complex data structures are sometimes specified using operations over a simpler model type. In this case, it may be more appropriate to translate `for ... of` quantification as quantification over the model's cursors. As an example, let us consider a package of linked lists that is specified using a sequence that allows accessing the element stored at each position:

```
package Lists with SPARK_Mode is

 type Sequence is private with
   Ghost,
   Iterable => (...,
                Element     => Get);
 function Length (M : Sequence) return Natural with Ghost;
 function Get (M : Sequence; P : Positive) return Element_Type with
   Ghost,
   Pre => P <= Length (M);

 type Cursor is private;
 type List is private with
   Iterable => (...,
                Element     => Element);

 function Position (L : List; C : Cursor) return Positive with Ghost;
 function Model (L : List) return Sequence with
   Ghost,
   Post => (for all I in 1 .. Length (Model'Result) =>
                (for some C in L => Position (L, C) = I));

 function Element (L : List; C : Cursor) return Element_Type with
   Pre  => Has_Element (L, C),
   Post => Element'Result = Get (Model (L), Position (L, C));

 function Has_Element (L : List; C : Cursor) return Boolean with
   Post => Has_Element'Result = (Position (L, C) in 1 .. Length (Model (L)));

 procedure Append (L : in out List; E : Element_Type) with
   Post => length (Model (L)) = Length (Model (L))'Old + 1
   and Get (Model (L), Length (Model (L))) = E
   and (for all I in 1 .. Length (Model (L))'Old =>
          Get (Model (L), I) = Get (Model (L'Old), I));

 function Init (N : Natural; E : Element_Type) return List with
   Post => length (Model (Init'Result)) = N
```

(continues on next page)

(continued from previous page)

```
     and (for all F of Init'Result => F = E);
```

Elements of lists can only be accessed through cursors. To specify easily the effects of position-based operations such as `Append`, we introduce a ghost type `Sequence`, that is used to represent logically the content of the linked list in specifications. The sequence associated to a list can be constructed using the `Model` function. Following the usual translation scheme for quantified expressions, the last line of the postcondition of `Init` is translated for proof as:

```
(for all C : Cursor =>
    (if Has_Element (Init'Result, C) then Element (Init'Result, C) = E));
```

Using the definition of `Element` and `Has_Element`, it can then be refined further into:

```
(for all C : Cursor =>
    (if Position (Init'Result, C) in 1 .. Length (Model (Init'Result))
     then Get (Model (Init'Result), Position (Init'Result, C)) = E));
```

To be able to link this property with other properties specified directly on models, like the postcondition of `Append`, it needs to be lifted to iterate over positions instead of cursors. This can be done using the postcondition of `Model` that states that there is a valid cursor in `L` for each position of its model. This lifting requires a lot of quantifier reasoning from the prover, thus making proofs more difficult.

The GNATprove `Iterable_For_Proof` annotation can be used to provide GNATprove with a *Model* function, that will be to translate quantification on complex containers toward quantification on their model. For example, on our lists, we could write:

```
function Model (L : List) return Sequence;
pragma Annotate (GNATprove, Iterable_For_Proof, "Model", Entity => Model);
```

With this annotation, the postcondition of `Init` is translated directly as a quantification on the elements of the result's model:

```
(for all I : Positive =>
   (if I in 1 .. Length (Model (Init'Result)) then
      Get (Model (Init'Result), I) = E));
```

Like with the previous annotation, care should be taken to define the model function such that it always return a model containing exactly the same elements as `L`.

Note: It is not possible to specify more than one `Iterable_For_Proof` annotation per container type with an `Iterable` aspect.

E.5 Annotation for Inlining Functions for Proof

Contracts for functions are generally translated by GNATprove as axioms on otherwise undefined functions. As an example, consider the following function:

```
function Increment (X : Integer) return Integer with
  Post => Increment'Result >= X;
```

It will be translated by GNATprove as follows:

```
function Increment (X : Integer) return Integer;

axiom : (for all X : Integer. Increment (X) >= X);
```

For internal reasons due to ordering issues, expression functions are also defined using axioms. For example:

```
function Is_Positive (X : Integer) return Boolean is (X > 0);
```

will be translated exactly as if its definition was given through a postcondition, namely:

```
function Is_Positive (X : Integer) return Boolean;

axiom : (for all X : Integer. Is_Positive (X) = (X > 0));
```

This encoding may sometimes cause difficulties to the underlying solvers, especially for quantifier instantiation heuristics. This can cause strange behaviors, where an assertion is proven when some calls to expression functions are manually inlined but not without this inlining.

If such a case occurs, it is sometimes possible to instruct the tool to inline the definition of expression functions using pragma `Annotate Inline_For_Proof`. When such a pragma is provided for an expression function, a direct definition will be used for the function instead of an axiom:

```
function Is_Positive (X : Integer) return Boolean is (X > 0);
pragma Annotate (GNATprove, Inline_For_Proof, Is_Positive);
```

The same pragma will also allow to inline a regular function, if its postcondition is simply an equality between its result and an expression:

```
function Is_Positive (X : Integer) return Boolean with
  Post => Is_Positive'Result = (X > 0);
pragma Annotate (GNATprove, Inline_For_Proof, Is_Positive);
```

In this case, GNATprove will introduce a check when verifying the body of `Is_Positive` to make sure that the inline annotation is correct, namely, that `Is_Positive (X)` and `X > 0` always yield the same result. This check may not be redundant with the verification of the postcondition of `Is_Positive` if the = symbol on booleans has been overridden.

Note that, since the translation through axioms is necessary for ordering issues, this annotation can sometimes lead to a crash in GNATprove. It is the case for example when the definition of the function uses quantification over a container using the `Iterable` aspect.

E.6 Annotation for Referring to a Value at the End of a Local Borrow

Local borrowers are objects of an anonymous access-to-variable type. At their declaration, the ownership of (a part of) an existing data-structure is temporarily transferred to the new object. The borrowed data-structure will regain ownership afterward. During the lifetime of the borrower, the borrowed object can be accessed indirectly through the borrower. It is forbidden to modify or even read the borrowed object during the borrow. As an example, let us consider a recursive type of doubly-linked lists:

```
type List;
type List_Acc is access List;
type List is record
   Val  : Integer;
```

(continues on next page)

(continued from previous page)

```
   Next : List_Acc;
end record;
```

Using this type, let us construct a list `X` which stores the numbers from 1 to 5:

```
X := new List'(1, null);
X.Next := new List'(2, null);
X.Next.Next := new List'(3, null);
X.Next.Next.Next := new List'(4, null);
X.Next.Next.Next.Next := new List'(5, null);
```

We can borrow the structure designated by `X` in a local borrower `Y`:

```
declare
   Y : access List := X;
begin
   V := Y.Val; --  OK, ownership has been transferred to Y temporarily
   V := X.Val; --  Illegal, X does not have ownership during the scope of Y
end;
```

While in the scope of `Y`, the ownership of the list designated by `X` is transferred to `Y`, so that it is not allowed to access it from `X` anymore. After the end of the declare block, ownership is restored to `X`, which can again be accessed or modified directly.

During the lifetime of the borrower, the borrowed object can be modified indirectly through the borrower. Therefore, when the borrower goes out of scope and ownership is transferred back to the borrowed object, GNATprove needs to reconstruct the new value of the borrowed object from the value of the borrower at the end of the borrow. In general, it can be done entirely automatically. However, it can happen that the exact relation between the values of the borrowed object and the borrower at the end of the borrow is lost by the tool. In particular, it is the case when the borrower is created inside a (traversal) function, as proof is modular on a per subprogram basis. It also happens when the borrower is modified inside a loop, as analysing loops involves cutpoints. In this case, GNATprove relies on the user to adequately describe the link between the values of the borrowed object and the borrower at the end of the borrow inside annotations - postconditions of traversal functions or loop invariants.

To this effect, it is possible to refer to the value of a local borrower or a borrowed expression at the end of the borrow using a ghost identity function annotated with `At_End_Borrow`. Calls to these functions are interpreted by the tool as markers of references to values at the end of the borrow:

```
function At_End_Borrow (E : access constant List_Acc) return access constant List_Acc is
  (E)
with Ghost,
  Annotate => (GNATprove, At_End_Borrow);
```

Note that the name of the function could be something other than `At_End_Borrow`. GNATprove will check that a function associated with the `At_End_Borrow` annotation is a ghost expression function which takes a single parameter of an access-to-constant type and returns it. `At_End_Borrow` functions can only be called inside regular assertions or contracts, within a parameter of a call to a lemma subprogram, or within the initialization of a (ghost) constant of anonymous access-to-constant type.

When GNATprove encounters a call to such a function, it checks that the actual parameter of the call is either a local borrower or an expression which is borrowed in the current scope. It does not interpret it as the current value of the expression, but rather as what is usually called a *prophecy variable* in the literature, namely, an imprecise value representing the value that the expression will have at the end of the borrow. As GNATprove does not do any look-ahead, nothing will be known about the actual value of a local borrower at the end of the borrow. However, the tool will still be aware of the relation between this final value and the final value of the expression it borrows. As an example,

consider a local borrower `Y` of the list `X` as defined above. The `At_End_Borrow` function can be used to give properties that are known to hold during the scope of `Y`. For example, since `Y` and `X` designate the same value, GNATprove can verify that no matter what happens during the scope of `Y`, at the end of the borrow, the `Val` component of `X` will be the `Val` component of `Y`:

```
pragma Assert (At_End_Borrow (X).Val = At_End_Borrow (Y).Val);
```

However, even though at the beginning of the declare block, the first value of `X` is 1, it is not correct to assert that it will necessarily be so at the end of the borrow:

```
pragma Assert (Y.Val = 1);                 --  proved
pragma Assert (At_End_Borrow (X).Val = 1); --  incorrect
```

Indeed, `Y` could be modified later so that `X.Val` is not 1 anymore:

```
declare
   Y : access List := X;
begin
   Y.Val := 2;
end;
pragma Assert (X.Val = 2);
```

Note that the assertion above is invalid even if `Y.Val` is *not* modified in the following statements. It needs to be provable only from the information available at the assertion point, not knowing what will actually happen later in the scope of the borrow. The analysis performed by GNATprove only considers those statements that occur before the assertion to be proved; GNATprove does not consider statements that occur later in the control flow. In other words, there is no lookahead when an assertion is to be proved.

Note: Since `At_End_Borrow` functions are identity functions, the current values of the borrower and borrowed expression are used when executing assertions containing prophecy variables. This is sound. Indeed, GNATprove will show that the assertion holds for all possible modifications of the borrower. As not modifying the borrower is a valid senario, this is enough to ensure that the assertion necessarily evaluates to True at runtime.

Let us now consider a case where `X` is not borrowed completely. In the declaration of `Y`, we can decide to borrow only the last three elements of the list:

```
declare
   Y : access List := X.Next.Next;
begin
   pragma Assert (At_End_Borrow (X.Next.Next).Val = At_End_Borrow (Y).Val);
   pragma Assert (At_End_Borrow (X.Next.Next) /= null);
   -- Proved, this follows from the relationship between X and Y

   pragma Assert (At_End_Borrow (X.Next.Next.Val) = 3);
   -- Incorrect, X could be modified through Y

   X.Val := 42;
end;
```

Here, like in the previous example, we can state that, at the end of the borrow, `X.Next.Next.Val` is `Y.Val`, and then `X.Next.Next` cannot be set to null. We cannot assume anything about the part of `X` designated by `Y`, so we won't be able to prove that `X.Next.Next.Val` will remain 3. Note that we cannot get the value at the end of the borrow of an expression which is not borrowed in the current scope. Here, even if `X.Next.Next` is borrowed, `X` and `X.Next` are not. As a result, calls to `At_End_Borrow` on them will be rejected by the tool.

Inside the scope of `Y`, it is possible to modify the variable `Y` itself, as opposed to modifying the structure it designates, so that it gives access to a subcomponent of the borrowed structure. It is called a *reborrow*. During a reborrow, the part of the structure designated by the borrower is reduced, so the prophecy variable giving the value of the borrower at the end of the borrow is reduced as well. The part of the borrowed expression which is no longer accessible through the borrower cannot be modified anymore for the rest of the borrow. It is said to be *frozen* and its final value is known. For example, let's use `Y` to borrow `X` entirely and then modify it to only designate `X.Next.Next`:

```
declare
   Y : access List := X;
begin
   Y := Y.Next.Next; -- reborrow

   pragma Assert (At_End_Borrow (X).Next.Next /= null);
   pragma Assert (At_End_Borrow (X).Val = 1);
   pragma Assert (At_End_Borrow (X).Next.Val = 2);
   pragma Assert (At_End_Borrow (X).Next.Next.Val = 3);       --  incorrect
   pragma Assert (At_End_Borrow (X).Next.Next.Next /= null); --  incorrect
end;
```

After the reborrow, the part of `X` still accessible from the borrower is reduced, but since `X` was borrowed entirely to begin with, the ownership policy of SPARK still forbids direct access to any components of `X` while in the scope of `Y`. As a result, we have more information about the final value of `X` than in the previous case. We still know that `X` will hold at least three elements, that is `X.Next.Next /= null`. Additionally, the first and second components of `X` are no longer accessible from `Y`, and since they cannot be accessed directly through `X`, we know that they will keep their current values. This is why we can now assert that, at the end of the borrow, `X.Val` is 1 and `X.Next.Val` is 2. However, we still cannot know anything about the part of `X` still accessible from `Y` as these properties could be modified later in the borrow:

```
Y.Val := 42;
Y.Next := null;
```

During sequences of re-borrows, it is additionally possible to use constants of anonymous access-to-constant type in order to save prophecy variables for intermediate values of an access variable, as in the following example:

```
declare
   Y : access List := X;
begin
   Y := Y.Next; --  first reborrow
   declare
      Z : constant access constant List := At_End_Borrow (Y) with Ghost;
      --  At_End_Borrow is Ghost, so Z too.
   begin
      Y := Y.Next; -- second reborrow
      pragma Assert (At_End_Borrow (X).Next.Val = Z.Val);
      --  Z match prophecy of first reborrow of Y
      pragma Assert (Z.Next.Val = At_End_Borrow (Y).Val);
   end
end
```

As `Z` serves to save a prophecy variable, it is subject to the same usage restrictions as the corresponding `At_End_Borrow (Y)` call, in place of the usual rules for local observers.

As said earlier, in general, GNATprove can handle local borrows without any additional user written annotations. Therefore, `At_End_Borrow` functions are mostly useful at places where information is lost by the tool: in postconditions of borrowing traversal functions (which return a local borrower of their first parameter) and in loop invariants if the

loop involves a reborrow (in this case the value of the borrower at the end of the borrow is modified inside the loop and needs to be described in the invariant). Let us consider the following example:

```
with SPARK.Big_Integers;  use SPARK.Big_Integers;
with SPARK.Big_Intervals; use SPARK.Big_Intervals;
with Lists;               use Lists;

procedure List_Borrows with SPARK_Mode is

   function Length (L : access constant List) return Big_Natural is
     (if L = null then 0 else Length (L.Next) + 1)
   with Subprogram_Variant => (Structural => L);

   function Get (L : access constant List; P : Big_Positive) return Integer is
     (if P = Length (L) then L.Val else Get (L.Next, P))
   with Subprogram_Variant => (Structural => L),
     Pre => P <= Length (L);

   function Eq (L, R : access constant List) return Boolean is
     (Length (L) = Length (R)
      and then (for all P in Interval'(1, Length (L)) => Get (L, P) = Get (R, P)))
   with Annotate => (GNATprove, Inline_For_Proof);

   function Tail (L : access List) return access List with
     Contract_Cases =>
       (L = null =>
          Tail'Result = null and At_End_Borrow (L) = null,
        others   => At_End_Borrow (L).Val = L.Val
          and Eq (L.Next, Tail'Result)
          and Eq (At_End_Borrow (L).Next, At_End_Borrow (Tail'Result)));
   --  No matter what is done later with the result of Tail, at the end of the
   --  borrow L.Val will be the current value of L.Val and the remainder of the
   --  list will be the (updated) value of Tail'Result.

   function Tail (L : access List) return access List is
   begin
      if L = null then
         return null;
      else
         return L.Next;
      end if;
   end Tail;

   procedure Set_All_To_Zero (L : access List) with
     Post => Length (L) = Length (L)'Old
     and then (for all P in Interval'(1, Length (L)) => Get (L, P) = 0);

   procedure Set_All_To_Zero (L : access List) is
      X : access List := L;
      C : Big_Natural := 0 with Ghost;
      --  Number of traversed elements

   begin
```

(continues on next page)

(continued from previous page)

```
      while X /= null loop
         pragma Loop_Invariant (C = Length (X)'Loop_Entry - Length (X));
         --  C is the number of traversed elements
         pragma Loop_Invariant
           (Length (At_End_Borrow (L)) = C + Length (At_End_Borrow (X)));
         --  At the end of the borrow, L will have C more elements than X
         pragma Loop_Invariant
           (for all P in Interval'(1, Length (At_End_Borrow (L))) =>
                (if P <= Length (At_End_Borrow (X))
                 then Get (At_End_Borrow (L), P) = Get (At_End_Borrow (X), P)
                 else Get (At_End_Borrow (L), P) = 0));
         --  At the end of the borrow, the tail of L will be X and the rest
         --  will contain zeros.

         X.Val := 0;
         X := X.Next; --  Reborrow
         C := C + 1;
      end loop;
   end Set_All_To_Zero;

   X : List_Acc :=
     new List'(1, new List'(2, new List'(3, new List'(4, new List'(5, null)))));
begin
   declare
      Y : access List := Tail (X.Next);
   begin
      Y.Val := 42;
   end;

   pragma Assert (X.Val = 1);
   pragma Assert (X.Next.Val = 2);
   pragma Assert (X.Next.Next.Val = 42);
   pragma Assert (X.Next.Next.Next.Val = 4);
end List_Borrows;
```

The function `Tail` is a borrowing traversal function. It returns a local borrower of its parameter `L`. As GNATprove works modularly on a per subprogram basis, it is necessary to specify in its postcondition how the value of the borrowed parameter `L` can be reconstructed from the value of the borrower `Tail'Result` at the end of the borrow. Otherwise, GNATprove would not be able to recompute the value of the borrowed parameter after the returned borrower goes out of scope in the caller.

The `Tail` function returns the `Next` component of a list if there is one, and `null` otherwise. As pointer equality is not allowed in SPARK, we define our own equality function `Eq` which compares the elements of the list one by one. Note that the `Get` function indexes the list from the end (the first element of the list is accessed by `Get (L, Length (L))`). This is done to avoid arithmetic in the recursive definition of `Get` as it slows the proofs down. In the postcondition of `Tail`, we consider the two cases, and, in each case, specify both the value returned by the function and how the parameter `L` is related to the returned borrower:

- If `L` is `null` then `Tail` returns `null` and `L` will stay `null` for the duration of the borrow.
- Otherwise, `Tail` returns `L.Next`, the first element of `L` will stay as it was at the time of call, and the rest of `L` stays equal to the object returned by `Tail`.

Thanks to this postcondition, GNATprove can verify a program which borrows a part of `L` using the `Tail` function and modifies `L` through this borrower, as can be seen in the body of `List_Borrows`.

Postconditions of borrowing traversal functions systematically need to provide two properties: one specifying the result, and another specifying how the parameter is related to the borrower. This is generally redundant, as by nature the parameter/borrower relation always holds at the point of return of the function. For example, on the post of `Tail`, `Eq (L.Next, Tail'Result)` repeats `Eq (At_End_Borrow (L).Next, At_End_Borrow (Tail'Result))`.

The tool limits that redundancy by letting the user write only the parameter/borrower relation. Properties of the result are automatically derived by duplicating the postcondition, with calls to `At_End_Borrow` replaced by their arguments. This covers most (if not all) properties of the result, and additional properties of the result can be explicitly written if needed. This means we get equivalent behavior for function `Tail` by removing the second conjunct of the postcondition.

`At_End_Borrow` functions are also useful to write loop invariants in loops involving reborrows. This is exemplified in the `Set_All_To_Zero` procedure which traverses a list and sets all its elements to 0. The variable `X` borrows the whole input list `L` at the beginning of the function. Inside the loop, `X` is used to modify the structure designated by `L`. At the end of the procedure, ownership is transferred back to `L` automatically. To prove the postcondition of `Set_All_To_Zero`, GNATprove needs to know precisely how to reconstruct the value of `L` at this point. As `X` is reborrowed in the loop, the relation between its value and the value of `L` at the end of the borrow (the end of the scope of `X`) changes at each iteration. At the beginning of the loop, `X` is an alias of `L`, so the value designated by `L` is equal to the value designated by `X` at the end of the borrow. At each iteration, an element is dropped from `X`, so the value designated by the current value of `X` at the end of the borrow shrinks. At the same time, we get more information about the value designated by `L` at the end of the borrow as more and more elements are *frozen* and therefore definitely set to their current value, that is, 0.

Because proof uses cutpoints to reason about loops, it is necessary to supply all this information in a loop invariant. This is what is done in the body of `Set_All_To_Zero`. To help readability, a ghost variable `C` is introduced to count the number of iterations in the loop. The first invariant is a regular invariant, it maintains the value of `C` at each iteration. The two following ones are used to describe how `L` can be reconstructed from `X` at the end of the borrow: `L` will be made of `C` zeros followed by the final value of `X`. Note that, in the invariant, no assumption is made about the changes that can be made to `X` during the rest of the borrow, there is no look ahead. Both `Tail` and `Set_All_To_Zero` can be entirely verified by GNATprove:

```
1  list_borrows.adb:7:13: info: implicit aspect Always_Terminates on "Length" has been␣
   ↪proved, subprogram will terminate
2  list_borrows.adb:8:24: info: predicate check proved
3  list_borrows.adb:8:31: info: subprogram variant proved
4  list_borrows.adb:8:31: info: predicate check proved
5  list_borrows.adb:8:40: info: pointer dereference check proved
6  list_borrows.adb:8:47: info: predicate check proved
7  list_borrows.adb:8:49: info: predicate check proved
8  list_borrows.adb:11:13: info: implicit aspect Always_Terminates on "Get" has been proved,
   ↪ subprogram will terminate
9  list_borrows.adb:12:10: info: predicate check proved
10 list_borrows.adb:12:14: info: predicate check proved
11 list_borrows.adb:12:31: info: pointer dereference check proved
12 list_borrows.adb:12:41: info: subprogram variant proved
13 list_borrows.adb:12:41: info: precondition proved
14 list_borrows.adb:12:47: info: pointer dereference check proved
15 list_borrows.adb:14:13: info: predicate check proved
16 list_borrows.adb:14:18: info: predicate check proved
17 list_borrows.adb:16:13: info: implicit aspect Always_Terminates on "Eq" has been proved,␣
   ↪subprogram will terminate
18 list_borrows.adb:17:07: info: predicate check proved
19 list_borrows.adb:17:20: info: predicate check proved
20 list_borrows.adb:18:58: info: precondition proved
21 list_borrows.adb:18:66: info: predicate check proved
22 list_borrows.adb:18:71: info: precondition proved
```

(continues on next page)

(continued from previous page)

```
list_borrows.adb:18:79: info: predicate check proved
list_borrows.adb:21:13: info: implicit aspect Always_Terminates on "Tail" has been␣
↪proved, subprogram will terminate
list_borrows.adb:23:18: info: contract case proved
list_borrows.adb:25:18: info: contract case proved
list_borrows.adb:25:38: info: pointer dereference check proved
list_borrows.adb:25:46: info: pointer dereference check proved
list_borrows.adb:26:20: info: pointer dereference check proved
list_borrows.adb:27:36: info: pointer dereference check proved
list_borrows.adb:37:18: info: pointer dereference check proved
list_borrows.adb:42:14: info: predicate check proved
list_borrows.adb:42:14: info: postcondition proved
list_borrows.adb:42:37: info: predicate check proved
list_borrows.adb:43:57: info: precondition proved
list_borrows.adb:43:65: info: predicate check proved
list_borrows.adb:47:26: info: predicate check proved
list_borrows.adb:52:33: info: predicate check proved
list_borrows.adb:52:33: info: loop invariant preservation proved
list_borrows.adb:52:33: info: loop invariant initialization proved
list_borrows.adb:52:47: info: predicate check proved
list_borrows.adb:52:61: info: predicate check proved
list_borrows.adb:55:13: info: predicate check proved
list_borrows.adb:55:13: info: loop invariant initialization proved
list_borrows.adb:55:13: info: loop invariant preservation proved
list_borrows.adb:55:42: info: predicate check proved
list_borrows.adb:55:46: info: predicate check proved
list_borrows.adb:58:13: info: loop invariant initialization proved
list_borrows.adb:58:13: info: loop invariant preservation proved
list_borrows.adb:59:21: info: predicate check proved
list_borrows.adb:59:26: info: predicate check proved
list_borrows.adb:60:23: info: precondition proved
list_borrows.adb:60:47: info: predicate check proved
list_borrows.adb:60:52: info: precondition proved
list_borrows.adb:60:76: info: predicate check proved
list_borrows.adb:61:23: info: precondition proved
list_borrows.adb:61:47: info: predicate check proved
list_borrows.adb:65:11: info: pointer dereference check proved
list_borrows.adb:66:16: info: pointer dereference check proved
list_borrows.adb:67:15: info: predicate check proved
list_borrows.adb:67:17: info: predicate check proved
list_borrows.adb:67:19: info: predicate check proved

list_borrows.adb:71:04: medium: resource or memory leak might occur at end of scope
   71>|   X : List_Acc :=
   72 |     new List'(1, new List'(2, new List'(3, new List'(4, new List'(5, null)))));
list_borrows.adb:75:33: info: pointer dereference check proved
  in reconstructed value at the end of the borrow at list_borrows.adb:75
list_borrows.adb:77:08: info: pointer dereference check proved
list_borrows.adb:80:19: info: assertion proved
list_borrows.adb:80:20: info: pointer dereference check proved
list_borrows.adb:81:19: info: assertion proved
list_borrows.adb:81:20: info: pointer dereference check proved
```

(continues on next page)

(continued from previous page)

```
list_borrows.adb:81:25: info: pointer dereference check proved
list_borrows.adb:82:19: info: assertion proved
list_borrows.adb:82:20: info: pointer dereference check proved
list_borrows.adb:82:25: info: pointer dereference check proved
list_borrows.adb:82:30: info: pointer dereference check proved
list_borrows.adb:83:19: info: assertion proved
list_borrows.adb:83:20: info: pointer dereference check proved
list_borrows.adb:83:25: info: pointer dereference check proved
list_borrows.adb:83:30: info: pointer dereference check proved
list_borrows.adb:83:35: info: pointer dereference check proved
```

E.7 Annotation for Accessing the Logical Equality for a Type

In Ada, the equality is not the logical equality in general. In particular, arrays are considered to be equal if they contain the same elements, even with different bounds, +0.0 and -0.0 are considered equal...

It is possible to use a `pragma Annotate (GNATprove, Logical_Equal)` to ask GNATprove to interpret a function with an equality profile as the logical equality for the type. If the function body is visible in SPARK, a check will be emitted to ensure that it indeed checks for logical equality as understood by the proof engine (which depends on the underlying model used by the tool, see below). It comes in handy for example to express that a (part of a) data-structure is left unchanged by a procedure, as is done in the example below:

```
subtype Length is Natural range 0 .. 100;
type Word (L : Length := 0) is record
   Value : String (1 .. L);
end record;
function Real_Eq (W1, W2 : String) return Boolean with
  Ghost,
  Import,
  Annotate => (GNATprove, Logical_Equal);
type Dictionary is array (Positive range <>) of Word;

procedure Set (D : in out Dictionary; I : Positive; W : String) with
  Pre  => I in D'Range and W'Length <= 100,
  Post => D (I).Value = W
  and then (for all J in D'Range =>
              (if I /= J then Real_Eq (D (J).Value, D'Old (J).Value)))
is
begin
   D (I) := (L => W'Length, Value => W);
end Set;
```

The actual interpretation of logical equality is highly dependent on the underlying model used for formal verification. However, the following properties are always valid, no matter the actual type on which a logical equality function applies:

- Logical equality functions are equivalence relations, that is, they are reflexive, symmetric, and transitive. This implies that such functions can always be used to express that something is preserved as done above.
- There is no way to tell the difference between between two values which are logically equal. Said otherwise, all SPARK compatible functions will return the same result on logically equal inputs. Note that Ada functions which do not follow SPARK assumptions may not, for example, if they compare the address of pointers which

are not modelled by GNATprove. Such functions should never be used inside SPARK code as they can introduce soundness issues.

- Logical equality is handled efficiently, in a builtin way, by the underlying solvers. This is different from the regular Ada equality which is basically handled as a function call, using potentially complex defining axioms (in particular Ada equality over arrays involves quantifiers).

Note: In Ada, copying an expression might not necessarily return a logically equal value. This happens for example when a value is assigned into a variable or returned by a function. Indeed, such copies might involve for example sliding (for arrays) or a partial copy (for view conversions of tagged types).

It might happen that the underlying model of values of an Ada type contain components which are not present in the Ada value. This makes it impossible to implement a comparison function in Ada which would compute logical equality on such types. This is the case in particular for arrays and discriminated records with variant parts. More precisely, logical equality can be implemented using the regular Ada equality for discrete types and fixed point types. For floating point types, both the value and the sign need to be compared (to tell the difference between +0.0 and -0.0). For pointers, as the address is not modeled in SPARK, it is enough to check for nullity and compare the designated value. Logical equality on untagged record with no variant parts can be achieved by comparing the components. For other composite types, it cannot be implemented and has to remain non-executable as in the example above. Additionally, a user can safely annotate a comparison function on private types whose full view is not in SPARK using the `Logical_Equal` annotation if it behaves as expected, namely, it is an equivalence relation, and there is no way, using the API provided in the public part of the enclosing package, to tell the difference between two values on which this function returns True.

Note: For private types whose full view is not in SPARK, GNATprove will peek inside the full view to try and determine whether or not to interpret the primitive equality on such types as the logical equality.

Note that, for types on which implementing the logical equality in Ada is impossible, GNATprove might not be able to prove that two Ada values are logically equal even if there is no way to tell the difference in SPARK. For example, it might not be able to prove that two arrays are logically equal even if they have the same bounds and the same value. It is because it is not necessarily true in the underlying model, where values outside of the array bounds are represented. Therefore, using an assumption to state that two objects which are equal-in-Ada are logically equal might introduce an unsoundness, in particular in the presence of slices. It is demonstrated in the example below where GNATprove can prove that two strings are not logically equal even though they have the same bounds and the same elements. However, logical equality can be used safely as long as everything is proved correct (no assumption is used).

```
procedure Test is
   S1 : constant String := "foo1";
   S2 : constant String := "foo2";

begin
   pragma Assert (S1 (1 .. 3) = S2 (1 .. 3));
   pragma Assert (not Real_Eq (S1 (1 .. 3), S2 (1 .. 3)));
end Test;
```

E.8 Annotation for Enforcing Ownership Checking on a Private Type

Private types whose full view is not in SPARK can be annotated with a `pragma Annotate (GNATprove, Ownership, ...)`. Such a type is handled by GNATprove in accordance to the *Memory Ownership Policy* of SPARK. For example, the type `T` declared in the procedure `Simple_Ownership` below has an ownership annotation. As a result, GNATprove will reject the second call to `Read` in the body of `Simple_Ownership`, because the value designated by `X` has been moved by the assignment to `Y`.

```
procedure Simple_Ownership with SPARK_Mode is
   package Nested is
      type T is private with
        Default_Initial_Condition,
        Annotate => (GNATprove, Ownership);

      function Read (X : T) return Boolean;
   private
      pragma SPARK_Mode (Off);
      type T is null record;
      function Read (X : T) return Boolean is (True);
   end Nested;
   use Nested;

   X : T;
   Y : T;

begin
   pragma Assert (Read (X)); -- OK

   Y := X;

   pragma Assert (Read (X)); -- Error, X has been moved
end Simple_Ownership;
```

In addition, it is possible to state that a type needs reclamation with a `pragma Annotate (GNATprove, Onwership, "Needs_Reclamation", ...)`. In this case, GNATprove emits checks to ensure that the resource is not leaked. For these checks to be handled precisely, the user should annotate a function taking a value of this type as a parameter and returning a boolean with a `pragma Annotate (GNATprove, Onwership, "Needs_Reclamation", ...)` or `pragma Annotate (GNATprove, Onwership, "Is_Reclaimed", ...)`. This function will be used to check that the resource cannot be leaked. A function annotated with `"Needs_Reclamation"` shall return True when its input holds some resource while a function annotated with `"Is_Reclaimed"` shall return True when its input has already been reclaimed. Only one such function shall be provided for a given type. Two examples of use are given below.

```
package Hidden_Pointers with
  SPARK_Mode,
  Always_Terminates
is
   type Pool_Specific_Access is private with
     Default_Initial_Condition => Is_Null (Pool_Specific_Access),
     Annotate => (GNATprove, Ownership, "Needs_Reclamation");

   function Is_Null (A : Pool_Specific_Access) return Boolean with
     Global => null,
     Annotate => (GNATprove, Ownership, "Is_Reclaimed");
```

(continues on next page)

(continued from previous page)

```
   function Deref (A : Pool_Specific_Access) return Integer with
     Global => null,
     Pre => not Is_Null (A);

   function Allocate (X : Integer) return Pool_Specific_Access with
     Global => null,
     Post => not Is_Null (Allocate'Result) and then Deref (Allocate'Result) = X;
   procedure Deallocate (A : in out Pool_Specific_Access) with
     Global => null,
     Post => Is_Null (A);

private
   pragma SPARK_Mode (Off);
   type Pool_Specific_Access is access Integer;
end Hidden_Pointers;
```

The package `Hidden_Pointers` defines a type `Pool_Specific_Access` which is really a pool specific access type. The ownership annotation on `Pool_Specific_Access` instructs GNATprove that objects of this type should abide by the *Memory Ownership Policy* of SPARK. It also states that the type needs reclamation when a value is finalized. Because of the annotation on the `Is_Null` function, GNATprove will attempt to prove that `Is_Null` returns True when an object of type `Pool_Specific_Access` is finalized unless it was moved.

The mechanism can also be used for resources which are not linked to memory management. For example, the package `Text_IO` defines a limited type `File_Descriptor` and uses ownership annotations to force GNATprove to verify that all file descriptors are closed before being finalized.

```
package Text_IO with
  SPARK_Mode,
  Always_Terminates
is
   type File_Descriptor is limited private with
     Default_Initial_Condition => not Is_Open (File_Descriptor),
     Annotate => (GNATprove, Ownership, "Needs_Reclamation");

   function Is_Open (F : File_Descriptor) return Boolean with
     Global => null,
     Annotate => (GNATprove, Ownership, "Needs_Reclamation");
   function Read_Line (F : File_Descriptor) return String with
     Global => null,
     Pre => Is_Open (F);

   function Open (N : String) return File_Descriptor with
     Global => null,
     Post => Is_Open (Open'Result);
   procedure Close (F : in out File_Descriptor) with
     Global => null,
     Post => not Is_Open (F);

private
   pragma SPARK_Mode (Off);
   type Text;
   type File_Descriptor is access all Text;
end Text_IO;
```

E.9 Annotation for Instantiating Lemma Procedures Automatically

As featured in *Manual Proof Using User Lemmas*, it is possible to write lemmas in SPARK as ghost procedures. However, actual calls to the procedure need to be manually inserted in the program whenever an instance of the lemma is necessary. It is possible to use a pragma `Annotate` to instruct GNATprove that an axiom should be introduced for a lemma procedure so manual instantiations are no longer necessary. This annotation is called `Automatic_Instantiation`. As an example, the `Equivalent` function below is an equivalence relation. This is expressed using three lemma procedures which should be instantiated automatically:

```
function Equivalent (X, Y : Integer) return Boolean with
  Global => null;

procedure Lemma_Reflexive (X : Integer) with
  Ghost,
  Global => null,
  Annotate => (GNATprove, Automatic_Instantiation),
  Post => Equivalent (X, X);

procedure Lemma_Symmetric (X, Y : Integer) with
  Ghost,
  Global => null,
  Annotate => (GNATprove, Automatic_Instantiation),
  Pre  => Equivalent (X, Y),
  Post => Equivalent (Y, X);

procedure Lemma_Transitive (X, Y, Z : Integer) with
  Ghost,
  Global => null,
  Annotate => (GNATprove, Automatic_Instantiation),
  Pre  => Equivalent (X, Y) and Equivalent (Y, Z),
  Post => Equivalent (X, Z);
```

Such lemmas should be declared directly after a function declaration, here the `Equivalent` function. The axiom will only be available when the associated function is used in the proof context.

E.10 Annotation for Hiding Information

By default, when verifying a part of a program, GNATprove makes all information about used entities available in the context. For example, it assumes values of global constants, postconditions of called subprograms, bodies of expression functions... While in general this behavior is desirable, it might result in untractable proof contexts on large programs. It is possible to use an annotation to manually add or remove information from the proof context. For the time being, only bodies of expression functions can be handled in this way.

Information hiding is decided at the level of an entity for which checks are generated, namely, a subprogram or entry, or the elaboration of a package. It cannot be refined in a smaller scope. To state that the body of an expression function should be hidden when verifying an entity `E`, a pragma with the `Hide_Info` annotation should be used either at the begining of the body of `E` or just after its specification or body. In the following example, the body of the expression function `F` is hidden when verifying its callers, making it impossible to prove their postconditions:

```
function F (X, Y : Boolean) return Boolean is (X and Y);

function Call_F (X, Y : Boolean) return Boolean is
```

(continues on next page)

(continued from previous page)

```
  (F (X, Y))
with Post => Call_F'Result = (X and Y); --  Unprovable, F is hidden
pragma Annotate (GNATprove, Hide_Info, "Expression_Function_Body", F);

function Call_F_2 (X, Y : Boolean) return Boolean with
  Post => Call_F_2'Result = (X and Y); --  Unprovable, F is hidden

function Call_F_2 (X, Y : Boolean) return Boolean is
begin
   return F (X, Y);
end Call_F_2;
pragma Annotate (GNATprove, Hide_Info, "Expression_Function_Body", F);

function Call_F_3 (X, Y : Boolean) return Boolean with
  Post => Call_F_3'Result = (X and Y); --  Unprovable, F is hidden

function Call_F_3 (X, Y : Boolean) return Boolean is
   pragma Annotate (GNATprove, Hide_Info, "Expression_Function_Body", F);
begin
   return F (X, Y);
end Call_F_3;
```

It is also possible to hide information by default and then use an annotation to disclose it when needed. A `Hide_Info` annotation located on the entity which is hidden is considered to provide a default. For example, the body of the expression function `G` is hidden by default. The postcondition of its caller `Call_G` cannot be proved.

```
function G (X, Y : Boolean) return Boolean is (X and Y) with
  Annotate => (GNATprove, Hide_Info, "Expression_Function_Body");
--  G is hidden by default

function Call_G (X, Y : Boolean) return Boolean is
  (G (X, Y))
with Post => Call_G'Result = (X and Y); --  Unprovable, G is hidden
```

When information is hidden by default, it is possible to disclose it while verifying an entity using the `Unhide_Info` annotation. This allows proving the `Call_G_2` function below:

```
function Call_G_2 (X, Y : Boolean) return Boolean is
  (G (X, Y))
with Post => Call_G_2'Result = (X and Y); --  Provable, G is no longer hidden
pragma Annotate (GNATprove, Unhide_Info, "Expression_Function_Body", G);
```

Remark that, when information is hidden by default, it is even hidden during the verification of the entity whose information we are hiding. For example, when verifying a recursive expression function whose body is hidden by default, the body of recursive calls is not available. If necessary, it can be disclosed using an `Unhide_Info` annotation:

```
--  Rec_F is hidden for its recursive calls

function Rec_F (X, Y : Boolean) return Boolean is
  (if not X then False elsif X = Y then True else Rec_F (Y, X))
    with
      Subprogram_Variant => (Decreases => X),
      Post => (if X then Rec_F'Result = Y), --  Unprovable, Rec_F is hidden
```

(continues on next page)

(continued from previous page)

```
      Annotate => (GNATprove, Hide_Info, "Expression_Function_Body");

--  The second annotation overrides the default for recursive calls

function Rec_F_2 (X, Y : Boolean) return Boolean is
  (if not X then False elsif X = Y then True else Rec_F_2 (Y, X))
    with
      Subprogram_Variant => (Decreases => X),
      Post => (if X then Rec_F_2'Result = Y), --  Provable, Rec_F_2 is visible
      Annotate => (GNATprove, Hide_Info, "Expression_Function_Body");
pragma Annotate (GNATprove, Unhide_Info, "Expression_Function_Body", Rec_F_2);
```

E.11 Annotation for Handling Specially Higher Order Functions

Functions for higher order programming can be expressed using parameters of an anonymous access-to-function type. As an example, here is a function `Map` returning a new array whose elements are the result of the application a function `F` to the elements of its input array parameter:

```
function Map
  (A : Nat_Array;
   F : not null access function (N : Natural) return Natural)
   return Nat_Array
with
  Post => Map'Result'First = A'First and Map'Result'Last = A'Last
    and (for all I in A'Range => Map'Result (I) = F (A (I)));
```

In a functional programming style, these functions are often called with an access to a local function as a parameter. Unfortunately, as GNATprove generally handles access-to-subprograms using refinement semantics, it is not possible to use a function accessing global data as an actual for an anonymous access-to-function parameter (see *Subprogram Pointers*). To workaround this limitation, it is possible to annotate the function `Map` with a pragma or aspect `Annotate => (GNATprove, Higher_Order_Specialization)`. When such a function is called on a reference to the `Access` attribute of a function, it will benefit from a partial exemption from the checks usually performed on the creation of such an access. Namely, the function will be allowed to access data, and it might even have a precondition if it can be proved to always hold at the point of call. We say that such a call is *specialized* for a particular value of its access-to-function parameter. As an example, consider the function `Divide_All` defined below. As the function `Map` is annotated with `Higher_Order_Specialization`, it can be called on the function `Divide`, even though it accesses some global data (the parameter `N`) and has a precondition.

```
function Map
  (A : Nat_Array;
   F : not null access function (N : Natural) return Natural)
   return Nat_Array
with
  Annotate => (GNATprove, Higher_Order_Specialization),
  Post => Map'Result'First = A'First and Map'Result'Last = A'Last
    and (for all I in A'Range => Map'Result (I) = F (A (I)));

function Divide_All (A : Nat_Array; N : Natural) return Nat_Array with
  Pre  => N /= 0,
  Post => (for all I in A'Range => Divide_All'Result (I) = A (I) / N);
```

(continues on next page)

(continued from previous page)

```
function Divide_All (A : Nat_Array; N : Natural) return Nat_Array is
   function Divide (E : Natural) return Natural is
     (E / N)
   with Pre => N /= 0;
begin
   return Map (A, Divide'Access);
end Divide_All;
```

Note: It might not be possible to annotate a function with `Higher_Order_Specialization` if the access value of its parameter is used in the contract of the function, as opposed to only its dereference. This happens in particular if the parameter is used in a call to a function which does not have the `Higher_Order_Specialization` annotation.

Because the analysis done by GNATprove stays modular, the precondition of the referenced function has to be provable on all possible parameters (but the specialized access-to-function parameters) and not only on the ones on which it is actually called. For example, even if we know that the precondition of `Add` holds for all the elements of the input array `A`, there will still be a failed check on the call to `Map` in `Add_All` defined below. Indeed, GNATprove does not peek into the body of `Map` and therefore has no way to know that its parameter `F` is only called on elements of `A`.

```
function Add_All (A : Nat_Array; N : Natural) return Nat_Array with
  Pre  => (for all E of A => E <= Natural'Last - N),
  Post => (for all I in A'Range => Add_All'Result (I) = A (I) + N);

function Add_All (A : Nat_Array; N : Natural) return Nat_Array is
   function Add (E : Natural) return Natural is
     (E + N)
   with Pre => E <= Natural'Last - N;
begin
   return Map (A, Add'Access);
end Add_All;
```

To avoid this kind of issue, it is possible to write a function with no preconditions and a fallback value as done below. Remark that this does not prevent the tool from proving the postcondition of `Add_All`.

```
function Add_All (A : Nat_Array; N : Natural) return Nat_Array is
   function Add (E : Natural) return Natural is
     (if E <= Natural'Last - N then E + N else 0);
begin
   return Map (A, Add'Access);
end Add_All;
```

Note: Only the regular contract of functions is available on specialized calls. Bodies of expression functions and refined postconditions will be ignored.

The `Higher_Order_Specialization` annotation can also be supplied on a lemma procedure (see *Manual Proof Using User Lemmas*). If this procedure has an `Automatic_Instantiation` annotation (see *Annotation for Instantiating Lemma Procedures Automatically*) and its associated function also has an `Higher_Order_Specialization` annotation, the lemma will generally be instantiated automatically on specialized calls to the function. As an example, the function `Count` defined below returns the number of elements with a property in an array. The function `Count` is specified in a recursive way in its postcondition. The two lemmas `Lemma_Count_All` and `Lemma_Count_None` give

the value of `Count` when all the elements of `A` or no elements of `A` fulfill the property. They will be automatically instantiated on all specializations of `Count`.

```
function Count
  (A : Nat_Array;
   F : not null access function (N : Natural) return Boolean)
   return Natural
with
  Annotate => (GNATprove, Higher_Order_Specialization),
  Subprogram_Variant => (Decreases => A'Length),
  Post =>
    (if A'Length = 0 then Count'Result = 0
     else Count'Result =
         (if F (A (A'Last)) then 1 else 0) +
           Count (A (A'First .. A'Last - 1), F))
    and Count'Result <= A'Length;

procedure Lemma_Count_All
  (A : Nat_Array;
   F : not null access function (N : Natural) return Boolean)
with
  Ghost,
  Annotate => (GNATprove, Automatic_Instantiation),
  Annotate => (GNATprove, Higher_Order_Specialization),
  Pre  => (for all E of A => F (E)),
  Post => Count (A, F) = A'Length;

procedure Lemma_Count_None
  (A : Nat_Array;
   F : not null access function (N : Natural) return Boolean)
with
  Ghost,
  Annotate => (GNATprove, Automatic_Instantiation),
  Annotate => (GNATprove, Higher_Order_Specialization),
  Pre  => (for all E of A => not F (E)),
  Post => Count (A, F) = 0;
```

Note: It can happen that some lemmas cannot be specialized with their associated function. It is the case in particular if the lemma contains several calls to the function with different access-to-function parameters. In this case, a warning will be emitted on the lemma declaration.

E.12 Annotation for Handlers

Some programs define *handlers*, subprograms which might be called asynchronously to perform specific treatments, for example when an interrupt occurs. These handlers are often registered using access-to-subprogram types. In general, access-to-subprograms are not allowed to access or modify global data in SPARK. However, this is too constraining for handlers which tend to work by side-effects. To alleviate this limitation, it is possible to annotate an access-to-subprogram type with a pragma or aspect `Annotate => (GNATprove, Handler)`. This instructs GNATprove that these access-to-subprograms will be called asynchronously. It is possible to create a value of such a type using a reference to the `Access` attribute on a subprogram accessing or modifying global data, but only when this global data

is synchronized (see SPARK RM 9.1). However, GNATprove will make sure that the subprograms designated by objects of these types are never called from SPARK code, as it could result in a missing data dependency.

For example, consider the following procedure which resets to zero a global variable `Counter`:

```
Counter : Natural := 0 with Atomic;

procedure Reset is
begin
   Counter := 0;
end Reset;
```

It is not possible to store an access to `Reset` in a regular access-to-procedure type, as it has a global effect. However, it can be stored in a type annotated with an aspect `Annotate => (GNATprove, Handler)` like below:

```
type No_Param_Proc is access procedure with
  Annotate => (GNATprove, Handler);

P : No_Param_Proc := Reset'Access;
```

However, it is not possible to call `P` from SPARK code as the effect to the `Counter` variable would be missed.

Note: As handlers are called asynchronously, GNATprove does not allow providing pre and postconditions for the access-to-subprogram type. As a result, a check will be emitted to ensure that the precondition of each specific subprogram designated by these handlers is provable in the empty context.

E.13 Annotation for Container Aggregates

The `Container_Aggregates` annotation allows specifying aggregates on types annotated with the *Aspect Aggregate* either directly, using predefined aggregate patterns, or through a model. The second option is the easiest. It is enough to provide a model function returning a type which supports compatible aggregates. When an aggregate is encountered, GNATprove deduces the value of its model using the `Container_Aggregates` annotation on the model type. In particular, functional containers from the *SPARK Libraries* are annotated with the `Aggregate` aspect. Other container libraries can take advantage of this support to specify aggregates on their own library by providing a model function returning a functional container. As an example, sequences can be used as a model to provide aggregates for a linked list of integers:

```
1  with SPARK.Containers.Functional.Infinite_Sequences;
2
3  package Through_Model with SPARK_Mode is
4     package My_Seqs is new SPARK.Containers.Functional.Infinite_Sequences (Integer);
5     use My_Seqs;
6
7     type List is private with
8       Aggregate => (Empty => Empty_List, Add_Unnamed => Add),
9       Annotate  => (GNATprove, Container_Aggregates, "From_Model");
10
11    Empty_List : constant List;
12
13    function Model (L : List) return Sequence with
14      Subprogram_Variant => (Structural => L),
```

(continues on next page)

(continued from previous page)

```
     Annotate => (GNATprove, Container_Aggregates, "Model");

   procedure Add (L : in out List; E : Integer) with
     Always_Terminates,
     Post => Model (L) = Add (Model (L)'Old, E);

private

   type List_Cell;
   type List is access List_Cell;
   type List_Cell is record
      Data : Integer;
      Next : List;
   end record;

   function Model (L : List) return Sequence is
     (if L = null then Empty_Sequence
      else Add (Model (L.Next), L.Data));

   Empty_List : constant List := null;
end Through_Model;
```

The `Container_Aggregates` annotation on type `List` indicates that its aggregates are specified using a model function. The annotation on the `Model` function identifies it as the function that should be used to model aggregates of type `List`. It is then possible to use aggregates of type `List` in SPARK as in the `Main` procedure below. GNATprove will use the handling of aggregates on infinite sequences to determine the value of the model of the aggregate and use it to prove the program.

```
with Through_Model; use Through_Model;

procedure Main with SPARK_Mode is
   L : constant List := [1, 2, 3, 4, 5, 6];
begin
   pragma Assert (My_Seqs.Get (Model (L), 4) = 4);
end Main;
```

It is also possible to specify directly how aggregates should be handled, without going through a model. To do this, GNATprove provides three different predefined patterns: one for vectors and sequences, one for sets, and one for maps. The chosen pattern is specified inside the `Container_Aggregates` annotation on the type. Depending on this pattern, different functions can be provided to describe it. As an example, aggregates on our linked list of integers can also be described directly by supplying the three functions required for predefined sequence aggregates - `First`, `Last`, and `Get`:

```
with SPARK.Big_Integers;  use SPARK.Big_Integers;
with SPARK.Big_Intervals; use SPARK.Big_Intervals;

package Predefined with SPARK_Mode is
   pragma Unevaluated_Use_Of_Old (Allow);
   type List is private with
     Aggregate => (Empty => Empty_List, Add_Unnamed => Add),
     Annotate  => (GNATprove, Container_Aggregates, "Predefined_Sequences");

```

(continues on next page)

(continued from previous page)

```
   Empty_List : constant List;

   function First return Big_Positive is (1) with
     Annotate => (GNATprove, Container_Aggregates, "First");

   function Length (L : List) return Big_Natural with
     Subprogram_Variant => (Structural => L),
     Annotate => (GNATprove, Container_Aggregates, "Last");

   function Get (L : List; P : Big_Positive) return Integer with
     Pre => P <= Length (L),
     Subprogram_Variant => (Structural => L),
     Annotate => (GNATprove, Container_Aggregates, "Get");

   function Eq (L1, L2 : List) return Boolean is
     (Length (L1) = Length (L2)
      and then
        (for all I in Interval'(1, Length (L1)) =>
              Get (L1, I) = Get (L2, I)));

   function Copy (L : List) return List with
     Subprogram_Variant => (Structural => L),
     Post => Eq (L, Copy'Result);

   procedure Add (L : in out List; E : Integer) with
     Always_Terminates,
     Post => Length (L) = Length (L)'Old + 1
     and then Get (L, Length (L)) = E
     and then (for all I in Interval'(1, Length (L) - 1) =>
                 Get (L, I) = Get (Copy (L)'Old, I));

private
   type List_Cell;
   type List is access List_Cell;
   type List_Cell is record
      Data : Integer;
      Next : List;
   end record;

   function Length (L : List) return Big_Natural is
     (if L = null then 0 else 1 + Length (L.Next));

   function Get (L : List; P : Big_Positive) return Integer is
     (if P = Length (L) then L.Data else Get (L.Next, P));

   Empty_List : constant List := null;
end Predefined;
```

For all types annotated with `Container_Aggregates`, GNATprove performs *consistency checks* to make sure that the description induced by the annotation is compatible with the subprograms supplied by its `Aggregate` aspect. For example, on the `List` type defined above, GNATprove generates checks to ensure that `Length` returns `First - 1` on `Empty_List`. It also checks that calling `Add` increases the result of `Length` by one and that its parameter `E` is associated to this last position. These checks succeed on our example thanks to the postcondition of `Add` as made explicit by the

info messages on line 8:

```
predefined.ads:8:06: info: Container_Aggregates annotation proved
  when establishing invariant on Length at predefined.ads:15
  after a call to Empty_List at predefined.ads:10
predefined.ads:8:06: info: Container_Aggregates annotation proved
  when reestablishing invariant on Get at predefined.ads:19
  after a call to Add at predefined.ads:34
predefined.ads:8:06: info: Container_Aggregates annotation proved
  when reestablishing invariant on Length at predefined.ads:15
  after a call to Add at predefined.ads:34
```

The Container_Aggregates annotation might also induce *restrictions on aggregate usage*. For example, if we had chosen a signed integer type for the return type of Length in Predefined, GNATprove would have introduced checks on all aggregates of type List to make sure that the number of elements doesn't exceed the number of indexes.

When a model is used to specify aggregates, both the restrictions on aggregate usage and the consistency checks of the model are inherited. For example, GNATprove attempts to prove the consistency of Empty_List and Add from Through_Model with the functions supplied for the model. These checks are proved thanks to the precise postcondition on Add. In the same way, if we had chosen to use a functional vector indexed by a signed integer type instead of an infinite sequence as a model in Through_Model, we would have been limited by the size of the integer type for our aggregates.

All types with a Container_Aggregates annotation can be supplied an additional Capacity function. In general, this function does not have any parameters. It provides a constant bound on the number of elements allowed in the container. If a Capacity function is supplied on a type using models for its aggregates, it overrides the Capacity function of the model if any. If present, the Capacity function is used both as an additional restriction on aggregate usage and as an additional assumption for the consistency checks: GNATprove assumes that the container holds strictly less than Capacity elements before a call to Add and it makes sure that the number of elements in actual aggregates never exceeds the capacity.

To accomodate containers which can have a different capacity depending on the object, Ada allows providing an Empty function which takes an integer value for the capacity as a parameter in the Aggregate aspect. If a Capacity function is specified for such a type, it shall take the container as a parameter since the capacity is specific to each container. In this case, it is the upper bound of the parameter type of the Empty function which is used as a restriction on aggregate usage. Examples of containers with a capacity function (both with and without parameters) can be found in formal container packages in the *SPARK Libraries*.

There are three kinds of predefined aggregates patterns: Predefined_Sequences like in the Predefined package, Predefined_Sets, and Predefined_Maps. The selected pattern should be provided as a string to the Container_Aggregates annotation specified on the container type. Additional Container_Aggregates annotations are necessary on each specific function supported by the pattern. They also require a string encoding its intended use. This usage is examplified in the Predefined package.

The Predefined_Sequences pattern can be applied to containers with positional aggregates. It supports three kinds of function annotations:

- The function First returns the index or position that should be used for the first element in a container.
- The function Last returns the index or position of the last element in a specific container.
- The function Get returns the element associated to a valid index or position in the container.

All three functions are mandatory. The return types of First and Last should be subtypes of the same signed integer type, or possibly of Big_Integer.

The consistency checks ensure that:

- Empty returns a container C such that Last (C) = First - 1,

- Add can be called on a container C such that Last (C) < Index_Type'Last, and
- after a call to Add, the result of Last (C) has been incremented by one, the result of calling Get on all previous indexes is unchanged, and calling Get on the last index returns the added element.

The contracts of Empty and Append below ensure conformance to these consistency checks:

```
type T is private with
  Aggregate => (Empty       => Empty,
                Add_Unnamed => Append),
  Annotate => (GNATprove, Container_Aggregates, "Predefined_Sequences");

function Empty return T with
  Post => Last (Empty'Result) = First - 1;
procedure Append (X : in out T; E : Element_Type) with
  Always_Terminates,
  Pre  => Last (X) < Index_Type'Last,
  Post => Last (X) = Last (X)'Old + 1 and Get (X, Last (X)) = E
  and (for all I in First .. Last (X) - 1 => Get (X, I) = Get (X'Old, I));

function Last (X : T) return Ext_Index with
  Annotate => (GNATprove, Container_Aggregates, "Last");

function First return Index_Type with
  Annotate => (GNATprove, Container_Aggregates, "First");

function Get (X : T; I : Index_Type) return Element_Type with
  Annotate => (GNATprove, Container_Aggregates, "Get"),
  Pre => I <= Last (X);
```

If Last returns a signed integer type, there is a restriction on predefined sequence aggregates usage: GNATprove will make sure that the number of elements in an aggregate never exceeds the maximum value of the return type of Last.

When an aggregate C is encountered, GNATprove automatically infers that:

- Last (C) - First + 1 is the number of elements in the aggregate and
- Get (First + N - 1) returns a copy of the N'th element.

The Predefined_Sets pattern can be applied to containers with positional aggregates. It supports three kinds of function annotations:

- The Contains function returns True if and only if its element parameter is in the container.
- Equivalent_Elements is an equivalence relation such that Contains always returns the same value on two equivalent elements.
- The Length function returns the number of elements in the set.

Contains and Equivalent_Elements are mandatory, Length is optional. If it is supplied, the Length function should return a signed integer type or a subtype of Big_Integer.

The consistency checks ensure that:

- Contains returns False for all elements on the result of Empty and Length, if specified, returns 0,
- Add can be called on a container C and an element E such that Contains (C, E) returns False,
- after a call to Add on a container C and an element E, Contains returns True on E and on all elements which were in C before the call, plus potential additional elements equivalent to E,
- after a call to Add, the result of the Length function if any is incremented by one.

The contracts of `Empty` and `Insert` below ensure conformance to these consistency checks:

```
type T is private with
  Aggregate => (Empty       => Empty,
                Add_Unnamed => Insert),
  Annotate => (GNATprove, Container_Aggregates, "Predefined_Sets");

function Empty return T with
  Post => Length (Empty'Result) = 0
  and then (for all E in Element_Type => not Contains (Empty'Result, E));
procedure Insert (X : in out T; E : Element_Type) with
  Always_Terminates,
  Pre  => not Contains (X, E),
  Post => Length (X) = Length (X'Old) + 1 and Contains (X, E)
  and (for all F in Element_Type =>
         (if Contains (X'Old, F) then Contains (X, F)))
  and (for all F in Element_Type =>
         (if Contains (X, F) then Contains (X'Old, F) or Eq_Elem (F, E)));

function Eq_Elem (X, Y : Element_Type) return Boolean with
  Annotate => (GNATprove, Container_Aggregates, "Equivalent_Elements");

function Contains (X : T; E : Element_Type) return Boolean with
  Annotate => (GNATprove, Container_Aggregates, "Contains");

function Length (X : T) return Natural with
  Annotate => (GNATprove, Container_Aggregates, "Length");
```

Aggregates of types annotated with `Predefined_Sets` cannot contain *duplicates*, that is, two elements on which `Equivalent_Elements` returns True. This restriction on aggregate usage is enforced by GNATprove. It allows the tool to properly assess the cardinality of the resulting set. If a `Length` function is supplied and returns a signed integer type, and no `Capacity` function applies to the type, GNATprove also makes sure that the number of elements in the aggregate does not exceed the upper bound of the return type of `Length`. This last check is replaced by a check on the capacity if there is one.

When an aggregate `C` is encountered, GNATprove automatically infers that:

- `Contains (C, E)` returns True for every element `E` of the aggregate,
- `Contains (C, E)` returns False for every element `E` which is not equivalent to any element in the aggregate, and
- `Length (C)`, if supplied, is the number of elements in the aggregate.

The `Predefined_Maps` pattern can be applied to containers with named aggregates. It supports five kinds of function annotations:

- The `Default_Item` function returns the element associated by default to keys of total maps.
- The `Has_Key` function returns True if and only if its key parameter has an association in a partial map.
- The `Get` function returns the element associated to a key in the container.
- `Equivalent_Keys` is an equivalence relation such that `Has_Key` and `Get` return the same value on two equivalent keys.
- The `Length` function returns the number of associations in a partial map.

`Get` and `Equivalent_Keys` are mandatory, exactly one of `Default_Item` and `Has_Key` should be supplied (depending on whether the map is total - it associates a value to all keys - or partial) , and `Length` is optional and can only

be supplied for partial maps. If it is supplied, the `Length` function should return a signed integer type or a subtype of `Big_Integer`.

The consistency checks ensure that:

- if `Default_Item` is supplied, `Get` returns `Default_Item` for all keys on the result of `Empty`,
- if `Has_Key` is supplied, it returns False for all keys on the result of `Empty`, and `Length`, if specified, returns 0,
- `Add` can be called on a container `C` and a key `K` such that `Has_Key (C, K)` returns False (for patial maps) or `Get (C, K) = Default_Item` (for total maps),
- after a call to `Add` on a container `C`, a key `K`, and an element `E`, `Has_Key`, if any, returns True on `K` and on all keys which were in `C` before the call, plus potential additional keys equivalent to `K`,
- after a call to `Add` on a container `C`, a key `K`, and an element `E`, `Get` returns a copy of `E` on `K` and its association in `C` before the call on keys which are not equivalent to `K`, and
- after a call to `Add`, the result of the `Length` function if any is incremented by one.

The contracts of `Empty` and `Insert` below ensure conformance to these consistency checks for total maps:

```
type T is private with
  Aggregate => (Empty     => Empty,
                Add_Named => Insert),
  Annotate => (GNATprove, Container_Aggregates, "Predefined_Maps");

function Empty return T with
  Post => (for all K in Key_Type => Get (Empty'Result, K) = Default_Value);
procedure Insert (X : in out T; K : Key_Type; E : Element_Type) with
  Always_Terminates,
  Pre  => Get (X, K) = Default_Value,
  Post => Get (X, K) = E
  and (for all F in Key_Type =>
         (if not Eq_Key (F, K) then Get (X, F) = Get (X'Old, F)));

function Eq_Key (X, Y : Key_Type) return Boolean with
  Annotate => (GNATprove, Container_Aggregates, "Equivalent_Keys");

function Default_Value return Element_Type with
  Annotate => (GNATprove, Container_Aggregates, "Default_Item");

function Get (X : T; K : Key_Type) return Element_Type with
  Annotate => (GNATprove, Container_Aggregates, "Get";
```

And here is a version for partial maps:

```
type T is private with
  Aggregate => (Empty     => Empty,
                Add_Named => Insert),
  Annotate => (GNATprove, Container_Aggregates, "Predefined_Maps");

function Empty return T with
  Post => Length (Empty'Result) = 0
  and (for all K in Key_Type => not Has_Key (Empty'Result, K));
procedure Insert (X : in out T; K : Key_Type; E : Element_Type) with
  Always_Terminates,
  Pre  => not Has_Key (X, K),
```

(continues on next page)

(continued from previous page)

```
  Post => Length (X) - 1 = Length (X)'Old
  and Has_Key (X, K)
  and (for all F in Key_Type =>
         (if Has_Key (X'Old, F) then Has_Key (X, F)))
  and (for all L in Key_Type =>
         (if Has_Key (X, L) then Has_Key (X'Old, L) or Eq_Key (L, K)))
  and Get (X, K) = E
  and (for all F in Key_Type =>
         (if Has_Key (X'Old, F) then Get (X, F) = Get (X'Old, F)));

function Eq_Key (X, Y : Key_Type) return Boolean with
  Annotate => (GNATprove, Container_Aggregates, "Equivalent_Keys");

function Has_Key (X : T; K : Key_Type) return Boolean with
  Annotate => (GNATprove, Container_Aggregates, "Has_Key");

function Get (X : T; K : Key_Type) return Element_Type with
  Pre => Has_Key (X, K),
  Annotate => (GNATprove, Container_Aggregates, "Get");

function Length (X : T) return Natural with
  Annotate => (GNATprove, Container_Aggregates, "Length");
```

Aggregates of types annotated with `Predefined_Maps` cannot contain *duplicates*, that is, two keys on which `Equivalent_Keys` returns True. This restriction on aggregate usage is enforced by GNATprove. It allows the tool to determine the associated element uniquely and to assess the cardinality of the resulting map. If a `Length` function is supplied and returns a signed integer type, and no `Capacity` function applies to the type, GNATprove also makes sure that the number of elements in the aggregate does not exceed the upper bound of the return type of `Length`. This last check is replaced by a check on the capacity if there is one.

When an aggregate `C` is encountered, GNATprove automatically infers that:

- if `Has_Key` is supplied, `Has_Key (C, K)` returns True for every association `K => _` in the aggregate,
- `Get (C, K)` returns a copy of `E` for every association `K => E` in the aggregate,
- for every key `K` which is not equivalent to any key in the aggregate either `Has_Key (C, K)` returns False (for patial maps) or `Get (C, K)` returns `Default_Item` (for total maps), and
- `Length (C)`, if supplied, is the number of elements in the aggregate.

APPENDIX

F

GNATPROVE LIMITATIONS

F.1 Tool Limitations Leading to an Error Message

The following unsupported constructs are detected by GNATprove and reported through an error message:

- an abstract state marked as Part_Of a concurrent object,
- a reference to the "Access" attribute of an ownership type which does not occur directly inside an assignment statement, an object declaration, or a simple return statement,
- a conversion between access types with different designated types,
- a formal parameter of an access-to-subprogram type which is annotated with a type invariant,
- an access-to-subprogram type designating a protected subprogram,
- an access-to-subprogram type whose return type is annotated with a type invariant,
- an access-to-subprogram type designating a borrowing traversal function,
- an access-to-subprogram type designating a dispatching operation,
- an access-to-subprogram type designating a No_Return procedure,
- an access-to-subprogram type designating a subprogram annotated with Relaxed_Initialization,
- access attribute on a procedure which might raise exceptions,
- a reference to the "Address" attribute occuring within a subtype indication, a range constraint, or a quantified expression,
- a conversion between array types if some matching index types are modular types of different sizes,
- a conversion between array types if some matching index types are not both signed or both modular,
- a pragma Assert_And_Cut occurring immediately within a sequence of statements containing a Loop_Invariant,
- a borrowing traversal function whose first formal parameter does not have an anonymous access-to-variable type,
- a borrowing traversal function marked as a volatile function,
- a reference to the "Class" attribute on a constrained type,
- a representation attribute on an object of classwide type,
- a subtype predicate on a classwide type,
- a raise expression occurring in a precondition, unless it is only used to change the reported error and can safely be interpreted as False,
- a type constraint on a classwide subtype declaration,

- a type contract (subtype predicate, default initial condition, or type invariant) on a private type whose full view is another private type,
- a conversion between a fixed-point type and a floating-point type,
- a conversion between a fixed-point type and an integer type when the small of the fixed-point type is neither an integer nor the reciprocal of an integer,
- a conversion between a floating point type and a modular type of size 128,
- a conversion between fixed point types whose smalls are not "compatible" according to Ada RM G.2.3(21-24): the division of smalls is not an integer or the reciprocal of an integer,
- an object with subcomponents of an access-to-variable type annotated with an address clause whose value is the address of another object,
- delta aggregate with possible aliasing of component associations of an ownership type,
- entry families,
- aspect "Exceptional_Cases" on dispatching operations,
- procedures with exceptional contracts and parameters of mode "in out" or "out" subjected to ownership which might not be passed by reference,
- an extension aggregate whose ancestor part is a subtype mark,
- a goto statement occuring in a loop before the invariant which refers to a label occuring inside the loop but after the invariant,
- a reference to the "Image" or "Img" attribute on a type or an object of a type which is not a scalar type,
- interpolated string literals,
- container aggregates,
- an iterated component associations with an iterator specification ("for … of") in an array aggregate,
- the use of an incomplete view of a type coming from a limited with,
- a loop invariant in a list of statements with an exception handler,
- a loop with an iterator filter in its parameter specification,
- an array type with more than 4 dimensions,
- a modular type with a modulus greater than 2 ** 128,
- a move operation occuring as part of a conversion to an access-to-constant type,
- a function annotated as No_Return,
- a reference to a non-static attribute,
- a primitive operation which is inherited from several interfaces in a tagged derivation,
- a primitive operation which is inherited both from the parent type and from an interface in a tagged derivation,
- an iterator specification on a multidimensional array,
- a delta aggregate on a multidimensional array,
- a null aggregate which is a subaggregate of a multidimensional array aggregate with multiple associations,
- a non-scalar object declared in a loop before the loop invariant,
- a multiplication or division between a fixed-point and a floating-point value,

- a multiplication or division between different fixed-point types if the result is not in the "perfect result set" according to Ada RM G.2.3(21),
- a reference to the "Address" attribute in an address clause whose prefix has subcomponents of an access-to-variable type,
- a package declaration occurring in a loop before the loop invariant,
- a private type whose full view is not in SPARK annotated with two subtype predicates, one on the full view and one on the private view,
- a call to a primitive operation of a tagged type T occurring in the default initial condition of T with the type instance as a parameter,
- a call to a protected operation of a protected component inside a protected object,
- a call to a protected operation of the formal parameter of a subprogram,
- a protected entry annotated with a Refined_Post,
- an access-to-subprogram type used as a subcomponent of a type or an object annotated with Relaxed_Initialization,
- an object annotated with Relaxed_Initialization is part of an overlay,
- a concurrent type used as a subcomponent of a type or an object annotated with Relaxed_Initialization,
- a type annotated with an invariant used as a subcomponent of a type or an object annotated with Relaxed_Initialization,
- a variable annotated both with Relaxed_Initialization and as Part_Of a concurrent object,
- a protected component annotated with Relaxed_Initialization,
- a tagged type used as a subcomponent of a type or an object annotated with Relaxed_Initialization,
- a subtype with a discriminant constraint containing only subcomponents whose type is annotated with Relaxed_Initialization,
- a subprogram declaration occurring in a loop before the loop invariant,
- a call to a suspend operation on a suspension formal parameter,
- an occurrence of the target name @ in an assignment to an object of an anonymous access-to-variable type,
- an occurrence of the target name @ in an assignment to an object containing subcomponents of a named access-to-variable type,
- an access type designating an incomplete or private type with a subcomponent annotated with a type invariant,
- a private type declared in a nested package annotated with a type invariant,
- a private type declared in a private child package annotated with a type invariant,
- a protected type annotated with a type invariant,
- a tagged type with a subcomponent annotated with a type invariant,
- a tagged type annotated with a type invariant,
- a volatile object with asynchronous writers or readers and a type invariant,
- an uninitialized allocator inside an expression function,
- a reference to the "Alignment" attribute on a prefix which is not a type with an alignment clause,
- a component of an unconstrained unchecked union type in a tagged extension.

F.2 Other Tool Limitations

The generation of Global contracts by GNATprove has limitations. When this capability is used in a certification context, the generated contracts should be verified by the user. In particular:

- The Global contracts generated automatically by GNATprove for subprograms without an explicit one, whose body is not in SPARK, do not take into account indirect calls (through access-to-subprogram and dynamic binding) and indirect reads/writes to global variables (through access variables). See *Coarse Generation for non-SPARK Subprograms*.

Some features defined in the reference manual have not been implemented in the tool. It is the case of:

- classwide Global and Depends contracts as defined in SPARK RM 6.1.6, and
- the use of Ada.Synchronous_Barriers.Synchronous_Barrier type.

F.3 Flow Analysis Limitations

These limitations in the flow analysis part of GNATprove may result in a less precise (but sound) analysis.

- Flow dependencies caused by record assignments is not captured with perfect accuracy. This means that the value of one field might incorrectly be considered to participate in the derivation of another field that it does not really participate in.
- Initialization of multi-dimensional arrays with nested FOR loops can be only detected if the array bounds are given by static expressions. A possible solution is to use *Aspect Relaxed_Initialization and Ghost Attribute Initialized* instead in such a case and to prove that only initialized data is read.

F.4 Proof Limitations

These limitations in the proof part of GNATprove may result in a less precise (but sound) analysis.

- The following attributes are not yet supported in proof: Adjacent, Aft, Bit_Order, Body_Version, Copy_Sign, Definite, Denorm, First_Valid, Fore, Last_Valid, Machine, all Machine_* attributes, Model, all Model_* attributes, Partition_Id, Remainder, Round, Safe_First, Safe_Last, Scale, Scaling, Small, Unbiased_Rounding, Version, Wide_Image, Wide_Value, Wide_Width, Wide_Wide_Image, Wide_Wide_Value, Wide_Wide_Width, Width.

  The attributes First_Bit, Last_Bit and Position are supported but if there is no record representation clause then we assume that their value is nonnegative.
- Constants declared in loops before the loop invariant are handled as variables by the tool. This means in particular that any information about their values needed after the loop invariant must be stated explicitly in the loop invariant.

APPENDIX

G

PORTABILITY ISSUES

G.1 Compiling with a non-SPARK Aware Compiler

To execute a SPARK program, it is expected that users will compile the program (as an Ada program) using an Ada compiler. The SPARK language definition defines a number of implementation-defined (with respect to the Ada language definition) aspects, attributes, pragmas, and conventions. Ideally a SPARK program will be compiled using an Ada compiler that supports all of these constructs. Portability problems may arise if this is not the case.

This section is a discussion of the strategies available for coping with this situation.

Probably the most important rule is that pragmas should be used instead of aspect_specification syntax wherever this option is available. For example, use pragma Abstract_State rather than specifying the Abstract_State aspect of a package using aspect_specification syntax. Ada specifies that unrecognized pragmas shall be ignored, as opposed to being rejected. This is not the case for (syntactic) aspect specifications (this terminology is a bit confusing because a pragma can be used to specify an aspect; such a pragma is semantically, but not syntactically, an aspect specification). Furthermore, aspect specification syntax was introduced in Ada 2012 and will be rejected if the program is compiled as, for example, an Ada 95 program.

Many SPARK-defined constructs have no dynamic semantics (e.g., the Global, Depends, and Abstract_State aspects), so the run-time behavior of a program is unaffected if they are ignored by a compiler. Thus, there is no problem if these constructs are expressed as pragmas which are then ignored by the Ada compiler.

Of those constructs which do have dynamic semantics, most are run-time assertions. These include Loop_Variant, Loop_Invariant, Assert_And_Cut, Contract_Cases, Initial_Condition, and Refined_Postcondition. Because SPARK requires that the success of these assertions must be statically proven (and that the evaluation of the asserted condition can have no side effects), the run-time behavior of a program is unaffected if they are ignored by a compiler.

The situation with pragma Assume is slightly different because the success of the given condition is not statically proven. If ignoring an Assume pragma at run time is deemed to be unacceptable, then it can be replaced with an Assert pragma (at the cost of introducing a source code difference between the SPARK program that is analyzed statically and the Ada program that is executed). An ignored Assume pragma is the only case where the use of a SPARK-specific construct can lead to a portability problem which is not detected at compile time. In all other cases, either the Ada compiler will reject (as opposed to ignore) an unrecognized construct or the construct can safely be ignored.

An Ada compiler which does not support convention Ghost will reject any use of this convention. Two safe transformations are available for dealing with this situation - either replace uses of convention Ghost with convention Ada or delete the entities declared with a convention of Ghost. Just as was mentioned above in the case of modifying an Assume pragma, either choice introduces an analyzed/executed source code difference.

There are two SPARK attributes which cannot be used if they are not supported by the Ada compiler in question: the Update and Loop_Entry attributes.

SPARK includes a rule that a package which declares a state abstraction requires a body. In the case of a library unit package (or generic package) which requires a body only because of this rule, an Ada compiler that knows nothing about state abstractions would reject the body of the package because of the rule (introduced in Ada 95) that a library

unit package (or generic package) body is never optional; if it is not required then it is forbidden. In the unlikely event that this scenario arises in practice, the solution is to force the library unit package to require a body for some other reason, typically by adding an Elaborate_Body pragma.

If a SPARK program is to be compiled and executed as an Ada 95 program (or any other pre-2012 version of Ada), then of course any construct introduced in a later version of Ada must be avoided (unless it is expressed as a safely-ignored pragma). This seems worth mentioning because Ada 2012 constructs such as quantified expressions and conditional expressions are often heavily used in SPARK programs.

G.2 Implementation-specific Decisions

To make analysis as precise as possible and avoid producing too many false alarms, GNATprove makes some assumptions about the behavior of constructs which are listed in the reference manual of Ada as implementation specific. Note that GNATprove always adopts the same choices as the GNAT compiler, so these assumptions should be adequate when compiling with GNAT. However, when another compiler is used, it may be better to avoid these implementation specific constructs (see *Benefits of Using SPARK for Portability* for more details on how this can be achieved).

G.2.1 Parenthesized Arithmetic Operations

In Ada, non-parenthesized arithmetic operations could be re-ordered by the compiler, which may result in a failing computation (due to overflow checking) becoming a successful one, and vice-versa. By default, GNATprove evaluates all expressions left-to-right, like GNAT. When the switch `--pedantic` is used, a warning is emitted for every operation that could be re-ordered:

- any operand of a binary adding operation (+,-) that is itself a binary adding operation;
- any operand of a binary multiplying operation (*,/,mod,rem) that is itself a binary multiplying operation.

G.2.2 Base Type of User-Defined Integer Types

GNATprove follows GNAT in choosing as base type the smallest multiple-words-size integer type that contains the type bounds. For example, a user-defined type ranging from 1 to 100 will be given a base type ranging from -128 to 127 by both GNAT and GNATprove. The choice of base types influences in which cases intermediate overflows may be raised during computation. The choice made in GNATprove is the strictest one among existing compilers, as far as we know, which ensures that GNATprove's analysis detects a superset of the overflows that may occur at run time.

G.2.3 Size of 'Image and 'Img attributes

To avoid spurious range checks on string operations involving occurrences of the `'Img`, `'Image`, `'Wide_Image`, and `'Wide_Wide_Image` attributes, GNATprove makes an assumption about the maximal length of the returned string. If the attribute applies to an integer type, the bounds are the maximal size of the result of the attribute as specified in the language depending of the type's base type. Otherwise, GNATprove assumes that the length of such a string cannot exceed 255 (the maximal number of characters in a line) times 8 (the maximal size of a Wide_Wide_Character).

APPENDIX

H

SEMANTICS OF FLOATING POINT OPERATIONS

SPARK assumes that floating point operations are carried out in single precision (binary32), double precision (binary64) or extended precision (binary80) as defined in the IEEE-754 standard for floating point arithmetic. You should make sure that this is the case on your platform. For example, on x86 platforms, by default some intermediate computations may be carried out in extended precision, even if the type of the operands is of single or double precision, leading to unexpected results. With GNAT/GCC, you can specify the use of SSE arithmetic by using the compilation switches `-msse2 -mfpmath=sse` which cause all arithmetic to be done using the SSE instruction set which only provides 32-bit and 64-bit IEEE types, and does not provide extended precision. SSE arithmetic is also more efficient. Note that the ABI allows free mixing of units using the three types of floating-point, so it is not necessary to force all units in a program to use SSE arithmetic.

Several architectures also have fused-multiple-add (FMA) instructions, either in the base set (PowerPC, Aarch64) or in extensions (SPARC, x86). You should make sure that your compiler is instructed not to use such instructions, whose effect on the program semantics cannot be taken into account by SPARK source code analysis. On x86, such instructions are enabled in GNAT/GCC by means of an explicit switch like `-fma/-fma4` or by architecture switches like `-msse4.1` or `-mavx512f`, so make sure you're not using these. With GNAT/GCC, it is recommended to use switch `-ffp-contract=off` to disable all floating-point expression contractions, including FMA.

SPARK considers the floating point values which represent positive, negative infinity or NaN as invalid. Proof obligations are generated that such values cannot occur.

SPARK considers rounding on floating point arithmetic operations to follow Round-Nearest-Even (RNE) mode, where a real result is rounded to the nearest floating point value, and ties are resolved to the floating-point with a zero in the last place. This mode of rounding (the default in GNAT) should be forced if needed on the hardware to be able to rely on the results of GNATprove regarding floating point arithmetic (e.g. when using SSE arithmetic, the MXCSR register should specify RNE mode, which it does by default).

APPENDIX

I

SPARK ARCHITECTURE, QUALITY ASSURANCE AND MATURITY

I.1 Development Process and Quality Assurance

The SPARK development process and quality assurance are following the Adacore Quality Procedures in place for all development at AdaCore. This includes:

- The use of a report tracking system;
- Mechanisms for detecting and fixing defects;
- The usage of repositories and configuration management, the use of continuous integration technology, the stringent requirements on check-ins of source changes;
- The process for implementing new functionality;
- The process for maintaining user documentation;
- Ensuring quality of sources and technical documentation;
- Preparation of releases.

As an extension to Chapter 2, section "AdaCore internal testsuite", SPARK contains its own testsuites:

- The SPARK main testsuite: This testsuite contains 3600 tests. These tests are specifically targeted at the SPARK software and cover typical use cases, often represented by code sent to us by customers, as well as specific features of the SPARK software.
- The ACATS testsuite in SPARK mode: A selection of the ACATS testsuite mentioned in the AdaCore Quality Procedures for the compiler is also used to test the SPARK tools.

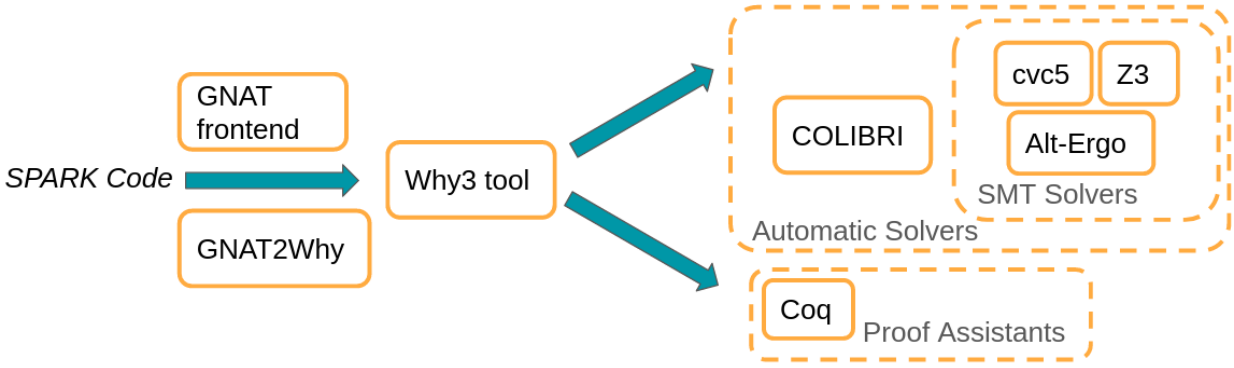
These tests are run on various occasions (see also Chapter 2 of the Adacore Quality Procedures):

- During nightly testing, once with assertions enabled, once without (the actual SPARK product);
- After every check-in, during continuous integration;
- To test a patch before check-in using the Mailserver or continuous builder technologies.

The known-problems file associated with a specific release of SPARK specially identifies soundness bugs, whose potential effect is to hide actual problems in the program, under the tags "missing-check" (when a check message could be missing) and "missing-violation" (for any other soundness issue). Such entries always include a "detection" field for detecting the possibility of triggering such a bug in the application of SPARK.

I.2 Structure of the SPARK Software

At a high level, the tool GNATprove reads source files in the Ada programming language, with some annotations specific to SPARK, processes them, and in the end issues a report about errors found and proved or unproved properties. Looking more closely at how this is achieved, one can see this high-level structure of SPARK:



The development of the GNAT front-end and GNAT2Why components entirely follows the procedures outlined in AdaCore Quality Procedures and the previous section. The other components, however, are mostly developed by third parties. Their development process and the relationship to AdaCore and Capgemini Engineering will be outlined below.

For the nightly testing of SPARK, the GNAT and GNAT2Why components are updated every night according to the changes made during the day by AdaCore and Capgemini Engineering developers. The other tools, however, contain also check-ins by other persons. We update these tools in a controlled way, and after careful testing of the consequences. In other words, a check-in made e.g. to Z3 at some specific date, will not be part of the SPARK package of the same day, instead it will be integrated into SPARK after some time and after thorough testing in the SPARK environment.

I.2.1 GNAT Front-end

SPARK shares its front-end (parsing and semantic analysis) with the GNAT compiler technology, which is very mature and has been used in countless projects for the last 20 years. The GNAT front-end is developed by AdaCore and follows the AdaCore quality procedures.

I.2.2 GNAT2Why

This part of GNATprove serves two purposes:

- Implement Flow Analysis, the part of the SPARK analysis which detects uninitialized variables, and computes and checks the use of global variables and parameters.
- Translate the SPARK source code to the Why language, for further processing by the Why3 tools. GNAT2Why is developed by AdaCore and Capgemini Engineering and follows the AdaCore quality procedures.

Because of its central role in GNATprove, and because it performs the critical encoding of the verification rules of SPARK when applied to a given program, this part of the tool receives special attention regarding soundness:

- Base Why3 theories used to encode Ada datatypes are realized in Coq, to rule out inconsistencies in the axiomatization (e.g. for bitvector or floating-point base theories).
- Unless the switch `--function-sandboxing=off` is used, the axioms generated by GNATprove to encode the semantics of functions in Why3 are "sandboxed", so that they are only applicable to contexts where the corresponding function is called, on the same arguments as those used in the call. This reduces the risk that a wrong contract could lead to an inconsistent axiom applicable everywhere.

- No axiom is generated for functions which might not return. This also reduces the risk that a wrong contract could lead to an inconsistent axiom.

A special run of the SPARK main testsuite allows to monitor statement coverage of the testsuite over GNAT2Why source code, which is maintained above 95%, and aiming at full statement coverage.

Addionally, through their use of the switch `--proof-warnings` aiming at detecting dead code at branching points in the program and other inconsistencies, users can protect against unsound tool behavior which would result in similar findings. This use is traditionally called a "smoke detector" in the scientific literature.

I.2.3 Why3

This part of GNATprove takes the information in the Why language produced by GNAT2Why, translates it further into a format suitable for SMT solvers such as Z3 and cvc5, and runs these tools. The results are reported back to gnat2why.

History: Started around the year 2000 by Jean-Christophe Filliâtre as "Why" (see Jean-Christophe Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003), it has undergone a number of redevelopments until its current version Why3 (since 2010).

Track record: Apart from SPARK, it is used by Frama-C, Atelier B, and other program verification tools.

Relationship with AdaCore/Capgemini Engineering: The Inria team around Why3 has strong ties with AdaCore and Capgemini Engineering. A number of research projects have been and are being carried out in collaboration with this team. This includes the Hi-Lite project, which led to the current version of SPARK based on Why3, a follow-up project SOPRANO, then a joint laboratory ProofInUse which evolved in the ProofInUse consortium. In addition, while Why3 is mainly developed at Inria, AdaCore and Capgemini Engineering have made important contributions to the technology, such as the so-called fast-WP, a more efficient implementation of the main algorithm of Why3, and the why3server, a more scalable method of running external tools such as SMT solvers.

- Main developers: Inria research institute
- Main website: http://why3.lri.fr
- Version Management: Git
- License: Open Source, LGPL 2.1
- Public mailing-list: why3-club@lists.gforge.inria.fr
- Bug tracking: https://gforge.inria.fr/tracker/?group_id=2990

I.2.4 Alt-Ergo

History: Started around the year 2005 at Inria by Sylvain Conchon and Evelyne Contejean as "Ergo" (see CC(X): Efficiently combining equality and solvable theories without canonizers. Sylvain Conchon, Évelyne Contejean, and Johannes Kanig. SMT Workshop, 2007). Starting from 2013, developed and distributed mainly by OCamlPro. Since then, OCamlPro issues every year a private release and a public release (lagging one year behind the private release). SPARK uses the public release of Alt-Ergo.

Track record: Apart from SPARK, it is used by Frama-C and Atelier B. In particular, used by Airbus for the qualification DO-178C of an aircraft [10].

Relationship with AdaCore/Capgemini Engineering: AdaCore and OCamlPro have collaborated in the SOPRANO project. AdaCore has contributed some minor changes to Alt-Ergo, including a deterministic resource limiting switch.

- Main developers: OCamlPro
- Main website: https://alt-ergo.ocamlpro.com/
- Version Management: Git

- License: CeCill-C (GPL compatible)
- Public mailing-list: alt-ergo-users@lists.gforge.inria.fr
- Bug tracking: https://github.com/OCamlPro/alt-ergo/issues

I.2.5 Z3

History: Started around the year 2007 at Microsoft Research by Leonardo de Moura and Nikolaj Bjørner (see Leonardo de Moura and Nikolaj Bjørner. Efficient E-Matching for SMT solvers. In Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings, volume 4603 of Lecture Notes in Computer Science, pages 183-198. Springer, 2007). Released to open source under a very permissive license in 2015.

Track record: Apart from SPARK, used by Dafny and PEX projects inside Microsoft. Has won the SMT competition several times in several categories.

Relationship with AdaCore/Capgemini Engineering: AdaCore and Capgemini Engineering have provided bug reports, feature requests and small fixes to the Z3 team, in particular related to a deterministic resource limiting switch.

- Main developers: Microsoft
- Main website: https://github.com/Z3Prover/z3
- Version Management: Git
- License: MIT License
- Stackoverflow community: http://stackoverflow.com/questions/tagged/z3
- Bug tracking: https://github.com/Z3Prover/z3/issues/

I.2.6 cvc5

History: cvc5 is the fifth version of the CVC prover family. It has evolved from the CVC4 prover, whose development started in 2012.

Track record: Very good results in various SMT competitions. Used in TNO tool.

Relationship with AdaCore/Capgemini Engineering: AdaCore and Capgemini Engineering have provided bug reports, feature requests and small fixes to the cvc5 team, in particular related to a deterministic resource limiting switch. AdaCore is a founding member of Centaur (the Center for Automated Reasoning at Stanford University) which is a main contributor to cvc5.

- Main developers: Stanford University and the University of Iowa
- Main website: https://cvc5.github.io/
- Version Management: Git
- License: Modified BSD License
- Discussions: https://github.com/cvc5/cvc5/discussions
- Bug tracking: https://github.com/cvc5/cvc5/issues

I.2.7 COLIBRI

History: COLIBRI is a library (COnstraint LIBrary for veRIfication) developed at CEA LIST and used for verification or test data generation purposes since 2000, using the techniques of constraint programming. The variety of types and constraints provided by COLIBRI makes it possible to use it in many testing and formal methods tools at CEA LIST.

Track record: Winner (2018) and Runner-up (2019) in the quantifier-free floating-point division of the SMT competition.

Relationship with AdaCore/Capgemini Engineering: AdaCore and CEA collaborate together to improve COLIBRI.

- Main developers: CEA
- Version Management: Git

APPENDIX

J

GNU FREE DOCUMENTATION LICENSE

Version 1.1, March 2000

59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

J.1 PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document 'free' in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of 'copyleft', which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

J.2 APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The 'Document', below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as 'you'.

A 'Modified Version' of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A 'Secondary Section' is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The 'Invariant Sections' are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The 'Cover Texts' are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A 'Transparent' copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not 'Transparent' is called 'Opaque'.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The 'Title Page' means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, 'Title Page' means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

J.3 VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

J.4 COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

J.5 MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled 'History', and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled 'History' in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the 'History' section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled 'Acknowledgements' or 'Dedications', preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled 'Endorsements'. Such a section may not be included in the Modified Version.
- Do not retitle any existing section as 'Endorsements' or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled 'Endorsements', provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

J.6 COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled 'History' in the various original documents, forming one section entitled 'History'; likewise combine any sections entitled 'Acknowledgements', and any sections entitled 'Dedications'. You must delete all sections entitled 'Endorsements.'

J.7 COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

J.8 AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an 'aggregate', and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

J.9 TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

J.10 TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

J.11 FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License 'or any later version' applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

J.12 ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (c)  YEAR  YOUR NAME.

Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being LIST THEIR TITLES, with the
Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
A copy of the license is included in the section entitled 'GNU
Free Documentation License'.
```

If you have no Invariant Sections, write 'with no Invariant Sections' instead of saying which ones are invariant. If you have no Front-Cover Texts, write 'no Front-Cover Texts' instead of 'Front-Cover Texts being LIST'; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

INDEX

Symbols

A

B

C

D

E

M

N

O

P

Q

R

S

T

U

V

W