Course Overview

About This Course

Styles

- *This* is a definition
- `this/is/a.path`
- `code is highlighted`
- `commands are emphasized --like-this`
- This → Is → An IDE Menu

A Little History

Motivating Example

- Consider these lines of code from the original release of the Tokeneer code (demonstrator for the NSA)

```
if Success and then
   (RawDuration * 10 <= Integer(DurationT'Last) and
    RawDuration * 10 >= Integer(DurationT'First))
then
    Value := DurationT(RawDuration * 10);
else
```

- Can you see the problem?
- This error escaped lots of testing and reviews!

The Verifying Compiler

- Could a compiler find the error we just saw?
  - Formal **verification** of source code
- What if we had a verifying compiler?
  - Check correctness at **compile time**
  - Perform **exhaustive** checking
  - Use types, assertions, and other information in the source code as correctness criteria
  - Work in combination with other program development and testing tools
- Grand Challenge for computer science [Hoare 2003]

Formal Verification and Programming Languages

- There is a catch...
- Our ability to deliver automatic formal verification **critically** depends on the **language** that is being analyzed.
- Most languages were **not** designed with formal verification as a primary design goal.

Formal Verification Goals

- Ideally we would like static verification to be:
  - Deep (tells you something **useful**)
  - Sound (with **no false negatives**)
  - Fast (tells you **now**)
  - Precise (with as few false alarms/positives as possible)
  - Modular (analyzes modules in parallel)
  - Constructive (works on incomplete programs)
- SPARK is designed with these goals in mind. Since the eighties!
  - But the language and tools have evolved considerably...

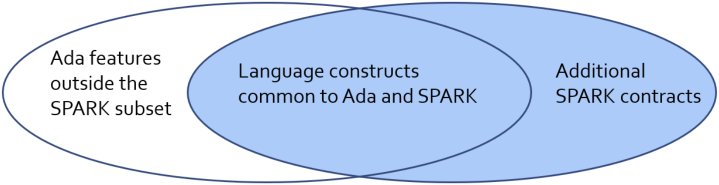
SPARK

What Is SPARK?

- SPARK is
  - A programming **language**
  - A set of formal verification **tools**
  - A **design approach** for high-integrity software
- All of the above!

What Is SPARK?

- Programming language - relationship with Ada:



Course Contents

Course Outline

- Introduction to SPARK
  - Formal Methods and SPARK
  - SPARK Language
  - SPARK Tools
- Formal verification in SPARK
  - Flow Analysis
  - Proof
- Specifications in SPARK
  - Specification Language
  - Subprogram Contracts
  - Type Contracts
- Advanced Formal Verification
  - Advanced Proof
  - Advanced Flow Analysis
- Advanced topics
  - Pointer Programs
  - Auto-Active Proof
  - State Abstraction
- SPARK Boundary

Course Goals

- What will you do after the course?
  - Be comfortable with the fundamentals of SPARK.
  - Know where to find out more.
  - Let SPARK work for you on your next project?
  - What else?

Formal Methods and SPARK

Introduction

High-Integrity Software

- Also known as (safety- or security- or mission-) *critical software*
- Has reliability as the most important requirement
  - More than cost, time-to-market, etc.
- Must be known to be **reliable** before being deployed
  - With **extremely** low failure rates
    - e.g., 1 in 10^9 hours (114,080 **years**)
  - Testing alone is insufficient and/or infeasible for such rates
- Is not necessarily safety-critical (no risk of human loss)
  - Satellites
  - Remote exploration vehicles
  - Financial systems

Developing High-Integrity Software

- Software quality obtained by a combination of
  - Process
    - Specifications
    - Reviews
    - Testing
    - Others: audits, independence, expertise...
  - Arguments
    - System architecture
    - Use cases
    - Programming language
    - Static code analysis
    - Dynamic code analysis
    - etc...
- Need to comply with a certification regime
  - Process-based or argument-based
  - Independently assessed (avionics, railway) or not (automotive)

Formal Methods

Formal Methods

- Mathematical techniques applied to the development or verification of software
  - *Heavyweight formal methods* expose the maths to users
  - *Lightweight formal methods* hide the maths from users
- Industrially usable formal methods
  - Are applicable to **existing** development **artifacts** (models, code, etc.)
  - Are automated and integrated in **existing processes**
  - Provide value for **certification**
  - Explicitly **encouraged** by some standards
    - Railway (EN 50128)
    - Avionics (DO-178C + DO-333 Formal Methods Supplement)
    - Security (Common Criteria)

Static Analysis of Programs

- *Abstract interpretation* (AbsInt)
  - AbsInt analyzes an **abstraction** of the program
- *Symbolic execution* (SymExe) and *bounded model checking* (BMC)
  - Both analyze possible traces of execution of the program
  - SymExe explores traces **one by one**
  - BMC explores traces **all at once**
- *Deductive verification* (Proof)
  - Proof analyzes functions **against their specification**
- Static analysis is a formal method when it is *sound*
  - Soundness means no missing alarms
- All techniques have different costs and benefits

Goals of Static Analysis of Programs

- **Automation** is better with AbsInt and SymExe/BMC
  - Proof incurs the cost of writing specification of functions
- **Precision** is better with SymExe/BMC and Proof
  - Automatic provers are **more powerful** than abstract domains
  - SymExe/BMC explore infinitely many traces
    - Limit the exploration to a subset of traces
- **Soundness** is better with AbsInt and Proof
  - Soundness is not missing alarms (aka *false negatives*)
  - AbsInt may cause false alarms (aka *false positives*)
  - Sound handling of loops and recursion in AbsInt and Proof

Capabilities of Static Analysis of Programs

- **Modularity** is the ability to analyze a partial program
  - Most programs are partial
    - Libraries themselves
    - Use of external libraries
    - Program during development
  - Proof is inherently modular
- **Speed** of the analysis drives usage
  - Unsound analysis can be much faster than sound one
  - For sound analysis, modular analysis is faster
- **Usage** depends on capabilities
  - Fast analysis with no false alarms is better for *bug-finding*
  - Modular analysis with no missing alarms is better for *formal verification*

Comparing Techniques on a Simple Code

- Consider a simple loop-based procedure

```
procedure Reset (T : in out Table; A, B : Index) is
begin
   for Idx in A .. B loop
      T(Idx) := 0;
   end loop;
end;
```

- `T(Idx)` is safe $\iff$ `Idx in T'Range`
- As a result of calling `Reset`:
  - Array `T` is initialized between indexes `A` and `B`
  - Array `T` has value zero between indexes `A` and `B`

Abstract Interpretation

- `Reset` is analyzed in the context of each of its calls
  - If the values of `Table`, `A`, `B` are precise enough, AbsInt can deduce that `Idx in T'Range`
  - Otherwise, an **alarm** is emitted (for sound analysis)
- Initialization and value of individual array cells is **not** tracked
  - The assignment to a cell is a *weak update*
    - The abstract value for the whole array now includes value zero
    - ... but is also possibly uninitialized or keeps a previous value
  - After the call to `Reset`, the analysis does **not** know that `T` is initialized with value zero between indexes `A` and `B`

Symbolic Execution and Bounded Model Checking

- `Reset` is analyzed in the context of **program traces**
  - If the values of `A` and `B` are *close enough*, SymExe/BMC can analyze all loop iterations and deduce that `Idx in T'Range`
  - Otherwise, an **alarm** is emitted (for sound analysis)
- Analysis of loops is limited to few iterations (same for recursion)
  - The other iterations are ignored or approximated, so the value of `T` is **lost**
  - After the call to `Reset`, the analysis does **not** know that `T` is initialized with value zero between indexes `A` and `B`

Deductive Verification

- `Reset` is analyzed in the context of a *precondition*
  - Predicate defined by the user which restricts the calling context
  - Proof checks if the precondition entails `Idx in T'Range`
  - Otherwise, an **alarm** is emitted
- Initialization and value of individual array cells is tracked
- Analysis of loops is based on user-provided *loop invariants*

```
T(A .. Idx)'Initialized and T(A .. Idx) = (A .. Idx => 0)
```

- Code after the call to `Reset` is analyzed in the context of a *postcondition*

```
T(A .. B)'Initialized and T(A .. B) = (A .. B => 0)
```

  - So the analysis now **knows** that `T` is initialized with value zero between indexes `A` and `B`

SPARK

SPARK Is a Formal Method

- **Soundness** is the most important requirement (no missing alarms)
- Analysis is a **combination of techniques**
  - *Flow analysis* is a simple form of modular abstract interpretation
  - *Proof* is modular deductive verification
- Inside proof, abstract interpretation is used to compute **bounds** on arithmetic expressions
  - Based on type bounds information
  - e.g if `X` is of type `Natural`
  - Then `Integer'Last - X` cannot overflow

SPARK Is a Language Subset

- Static analysis is **very tied** to the programming language
  - Strong typing **simplifies** analysis
  - Some language features **improve** analysis precision
    - e.g. first-class arrays with bounds `Table'First` and `Table'Last`
  - Some language features **degrade** analysis precision
    - e.g. arbitrary aliasing of pointers, dispatching calls in OOP

- SPARK hits the **sweet spot** for proof
  - Based on strongly typed feature-rich Ada programming language
  - **Restrictions** on Ada features to make proof easier
    1. Simplify user's effort for annotating the code
    2. Simplify the job of automatic provers

- "SPARK" originally stands for "SPADE Ada Ratiocinative Kernel"

History of SPARK

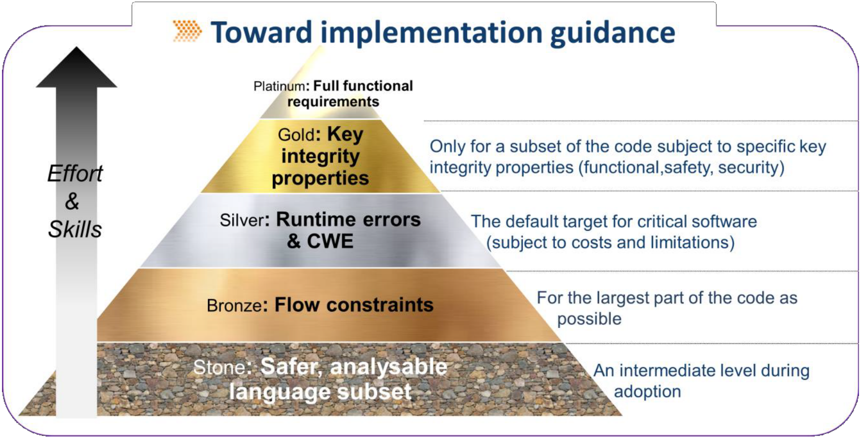
- *Vintage SPARK* followed Ada revisions
  - SPARK 83 based on Ada 83
  - SPARK 95 based on Ada 95
  - SPARK 2005 based on Ada 2005
- Since 2014, *SPARK* is updated annually
  - OO programming added in 2015
  - Concurrency added in 2016
  - Type invariants added in 2017
  - Pointers added in 2019
  - Exceptions added in 2023

Applying SPARK in Practice

Levels of Software Assurance

- Various reasons for using SPARK
- Levels of software assurance
  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
- Higher levels are more costly to achieve
- Higher levels build on lower levels
  - Project can decide to move to higher level later

Levels of Software Assurance in Pictures



Objectives of Using SPARK

- **Safe** coding standard for critical software
  - Typically achieved at **Stone or Bronze** levels
- Prove absence of run-time errors (*AoRTE*)
  - Achieved at **Silver** level
- Prove correct **integration** between components
  - Particular case is correct API usage
- Prove **functional correctness**
- Ensure correct behavior of parameterized software
- Safe **optimization** of run-time checks
- Address data and control coupling
- Ensure portability of programs

Project Scenarios

- Maintenance and evolution of existing Ada software
  - Requires migration to SPARK of a part of the codebase
  - Fine-grain control over parts in SPARK or in Ada
  - Migration guide available

https://www.adacore.com/books/implementation-guidance-spark

- Can progressively move to higher assurance levels

- New developments in SPARK
  - Either completely in SPARK
  - More often interfacing with other code in Ada/C/C++, etc.

Quiz

Quiz - Formal Methods

Which statement is correct?

A. A formal method analyses code.
B. A formal method has no missing alarms.
C. A formal method has no false alarms.
D. Static analysis of programs should be automatic, precise and sound.

Quiz - Formal Methods

Which statement is correct?

A. A formal method analyses code.
B. ***A formal method has no missing alarms.***
C. A formal method has no false alarms.
D. Static analysis of programs should be automatic, precise and sound.

Explanations

A. Formal methods can also apply to requirements, models, data, etc.
B. Correct
C. To achieve soundness, it may be impossible to avoid false alarms.
D. Not all three at the same time.

Quiz - SPARK

Which statement is correct?

A. SPARK is a recent programming language.
B. SPARK is based on proof.
C. SPARK analysis can be applied to any Ada program.
D. SPARK requires annotating the code with specifications.

Quiz - SPARK

Which statement is correct?

A. SPARK is a recent programming language.
B. SPARK is based on proof.
C. SPARK analysis can be applied to any Ada program.
D. ***SPARK requires annotating the code with specifications***.

Explanations

A. SPARK is a subset of Ada dating back to the 80s.
B. SPARK is also based on flow analysis which is a form of abstract interpretation.
C. SPARK subset restricts the features of Ada for proof.
D. Correct

Quiz - SPARK in Practice

Which statement is correct?

A. There are 5 levels of software assurance with SPARK.
B. Proving absence of run-time errors is hard with SPARK.
C. Full functional correctness is impossible to prove with SPARK.
D. SPARK code cannot be mixed with other programming languages.

Quiz - SPARK in Practice

Which statement is correct?

A. ***There are 5 levels of software assurance with SPARK.***
B. Proving absence of run-time errors is hard with SPARK.
C. Full functional correctness is impossible to prove with SPARK.
D. SPARK code cannot be mixed with other programming languages.

Explanations

A. Correct
B. AoRTE is a common objective with SPARK because it is simple.
C. Full functional correctness is hard but can be achieved.
D. SPARK code can be interfaced with code in Ada/C/C++, etc.

Summary

Formal Methods and SPARK

- Development of large, complex software is **difficult**
  - Especially so for high-integrity software
- Formal methods **can** be used industrially
  - During development and verification
  - To address objectives of certification
  - They must be sound (no missing alarm) in general
- SPARK is an **industrially** usable formal method
  - Based on flow analysis and proof
  - At various levels of software assurance

SPARK Language

Introduction

Design Goals for SPARK

- Support formal analysis that is
  - Deep - it tells you something **useful**
  - Sound - it has **no** missing alarms
  - Precise - it has **few** false alarms
  - Fast - it can run as part of development
  - Modular - it analyzes modules in **parallel**
  - Constructive - it works on **incomplete programs**
- Combine tool automation and user interaction
  - Automate as much as possible
  - Rely on the user to provide essential code annotations
- Combine execution and proof of specifications
- Support the largest possible subset of Ada 2022

Excluding Ambiguity

- Soundness requires that program semantics are **clear**
- Easiest way is to avoid **language** ambiguities:
  - No *erroneous behavior* from Ada Reference Manual
    - Cases where error can't be detected by the compiler or at run-time: e.g. dereference a pointer after it was deallocated
  - No *unspecified* features from Ada Reference Manual
    - Cases where the compiler makes a choice: e.g. order of evaluation of parameters in a call
  - Limit *implementation defined* features from Ada Reference Manual
    - Cases where the choice of the compiler is documented: e.g. size of standard integer types
    - Analyzer should make the same choices as the compiler
- Also facilitates **portability** across platforms and compilers!

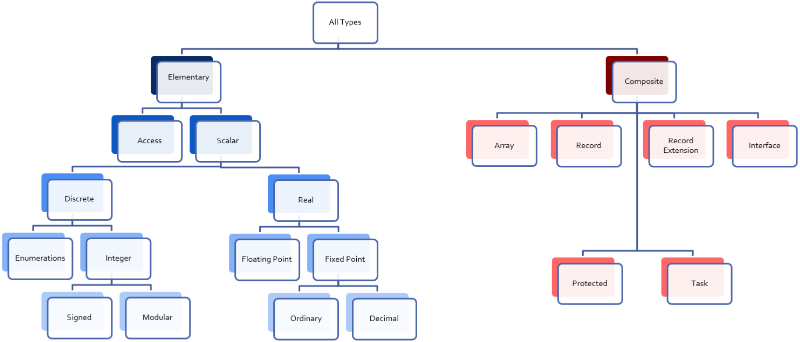
SPARK Reference Manual

- Precise definition of the SPARK subset
- Builds on the Ada Reference Manual
  - Follows the **same section numbering**
  - Has similar subsections:
    - **Syntax**
    - **Name Resolution Rules**
    - **Legality Rules**
    - **Static Semantics**
    - **Dynamic Semantics**
    - **Verification Rules** *(specific to SPARK RM)*
    - **Examples**

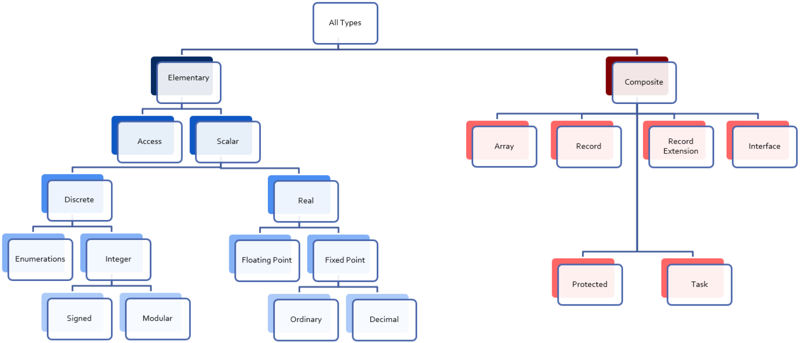
https://docs.adacore.com/live/wave/spark2014/html/spark2014_rm/packages.html

SPARK Language Subset

Categories of Types in Ada



Categories of Types in SPARK



SPARK supports all the types in Ada, with some restrictions

Assertions in SPARK

- Assertions in Ada are just `Boolean` expressions
  - They can be executed
  - Thus they can raise runtime errors (to be checked in SPARK)
- Low-level assertions

```
pragma Assert (Idx in T'Range and then T (Idx) = 0);
```

- High-level assertions, aka specifications, aka *contracts*

```
function Get (T : Table; Idx : Index) return Elem
  with Pre => Idx in T'Range and then T (Idx) = 0;
```

- Much more to come in later courses

Excluded Ada Features

- Backward `goto` statement
  - Can create loops, which require a specific treatment in formal verification
- Controlled types
  - Creates complex control flow with implicit calls
- Tasking features: `accept` statement (aka *rendezvous*), `requeue` statement, `select` statement, etc
  - But features in Ravenscar and Jorvik profiles are supported

Support for Generics

- Only **instances** of generics are analyzed
- Analysis of generics themselves would require:
    - Extending the SPARK language with new specifications
        - To name objects manipulated through calls to formal parameters
        - To add dependency contracts to formal subprogram parameters
    - More efforts from users to annotate programs
- **No restrictions** regarding use of generics

Support for OO Programming

- Root class and derived class (aka tagged types) must respect the *Liskov Substitution Principle* (LSP)
  - Behavior of overriding subprogram must be a subset of the allowed behaviors of the overridden subprogram
    - Overridden subprogram is in root class
    - Overriding subprogram is in derived class
- Overriding subprogram puts less constraints on caller than overridden one
  - *Precondition* must be weaker in overriding subprogram
- Overriding subprogram gives more guarantees to caller than overridden one
  - *Postcondition* must be stronger in overriding subprogram
- Overriding subprogram cannot access more global variables than overridden one

Support for Concurrency

- Ravenscar and Jorvik profiles of Ada are **supported**
- Tasks and protected objects must be defined at **library level**
- Tasks can only communicate through *synchronized objects*
  - Protected objects
  - Atomic objects
- This ensures absence of data races (aka race conditions)
  - One task writes an object while another task reads it
  - Two tasks write the object at the same time
- This is also a benefit for programs on a single core!
  - Concurrency $\neq$ parallelism

Language Restrictions

Main Language Restrictions

- Regular functions **without side-effects**
  - Thus expressions are also without side-effects
  - Aspect `Side_Effects` to signal function with side-effects
- Memory **ownership** policy (like in Rust)
- Absence of interferences
  - No problematic aliasing between variables
- Termination of subprograms
  - Functions must **always** terminate normally
- OO programming must respect Liskov Substitution Principle
- Concurrency must support Ravenscar or Jorvik profile

Functions Without Side-Effects

- *Side-effects* of a function are:
  - Writing to a global variable
  - Writing to an `out` or `in out` parameter
  - Reading a volatile variable
  - Raising an exception
  - Not terminating
- But *volatile functions* can read a volatile variable
  - Details discussed in the course on SPARK Boundary
- Only *functions with side-effects* can have side-effects
  - Signaled with aspect `Side_Effects`
  - Restricted to appear only as right-hand side of assignments

Side-Effects and Ambiguity

- If function `Fun` writes to global variable `Var`, what is the value of the expression `Fun = Var`?
  - `Var` may be evaluated before the call to `Fun`
  - ...or after the call to `Fun`
  - Thus leading to an ambiguity

```
Var : Integer := 0;
function Fun return Integer is
begin
   Var := Var + 1
   return Var;
end Fun;
pragma Assert (Fun = Var); -- Ambiguous evaluation
```

- Same with `Fun` writing to an `out` or `in out` parameter

Benefits of Functions Without Side-Effects

- Expressions have no side-effects
  - **Unambiguous** evaluation of expressions
  - Simplifies both flow analysis and proof
- Specifications and assertions have no side-effects
  - As specifications and assertions are expressions
- SPARK functions are **mathematical functions** from inputs to a result
  - Interpreted as such in proof

Absence of Interferences

- *Interferences* between names `A` and `B` when:
  - `A` and `B` designate the **same object** (*aliasing*)
  - and the code writes to `A`, then reads `B`
  - or the code writes to `A` and to `B`
- Interferences are caused by passing parameters
  - Parameter and global variable may designate the same object
  - Two parameters may designate the same object
- Thus no interferences on function calls!

Interferences and Ambiguity (1/2)

- If procedure `Proc` writes to parameter `A` then to parameter `B`, what is the value of **Var** after the call `Proc (Var, Var)`?
  - if `A` and `B` are passed by reference: the value of `B`
  - if `A` and `B` are passed by copy: the value of `A` or `B`, depending on which one is copied back last
  - Thus leading to an ambiguity

```
Var : Integer := 0;
procedure Proc (A, B : out Integer) is
begin
   A := 0;
   B := 1;
end Proc;
Proc (Var, Var); -- Ambiguous call
```

- Actually, Ada forbids this simple case and GNAT rejects it
  - But problem remains with `Table(Var)` instead of `Var`

Interferences and Ambiguity (2/2)

- If procedure `Proc` writes to parameter `A` then reads global variable `Var`, what is the value read in a call to `Proc (Var)`?
  - if `A` is passed by reference: the value written to `A`
  - if `A` is passed by copy: the initial value of `Var`
  - Thus leading to an ambiguity

```
type Int is record Value : Integer; end record;
Var : Int := (Value => 0);
procedure Proc (A : out Int) is
begin
   A := (Value => 1);
   pragma Assert (Var = A); -- Ambiguous
end Proc;
Proc (Var);
```

- Ada cannot forbid and GNAT cannot detect this case

Benefits of Absence of Interferences

- No hidden changes to an object `A` through another unrelated name
  - **Simplifies** both flow analysis and proof
- No need for users to add specifications about separation
  - Between parameters and global variables
  - Between parameters themselves
  - Between parts of objects (one could be a part of another)
- Program behavior does not depend on parameter-passing mechanism
  - This improves **portability** across platforms and compilers!

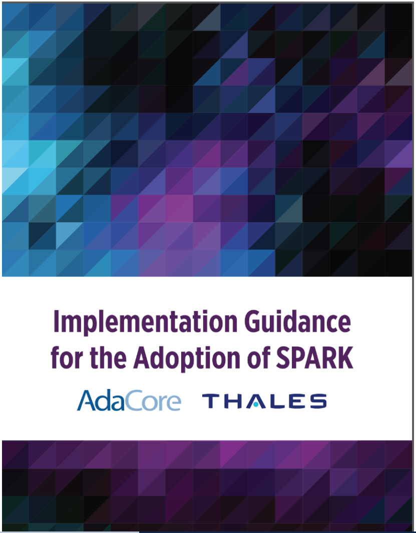
Migrating to SPARK

Migrating From Ada to SPARK

- Analyzing the Ada code will point to SPARK violations
- First goal is to reach **Stone level**: Valid SPARK
- Violation: functions with side-effects
  - Fix: add aspect `Side_Effects` to functions, move calls to assignments
- Violation: pointers do not respect ownership
  - Fix: change types and code to respect ownership
- Violation: illegal use of (volatile) variables inside expressions or functions
  - Fix: introduce temporaries, mark functions as volatile
- Define a SPARK interface for a unit in Ada
  - Details discussed in the course on SPARK Boundary

Adoption Guidance Document

- Based on adoption experience
- Proposes adoption levels
- For every level, presents:
  - Benefits, impact on process, costs, and limitations
  - Setup and tool **usage**
  - **Messages** issued by the tool
  - **Remediation** solutions



Migrating From C to SPARK

- Same recommendations as when migrating from C to Ada
- Even more important to use appropriate types
  - private types as much as possible (e.g. private type for flags with constants and boolean operator instead of modular type)
  - enumerations instead of `int`
  - ranges on scalar types
  - non-null access types
  - type predicates
- Special attention on the use of pointers
  - C uses pointers **everywhere**
  - Better to use parameter modes `out` and `in out` and array types in Ada
  - Choose between **different access types** in SPARK, with different semantics
    - Details discussed in the course on Pointer Programs

Summary

SPARK Language

- SPARK was designed **for formal analysis**
- **Soundness** is key!
  - No language ambiguities
  - Hence regular functions without side-effects
  - Hence absence of interferences
- Still, SPARK subset is most of Ada 2022
  - All categories of types
  - OO programming with LSP
  - Concurrency with Ravenscar and Jorvik
  - Pointer programs with ownership
- Recommendations for migration from Ada or C

SPARK Tools

Introduction

Identifying SPARK Code

- Pragma or aspect `SPARK_Mode` identifies SPARK code
- As a pragma in the global/local configuration pragmas file
- As a configuration pragma at the start of a unit
  - Note: it comes before **with** clauses

```
pragma SPARK_Mode (On); -- On is the default
with Lib; use Lib;
package Pack is ...
```

- As an aspect on the unit declaration

```
package Pack
  with SPARK_Mode
is ...
```

- **Both** unit spec and unit body need a pragma/aspect

Main Tools for SPARK

- GNAT development tools: SPARK is a subset of Ada 2022
    - Compiler also checks **SPARK-specific** legality rules
- SPARK analysis tools
    - Flow analysis and proof
    - File dependencies are **different** from the compiler
        - Due to generation of data dependencies
        - Analysis of unit depends on bodies of `with`'ed units
        - ...unless all data dependencies are specified
    - Behavior similar to builder like GPRBUILD
        - Units can be analyzed in parallel on multicore machines
        - Minimal rework if code and dependencies did not change
- IDEs for Ada/SPARK development

GNAT Development Tools

Compiling SPARK Code

- GNAT compiler for Ada/SPARK
  - Checks conformance of source with Ada and SPARK legality rules
  - Compiles source into executable
- Native and cross compilers
- Any runtime library: full, embedded, light-tasking, light

Enabling Assertions at Run-Time

- Assertions can be enabled globally with switch `-gnata`
- Assertions can be enabled/disabled locally with pragma `Assertion_Policy`

  For example to enable preconditions and disable postconditions:

  ```
  pragma Assertion_Policy (Pre => Check, Post => Ignore);
  ```
- Pragma can also be used in global/local configuration pragmas file
- Failing assertion raises exception `Assertion_Failure`

Debugging SPARK Code

- GDB debugger for Ada/SPARK
  - Code should be compiled with `-g -O0`
- Assertions can be debugged **too**!
  - Code should be compiled with `-gnata`

SPARK Analysis Tools

GNATPROVE - A Command Line Tool

- Invocation syntax: `gnatprove -P prj-file [switches]`
- If project file not given, like GPRBUILD:
  - Takes the project file in the **current directory** if present
  - Otherwise generates a basic project file
- See all switches with: `gnatprove --help`
  - Basic switches such as number of processors to use
    - Analysis modes with `--mode=`
    - Reporting mode with `--report=`
    - Warnings mode with `--warnings=`
    - Proof level with `--level=0/1/2/3/4`
  - Advanced switches for **fine-grained** control
    - Prover selection with `--prover=`
    - Prover control with `--timeout= --steps= --memlimit=`

GNATPROVE - Project File Usage

- Tool package `Prove` corresponds to GNATPROVE
  - Use attribute `Proof_Switches` to apply tool-defined switches
    - For all files with value `"Ada"`
    - For specific file with its name

```
project Proj is
  package Prove is
    for Proof_Switches ("Ada") use ("--level=2");
    for Proof_Switches ("file.adb") use ("--level=3");
  end Prove;
end Proj;
```

  - Use attribute `Proof_Dir` to specify directory for session files

Setting the Default SPARK_Mode Value

- Set SPARK_Mode in a global/local configuration pragmas file **config.adc**

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

- Set the Global_Configuration_Pragmas attribute in the project file

```
project Proj is
   package Builder is
      for Global_Configuration_Pragmas use "config.adc";
   end Builder;
end Proj;
```

Adapting the Project File for Analysis

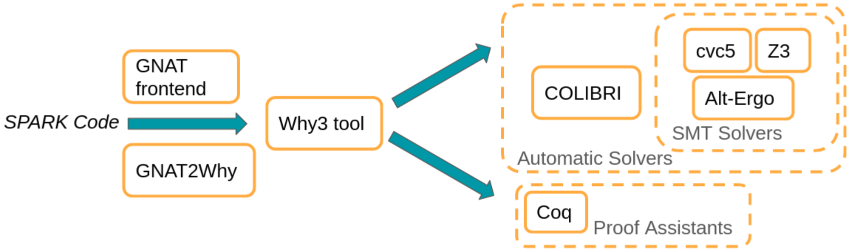
- If needed, define a **project variable** to control sources, compilation switches, etc.

```
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");
end case;
```

- Run GNATPROVE with appropriate value of `MODE` defined in the environment or on the command-line

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

Structure of GNATPROVE



Legality Checking

- **First step** in analysis
- GNATPROVE does only that with switch `--mode=check_all`
- Error messages on violations
  - Need to fix to go beyond this step
  - Ex: `<expr> cannot depend on variable input <var>`
  - May include fix:
    `use instead a constant initialized to the expression with`
    → apply the suggested fix
  - May include *explain code*: `[E0007]` → launch
    `gnatprove --explain=E0007` for more information
- Includes ownership checking, detailed in course on Pointer Programs

Flow Analysis

- *Flow analysis* is a prerequisite to proof
- GNATprove does that with switch `--mode=flow`
  - This follows legality checking
- Corresponds to Examine menus in IDEs
- GNATprove applies flow analysis to each subprogram separately
  - Notion of dependency contracts summarize effects of call
- Outputs messages:
  - Error messages need to be fixed
  - Check messages need to be reviewed, and either fixed or justified
  - Warnings can be inspected and silenced

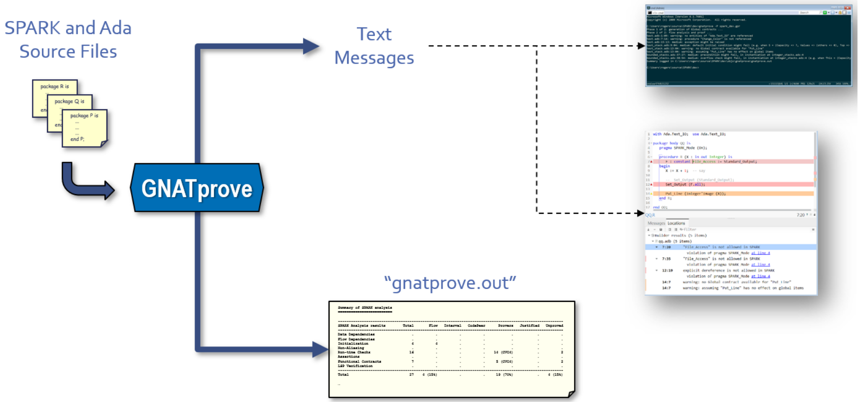
Proof

- *Proof* is the final step
- GNATPROVE does it all with switch `--mode=all` (the default)
- Corresponds to Prove menus in IDEs
- GNATPROVE applies proof to each subprogram separately
  - Notion of functional contracts summarize effects of call
- Outputs messages:
  - Check messages need to be reviewed, and either fixed or justified
  - Warnings can be inspected and silenced

Categories of Messages

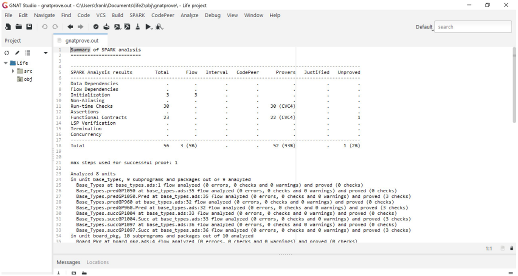
- *Error messages* start with `error:`
  - GNATPROVE aborts analysis and exits with error status
- *Check messages* start with severity `high:`, `medium:` or `low:`
  - With switch `--checks-as-errors=on`, GNATPROVE exits with error status
- *Warnings* start with `warning:`
  - With switch `--warnings=error`, GNATPROVE exits with error status
  - Some warnings are guaranteed to be issued
- *Information messages* start with `info:`
  - Report proved checks with switch `--report=all`
  - Report information about analysis with switch `--info`

GNATprove Output for Users



Analysis Summary File `gnatprove.out`

- Located in `gnatprove/` under project object dir
- An overview of results for all checks in project
- Especially useful when results must be documented
- Details in SPARK User's Guide

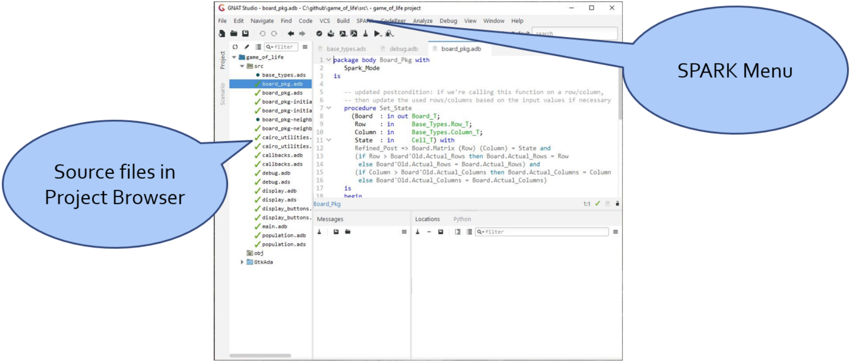


IDEs for SPARK

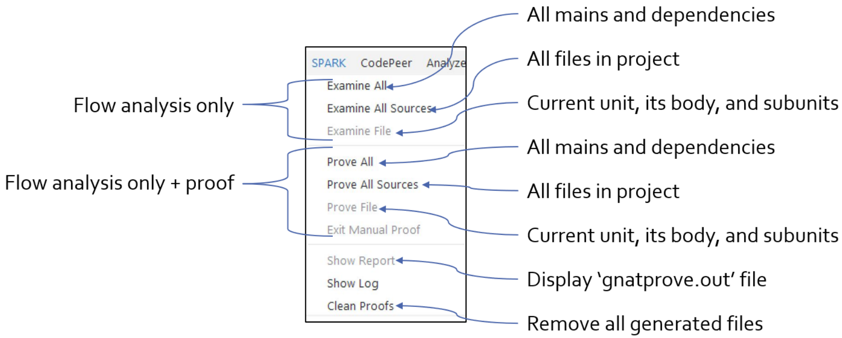
Three Available IDEs Supporting SPARK

- GNAT Studio
  - The AdaCore flagship IDE
  - **Best** integration overall
    - Most interaction capabilities
    - Specialized display of rich messages
    - Display of traces and counterexamples
- GNATbench for Eclipse
  - If you are already using Eclipse
- Ada/SPARK extension for Visual Studio Code
  - If you are already using VS Code

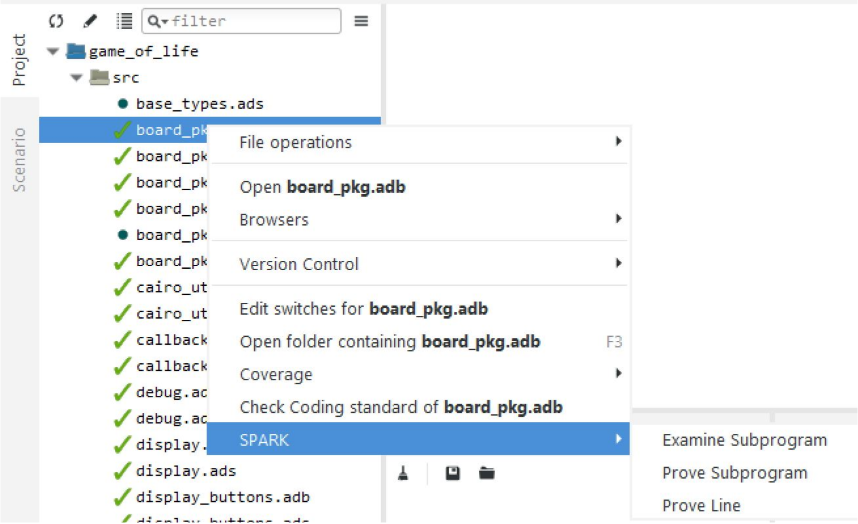
Basic GNAT STUDIO Look and Feel



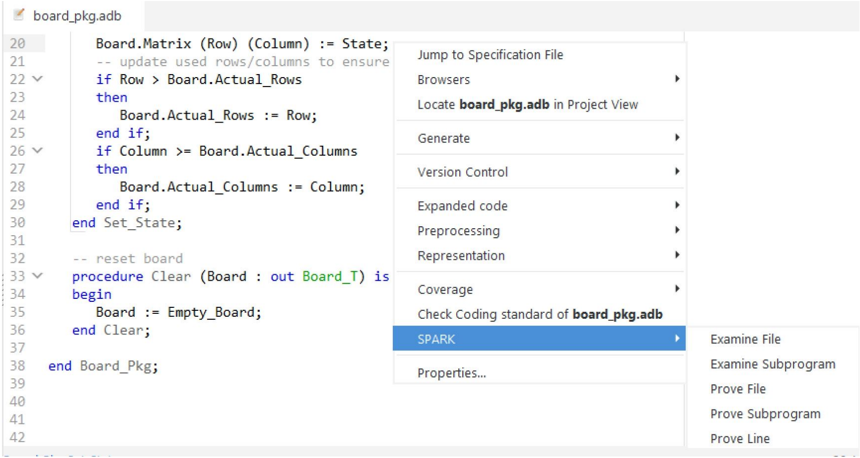
GNATPROVE SPARK Main Menu



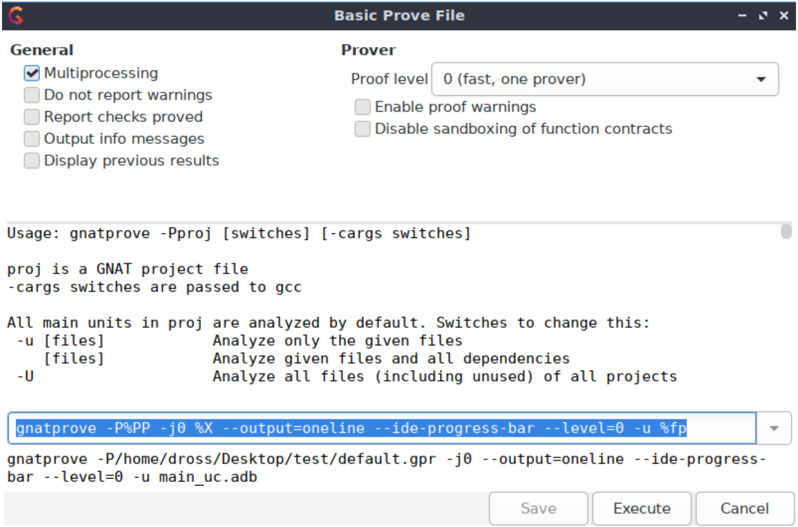
Project Tree Contextual Menu



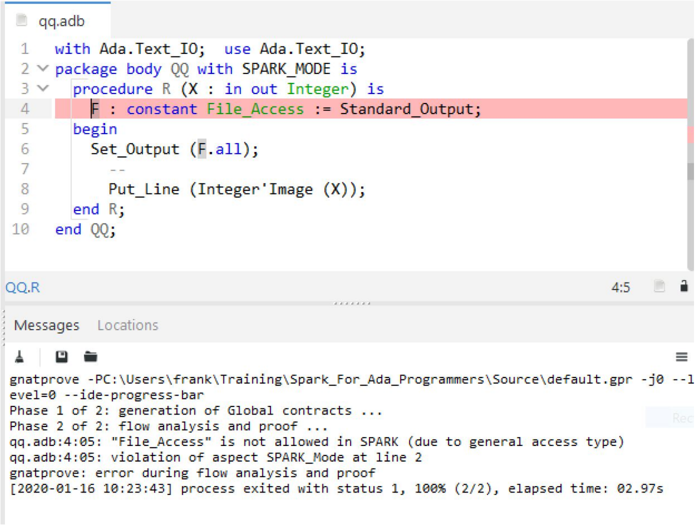
Source Code Contextual Menu



"Basic" Proof Dialog Panel

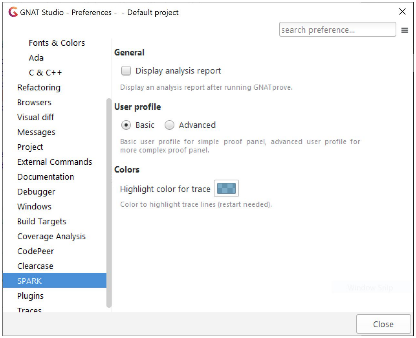


Example Analysis Results in GNAT STUDIO

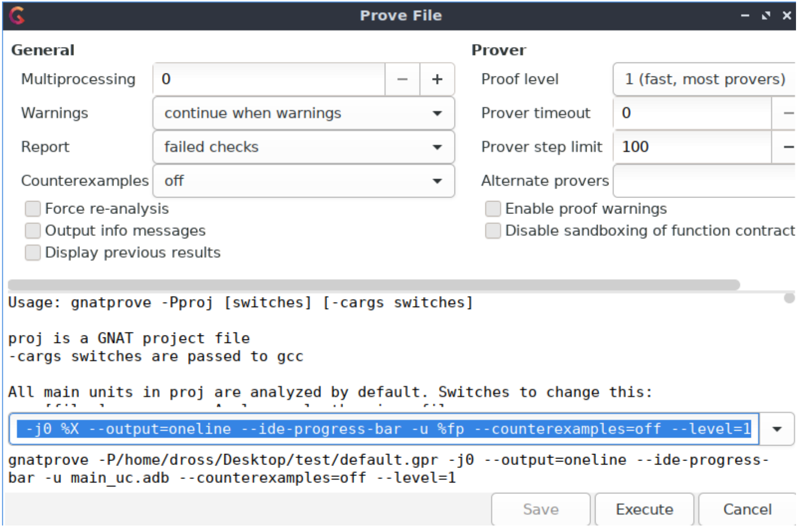


Preference for Selecting Profile

- Controlled by SPARK preference "User profile"
  - Basic
  - Advanced
- Allow more control and options
  - Prover timeout (seconds)
  - Prover steps (effort)
  - Etc.



"Advanced" Proof Dialog Panel



Lab

SPARK Tutorial

- Open the SPARK User's Guide
  - From your SPARK release (under menu Help → SPARK → SPARK User's Guide in GNAT STUDIO)
  - Or online at https://www.adacore.com/documentation
- Go to section 6 about the SPARK Tutorial
- Follow intructions to use the development and analysis tools
- Discuss these with the instructor

Summary

SPARK Tools

- Development tools for SPARK are those for Ada
- Analysis tools in GNATprove
  - Flow analysis
  - Proof
- Project files supports both command-line and IDEs use
  - Package `Prove` specific to GNATprove
  - Possibility to indicate that all code is in SPARK by default
- All integrated in multiple IDEs
  - But GNAT Studio provides the best integration

Flow Analysis

Introduction

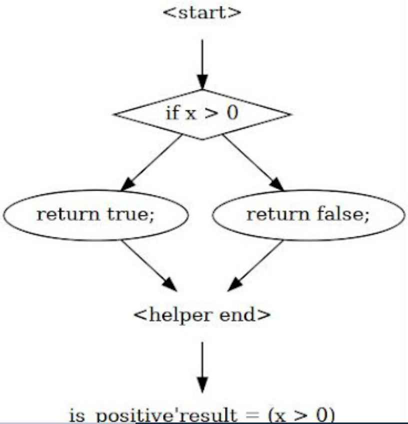
What Is Flow Analysis?

- **First** static analysis performed by GNATprove
- Models the **variables** used by a subprogram
  - Global variables
  - Scope variables (local variables of enclosing scope)
  - Local variables
  - Formal parameters
- Models how **information flows** through the statements in the subprogram
  - From initial values of variables
  - To final values of variables
- Performs checks and detects **violations**

Control Flow Graph (CFG)

- A representation, using **graph notation**, of all **paths** that might be traversed through a program during its execution [Wikipedia]

```
function Is_Positive
  (X : Integer)
  return Boolean
with Post =>
  Is_Positive'Result = (X > 0)
is
begin
   if X > 0 then
      return True;
   else
      return False;
   end if;
```

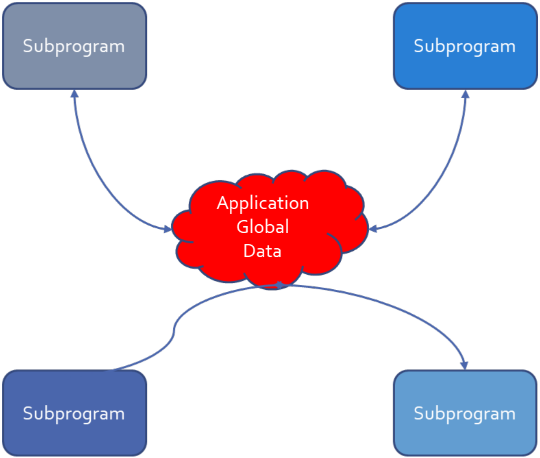


Program Dependence Graph (PDG)

- **Extension** of the CFG with information on **data flows**
- Control Dependence Graph
    - Compute post-dominators nodes: a node z is said to post-dominate a node n if **all** paths to the exit node of the graph starting at n must go through z
- Data Dependence Graph
    - Compute def-use chains rooted at variable definitions
- Transitive Dependence Graph
    - Compute how outputs of a call depend on its inputs
- Flow analysis checks are translated into **queries** on the PDG

Flow Analysis

Uncontrolled Data Visibility Problem



- Effects of changes are **potentially pervasive** so one must understand everything before changing anything

Data Dependency Contracts

- Introduced by aspect `Global`
- Optional, but must be complete if specified
- Optional mode can be `Input` (default), `Output`, `In_Out` or `Proof_In`

```
procedure Proc
with
  Global => (Input    => X,
             Output   => (Y, Z),
             In_Out   => V,
             Proof_In => W);
```

- `Proof_In` used for inputs **only** referenced in **assertions**
- `Global => null` used to state that no global variable is read/written
- Functions can have only `Input` and `Proof_In` global variables
  - Remember: no side-effects in functions!

Data Initialization Policy

- Subprogram *inputs* are input parameters and globals
  - parameters of mode `in` and `in out`
  - global variables of mode `Input` and `In_Out`
- Subprogram *outputs* are output parameters and globals
  - parameters of mode `out` and `in out`
  - global variables of mode `Output` and `In_Out`
- Inputs should be completely initialized **before** a call
- Outputs should be completely initialized **after** a call
- Stricter policy than in Ada
  - Allows **modular analysis** of initialization
  - Relaxed initialization will be seen in course on Advanced Proof

Stricter Parameter Modes

Initial Read - Initial value read

Partial Write - Object partially written: either part of the object written, or object written only on some paths, or both

Full Write - Object fully written on all paths

Initial Read	Partial Write	Full Write	Parameter Mode
✓			`in`
✓	✓		`in out`
✓		✓	`in out`
	✓		`in out`
		✓	`out`

- Similar rules for modes of global variables

Violations of the Data Initialization Policy

- Parameter only partially written should be of mode `in out`

```
procedure Cond_Init
  (X    : out T;
   -- Incorrect
   Cond : Boolean)
is
begin
   if Cond then
      X := ..;
   end if;
end Cond_Init;
```

- Global variable only partially written should be of mode `In_Out`

```
X : T;
procedure Cond_Init
  (Cond : Boolean)
with
  Global => (Output => X)
  -- Incorrect
is
begin
   if Cond then
      X := ..;
   end if;
end Cond_Init;
```

Generation of Data Dependency Contracts

- GNATprove computes a correct approximation
  - Based on the implementation
  - Using either specified or generated contracts for calls
  - More precise generation for SPARK code than for Ada code
- Generated contract may be imprecise
  - Output may be computed as both input and output
    - Because it is not known if the initial value is read
    - Because it is not known if the object is fully written on all paths
  - Precision can be recovered by adding a user contract

Bronze Level

- Check that each object read has been initialized
- Check that code respects data dependency contracts

```
procedure Swap (X, Y : in out Integer)
with
  Global => null; -- Wrong

procedure Swap (X, Y : in out Integer) is
begin
   Temp := X;
   X := Y;
   Y := Temp;
end Swap;
```

- Errors for most serious issues, need fixing for proof
- Warn on unused variables, ineffective statements

Flow Warnings

- Ineffective statement = statement without effects
  - Dead code
  - Or statement does not contribute to an output
  - Or effect of statement is hidden from GNATprove
- Warnings can be suppressed with pragma `Warnings`

```
pragma Warnings (Off, "statement has no effect",
                 Reason => "debug");
Debug_Print (X);
pragma Warnings (On, "statement has no effect");
```

- Optional first pragma argument `GNATprove` indicates it is specific to GNATprove

Limitations of Flow Analysis

Analysis of Value-Dependent Flows

- Flow analysis depends only on control flow, not on values
- Flow analysis is imprecise on value-dependent flows

```
procedure Absolute_Value
  (X : Integer;
   R : out Natural) -- Initialization check fails
is
begin
  if X < 0 then
    R := -X;
  end if;
  if X >= 0 then
    R := X;
  end if;
end Absolute_Value;
```

- Use control flow instead: use `if-then-else` above

Analysis of Array Initialization (1/2)

- Array indexes are values
- Flow analysis does not depend on values
- Flow analysis treats array assignment as a partial write
    - When assigning to an array index
    - When assigning to an array slice

```
type T is array (1 .. 10) of Boolean;

-- Initialization check fails
procedure Init_Array (A : out T) is
begin
   A (1) := True;
   A (2 .. 10) := (others => False);
end Init_Array;
```

- No such imprecision for record components

Analysis of Array Initialization (2/2)

- Use array aggregates when possible

```
type T is array (1 .. 10) of Boolean;

procedure Init_Array (A : out T) is -- Initialization check proved
begin
   A := (1 => True, 2 .. 10 => False);
end Init_Array;
```

- Do not please the tool! `A` is not `in out` here!
    - Otherwise, caller is forced to initialize `A`
- Some built-in heuristics recognize an initializing loop

```
procedure Init_Array (A : out T) is -- Initialization check proved
begin
   for J in A'Range loop
      A (J) := False;
   end loop;
end Init_Array;
```

Dealing with False Alarms

- Check messages can be justified with pragma `Annotate`

```
procedure Init_Array
  (A : out T) -- Initialization check justified
is
   pragma Annotate (GNATprove, False_Positive,
                    """A"" might not be initialized",
                    "value-dependent init");
```

- Justification inserted immediately after the check message location
- Relaxed initialization will be seen in course on Advanced Proof

Lab

Flow Analysis Lab

- Find the `050_flow_analysis` sub-directory in `source`
  - You can copy it locally, or work with it in-place
- In that directory, open the project `lab.gpr` in GNAT Studio
  - Or, on the command-line, do `gnatstudio -P lab.gpr`
- Unfold the source code directory (.) in the project pane

Aliasing and Initialization

- Find and open the files `basics.ads` and `basics.adb` in GNAT Studio
- Study the code and see if you can predict what's wrong.
  - These examples illustrate the basic forms of flow analysis in SPARK.
- Use SPARK → Examine File... to analyse the body of package **Basics**.
- Click on the "Locations" tab to see the messages organised by unit.
- Make sure you understand the check messages that GNATprove produces.
  - Discuss these with the course instructor.
- Either change the code or justify the message with pragma `Annotate`.
  - The objective is to get no messages when running GNATprove.

Data Dependencies

- Run flow analysis. Right-click in the code to display the contextual menu. Display the data dependencies generated by GNATprove with the contextual menu SPARK → Globals → Show generated Global contracts.
  - Study the generated contracts and make sure you understand them.
- Add a null data dependencies contracts with aspect `Global => null` to all subprograms.
- Run flow analysis. Make sure you understand the check messages that GNATprove produces.
- Add correct data dependencies contracts with aspect `Global` to all subprograms.
  - The objective is to get no messages when running GNATprove.
- Rerun GNATprove with checkbox Report check proved selected.
  - Review the info messages and make sure you understand them.
- Modify the code or contracts and check that GNATprove detects mismatches between them. Make sure you understand the check messages that GNATprove produces.

Summary

Flow Analysis

- Flow analysis builds a Program Dependence Graph
- Flow analysis detects:
  - Interferences between parameters and global variables
  - Read of uninitialized variable
  - Violation of data dependency contracts (`Global`)
- Flow analysis allows to reach Bronze level
- Flow analysis is imprecise
  - On value-dependent flows
  - On array assignment to index/slice
  - During generation of data dependency contracts

Proof

Introduction

What Is Proof?

- **Second** static analysis performed by GNATprove
    - Depends on successful flow analysis
- Models the **computation** in a subprogram
- Models **assertions** in a subprogram
- Performs checks and detects **violations**
    - Generates **logical formulas**
        - aka Verification Conditions (VC)
        - aka Proof Obligations (PO)
    - Automatic provers check that the VC is valid (always true)
    - If not, a check message is emitted

Hoare Triples

- **Hoare triples** (1969) used to reason about program correctness
  - With pre- and postconditions
- Syntax: `{P} S {Q}`
  - `S` is a program
  - `P` and `Q` are **predicates**
  - `P` is the **precondition**
  - `Q` is the **postcondition**
- Meaning of `{P} S {Q}` triple:
  - If we start in a state where `P` is true and execute `S`, then `S` will terminate in a state where `Q` is true.

Quiz - Hoare Triples

Which one of these is **invalid**?

A. `{ X >= 3 } Y := X - 1 { Y >= 0 }`
B. `{ X >= 3 } Y := X - 1 { Y = X - 1 }`
C. `{ False } Y := X - 1 { Y = X }`
D. `{ X >= 3 } Y := X - 1 { Y >= 3 }`
E. `{ X >= 3 } Y := X - 1 { True }`

Quiz - Hoare Triples

Which one of these is **invalid**?

A. `{ X >= 3 } Y := X - 1 { Y >= 0 }`
B. `{ X >= 3 } Y := X - 1 { Y = X - 1 }`
C. `{ False } Y := X - 1 { Y = X }`
D. *`{ X >= 3 } Y := X - 1 { Y >= 3 }`*
E. `{ X >= 3 } Y := X - 1 { True }`

Explanations

A. `Y >= 2` entails `Y >= 0`
B. This is true independent of the precondition.
C. This is true independent of the postcondition.
D. **Invalid**: `Y >= 2` does not entail `Y >= 3`
E. This is true independent of the precondition.

VC Generation - Strongest Postcondition

- VC are generated using a *Strongest Postcondition Calculus*
- The strongest postcondition `Q` for a program `S` and a precondition `P` is such that:
  - `{P} S {Q}` is a valid Hoare triple
  - For every valid Hoare triple `{P} S {Q'}`, `Q` is **stronger** than `Q'`, i.e. `Q` implies `Q'`
- The strongest postcondition **summarizes** what is known at any program point
- The strongest postcondition is computed through a *predicate transformer*
  - Information is **propagated** from the precondition
  - VCs are generated each time a **check** is encountered

Quiz - Strongest Postcondition

Which one of these has a **Strongest Postcondition**?

A. `{ X >= 3 } Y := X - 1 { Y >= 0 }`
B. `{ X >= 3 } Y := X - 1 { Y = X - 1 }`
C. `{ X >= 3 } Y := X - 1 { Y >= 2 }`
D. `{ X >= 3 } Y := X - 1 { Y = X - 1 and Y >= 2 }`
E. `{ X >= 3 } Y := X - 1 { Y = X - 1 and X >= 3 }`

Quiz - Strongest Postcondition

Which one of these has a **Strongest Postcondition**?

A. `{ X >= 3 } Y := X - 1 { Y >= 0 }`
B. `{ X >= 3 } Y := X - 1 { Y = X - 1 }`
C. `{ X >= 3 } Y := X - 1 { Y >= 2 }`
D. ***`{ X >= 3 } Y := X - 1 { Y = X - 1 and Y >= 2 }`***
E. ***`{ X >= 3 } Y := X - 1 { Y = X - 1 and X >= 3 }`***

Explanations

A. Information about `X` is lost.
B. Information about `X` is lost.
C. Information about `X` is lost.
D. Correct
E. Correct (equivalent to answer D)

Proof

Functional Contracts

- Precondition introduced by aspect `Pre`
  - Boolean expression stating **constraint on the caller**
  - Constraint on the value of inputs
- Postcondition introduced by aspect `Post`
  - Boolean expression stating **constraint on the subprogram**
  - Constraint on the value of inputs and outputs
- On the first declaration of a subprogram
  - This can be a spec or a body
- Optional, default is `True`
  - Precondition: subprogram can be called in any context
  - Postcondition: subprogram gives no information on its behavior
- Special attributes in postconditions
  - `X'Old` denotes the input value of `X`
  - `F'Result` denotes the result of function `F`

Silver/Gold/Platinum Levels

- Check absence of runtime errors (AoRTE)
- Check that assertions are always true
- Check that code respects functional contracts

```
procedure Swap (X, Y : in out Integer)
with
  Post => X = Y'Old and Y = X'Old; -- Wrong

procedure Swap (X, Y : in out Integer) is
begin
   Temp := Y;
   X := Y;
   Y := Temp;
end Swap;
```

- Warn on dead code with switch `--proof-warnings`
  - More powerful than the detection by flow analysis

Run-Time Errors Are Pervasive

- A simple assignment statement
  `A (I + J) := P / Q;`
- Which are the possible run-time errors for this example?

- `I+J` might overflow the base type of the index range's subtype
- `I+J` might be outside the index range's subtype
- `P/Q` might overflow the base type of the component type
- `P/Q` might be outside the component subtype
- `Q` might be zero

Categories of Run-Time Errors

- Divide by zero
  - Arithmetic operations: division, `mod`, `rem`
- Index check
  - Read/write access in an array
- Overflow check
  - Most arithmetic operations
  - Checking that result is within bounds of the machine integer or float
- Range check
  - Type conversion, type qualification, assignment
  - Checking that the value satisfies range constraint of type
- Discriminant check
  - Read/write access in a discriminated record
- Length check
  - Assignment of an array or string
- Checks on pointer programs - Details in the course on Pointer Programs

Quiz - Special Cases of Run-Time Errors

Consider the following declarations:

```
type Table is array (Natural range <>) of Integer;
type Rec (Disc : Boolean) is record ...
T : Table := ...;
R : Rec := ...;
X : Integer;
```

Which of the following *cannot* cause a runtime error:

A. `X := T (T'First)`
B. `X := X / (-1);`
C. `X := abs X;`
D. `X := T'Length;`
E. `R := (Disc => True, ...);`

Quiz - Special Cases of Run-Time Errors

Consider the following declarations:

```
type Table is array (Natural range <>) of Integer;
type Rec (Disc : Boolean) is record ...
T : Table := ...;
R : Rec := ...;
X : Integer;
```

Which of the following *cannot* cause a runtime error:

A. `X := T (T'First)`
B. `X := X / (-1);`
C. `X := abs X;`
D. `X := T'Length;`
E. `R := (Disc => True, ...);`

Explanations: **all** of then can cause a runtime error!

A. Index check fails if `T` is empty.
B. Overflow check fails if `X = Integer'First`
C. Overflow check fails if `X = Integer'First`
D. Range check fails if `T'Range` is `Natural`
E. Discriminant check fails if `R.Disc /= True`

Categories of Assertions

- Pragma `Assert` and similar (`Assert_And_Cut`, `Assume`)
  - AoRTE is also proved for its expression
- Precondition on call
  - AoRTE is also proved for **any** calling context
  - This may require **guarding** the precondition

```
procedure Update (T : in out Table; X : Index; V : Value)
  with Pre => T (X) /= V; -- Index check might fail
  with Pre => X in T'Range and T (X) /= V; -- Same
  with Pre => X in T'Range and then T (X) /= V; -- OK
```

- Postcondition on subprogram
  - AoRTE is proved in the context of the subprogram **body**
  - Still better to include info for AoRTE in **caller**

```
procedure Find (T : Table; X : out Index; V : Value)
  with Post => T (X) = V; -- Not known that X in T'Range
  with Post => X in T'Range and then T (X) = V; -- OK
```

Levels of Software Assurance

- Silver level
  - Goal is **absence** of runtime errors
  - Functional contracts added to support that goal
    - Typically a few preconditions only

```
procedure Update (T : in out Table; X : Index; V : Value)
  with Pre => X in T'Range;
```

- Gold level
  - Builds on the Silver level
  - Functional contracts added to **express desired properties**

```
procedure Update (T : in out Table; X : Index; V : Value)
  with Pre  => X in T'Range,
       Post => T (X) = V;
```

- Platinum level
  - Same as Gold level
  - But the **full** functional specification is expressed as contracts

```
procedure Update (T : in out Table; X : Index; V : Value)
  with Pre  => X in T'Range,
       Post => T = (T'Old with delta X => V);
```

Preconditions

- Default precondition of `True` may **not** be sufficient

```
procedure Increment (X : in out Integer) is
begin
   X := X + 1; -- Overflow check might fail
end Increment;
```

- Precondition constrains **input context**

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

Postconditions

- Default postcondition of `True` may **not** be sufficient

```
procedure Add2 (X : in out Integer)
with
  Pre => X < Integer'Last - 1
is
begin
   Increment (X);
   Increment (X); -- Precondition might fail
end Add2;
```

- Postcondition constrains **output context**

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

procedure Add2 (X : in out Integer)
with
  Pre => X < Integer'Last - 1
is
begin
   Increment (X);
   Increment (X); -- Precondition proved
end Add2;
```

Contextual Analysis of Local Subprograms

- Local subprograms without contracts are *inlined* in proof
    - Local: declared inside private part or body
    - Without contracts: no `Global`, `Pre`, `Post`, etc.
    - Additional conditions, details in the SPARK User's Guide
- Benefit: no need to add a contract
- Possible cost: proof of caller may become more complex
    - Add explicit contract like `Pre => True` to disable inlining of a subprogram
    - Use switch `--no-inlining` to disable this feature globally
    - Use switch `--info` to get more information about inlined subprograms

Overflow Checking (1/2)

- Remember: assertions might fail overflow checks

```
procedure Saturate_Add (X, Y : Natural; Z : out Natural)
  with Post => Z = Integer'Min (X + Y, Natural'Last);
```

- Sometimes property can be expressed to avoid overflows

```
procedure Saturate_Add (X, Y : Natural; Z : out Natural)
  with Post => Z =
    (if X <= Natural'Last - Y then X + Y else Natural'Last);
```

- Or a larger integer type can be used for computations

```
subtype LI is Long_Integer;

procedure Saturate_Add (X, Y : Natural; Z : out Natural)
  with Post => LI(Z) =
    LI'Min (LI(X) + LI(Y), LI(Natural'Last));
```

Overflow Checking (2/2)

- Alternative: use a library of big integers
  - From SPARK Library `SPARK.Big_Integers`
  - Or Ada stdlib: `Ada.Numerics.Big_Numbers.Big_Integers`

```
function Big (Arg : Integer) return Big_Integer is
  (To_Big_Integer (Arg)) with Ghost;
procedure Saturate_Add (X, Y : Natural; Z : out Natural)
  with Post => Z =
    (if Big (X) + Big (Y) <= Big (Natural'Last)
     then X + Y else Natural'Last);
```

- Or use compiler switch `-gnato13` to use big integers in all assertions
  - Implicit use
  - Should be used also when compiling assertions
  - Only applies to arithmetic operations (not `Integer'Min`)

```
procedure Saturate_Add (X, Y : Natural; Z : out Natural)
  with Post => Z =
    (if X + Y <= Natural'Last then X + Y else Natural'Last);
```

Limitations of Proof

Functional Specifications

- **Non-functional** specifications **cannot** be expressed as contracts
  - Time or space complexity
  - Timing properties for scheduling
  - Call sequences
- But **automatons** can be encoded as contracts
  - Being in a given state is a functional property
  - Can use normal queries
    - e.g. contracts on `Ada.Text_IO` use `Is_Open`
  - Or ghost imported functions that cannot be executed
    - When query cannot be expressed in the code

Limitations of Automatic Provers - Arithmetic

- Provers struggle with non-linear arithmetic
  - Use of multiplication, division, `mod`, `rem`
  - e.g. monotonicity of division on positive values
  - Solution: use **lemmas** from the SPARK Lemma Library
- Provers struggle with mixed arithmetic
  - Mix of signed and modular integers
  - Mix of integers and floats
  - Solution: define lemmas for **elementary properties**

Limitations of Automatic Provers - Quantifiers

- Quantified expressions express property over a **collection**
  - Universal: `(for all I in T'Range => T(I) /= 0)`
  - Existential: `(for some I in T'Range => T(I) /= 0)`
- Provers struggle with **existential**
  - Need to exhibit a *witness* that satisfies the property
  - Solution: define a function that computes the witness
- Provers cannot **reason inductively**
  - Inductive reasoning deduces a property over integer `I`
    - If it can be proved for `I = 0`
    - If it can be proved for `I+1` from the property for `I`
  - Solution: lead the prover to this reasoning with a **loop**

Limitations of Automatic Provers - Proof Context

- Proof context for a check in a subprogram `S` is:
  - The contracts of all subprograms called by `S`
  - The body of `S` prior to the check
  - The logical modeling of all entities used in `S`
- Proof context can become **too large**
  - Thousands of lines in the VC
  - This can make the VC unprovable, or hard to prove
- Various solutions to reduce the proof context
  - Split the body of `S` in smaller subprograms
  - Extract **properties of interest** in lemmas
  - Use special SPARK features
    - Pragma `Assert_And_Cut`
    - SPARK Library `SPARK.Cut_Operations`
    - SPARK annotation `Hide_Info`

Cost/Benefit Analysis

- Not all provable properties are worth proving!
- Difficulty of proof (cost) not correlated with benefit
- e.g. proving that a sorting algorithm preserves the components
  - Trivial by review if the only operation is `Swap`
  - May require many **annotations** for proof
- Functional correctness of complex algorithms is **costly**
  - Specifications can be larger than code
  - Annotations typically much larger than code ($\times$ 10)

Dealing with False Alarms

- Check messages can be justified with pragma `Annotate`

```
pragma Annotate (GNATprove, Category, Pattern, Reason);
```

  - `GNATprove` is a fixed identifier
  - `Category` is one of `False_Positive` or `Intentional`
    - `False_Positive`: check cannot fail
    - `Intentional`: check can fail but is not a bug
  - `Pattern` is a substring of the check message
    - Asterisks `*` match zero or more characters in the message
  - `Reason` is a string literal for reviews
    - `Reason` is repeated in output with switch `--report=all` and in analysis summary file `gnatprove.out`

- Justification inserted immediately after the check message location
  - Or at the beginning of a scope
    - Applies to all the scope
    - Generally used when not suitable after the check message location

Lab

Proof Lab

- Find the `060_proof` sub-directory in `source`
  - You can copy it locally, or work with it in-place
- In that directory, open the project `lab.gpr` in GNAT Studio
  - Or, on the command-line, do `gnatstudio -P lab.gpr`
- Unfold the source code directory (.) in the project pane

Absence of Runtime Errors

- Find and open the files `basics.ads` and `basics.adb` in GNAT Studio
- Study the code and see if you can predict what's wrong.
  - These examples illustrate the basic forms of proof in SPARK.
- Use SPARK → Prove File... to analyse the body of package **Basics**.
- Click on the "Locations" tab to see the messages organised by unit.
- Make sure you understand the check messages that GNATprove produces.
  - Discuss these with the course instructor.
- Add preconditions to avoid runtime errors in subprograms
  - Hint: use function `Value_Rec` for procedures `Bump_Rec` and `Bump_The_Rec`
  - The objective is to get no messages when running GNATprove.

Functional Specifications (1/2)

- Add a postcondition to procedure `Swap_The_Table` stating that the values at indexes `I` and `J` have been exchanged.
- Run proof. Make sure you understand the check messages that GNATprove produces.
  - Study the generated contracts and make sure you understand them.
- Add a postcondition to procedure `Swap_Table` stating that the values at indexes `I` and `J` have been exchanged.
- Run proof.
  - The postcondition on procedure `Swap_The_Table` should be proved now.
  - Add a postcondition to procedure `Swap` to complete the proof.
- Add similarly a postcondition to procedures `Bump_The_Rec` and `Bump_Rec` stating that the value of component `A` or `B` (depending on the value of the discriminant) has been incremented
  - Hint: use again function `Value_Rec`

Functional Specifications (2/2)

- Add similarly a postcondition to procedures `Init_The_Rec` and `Init_Rec` stating that the value of component `A` or `B` (depending on the value of the discriminant) is 1.
- Add similarly a postcondition to procedures `Init_The_Table` and `Init_Table` stating that the value of the first and last components are 1 and 2.
    - Hint: you may have to strengthen the precondition of `Init_Table`.
- Rerun GNATPROVE with checkbox Report check proved selected.
    - Review the info messages and make sure you understand them.
- Modify the code or contracts and check that GNATPROVE detects mismatches between them. Make sure you understand the check messages that GNATPROVE produces.

Summary

Proof

- Proof uses Strongest Postcondition Calculus to generate formulas
- Formulas aka Verification Conditions (VC) are sent to provers
- Proof detects:
  - Possible run-time errors
  - Possible failure of assertions
  - Violation of functional contracts (`Pre` and `Post`)
- Proof allows to reach Silver/Gold/Platinum levels
- Proof is imprecise
  - On non-linear arithmetic and mixed arithmetic
  - On existential quantification and inductive reasoning
  - When the proof context is too large

Specification Language

Introduction

Simple Expressions

- Simple specifications use **simple** expressions
    - Arithmetic operations and comparisons
    - Membership tests `X in A .. B`

      ```
      I in T'Range
      ```

      is better than:

      ```
      I >= T'First and I <= T'Last
      ```

    - Conjunctions and disjunctions
        - Lazy operators `and then`/`or else` preferred in general to `and`/`or`
- But that's not sufficient to easily write **all** specifications

```
procedure Init_Table (T : out Table)
with
  Pre  => T'Length > 0,
  Post => -- if T is of length 1 ...
          -- else if T is of length 2 ...
          -- else for all components ...
```

Richer Expressions

- Counterparts of **conditional** statements
  - *if expressions* are the counterpart of *if statements*
  - *case expressions* are the counterpart of *case statements*
- Expressions over a **collection** (range or array or...)
  - *universally quantified expression* for properties over **all** components
  - *existentially quantified expression* for properties over **one** component
- New forms of **aggregates**
  - *delta aggregates* express the value of an updated composite object
  - *iterated component associations* express array aggregates where the expression depends on the **index**
  - *container aggregates* give the value of a container
- Structuring expressions
  - *declare expressions* introduce **names** for local constants
  - *expression functions* introduce **names** for common expressions

Conditional Expressions

If Expressions

- `(if Cond then A else B)` evaluates `A` or `B` depending on the value of `Cond`
  - Note: always in **parentheses**!
  - `A` and `B` must have the same type
  - ...not always `Boolean`!

    ```
    A := (if Cond then 2 else 3);
    ```
- Frequent use with `Boolean` type in specifications
  - `(if Cond then Property)` is shortcut for `(if Cond then Property else True)`
  - This expresses a **logical implication** `Cond` $\rightarrow$ `Property`
  - Also equivalent to `not Cond or else Property`
- Complete form has `elsif` parts

Case Expressions

- Extension of *if expressions* to non-Boolean discrete types

```
(case Day is
   when Monday
      | Friday
      | Sunday    => 6,
   when Tuesday   => 7,
   when Thursday
      | Saturday  => 8,
   when Wednesday => 9)
```

- Same **choice expressions** as in *case statements*
  - Can also use `others` as last alternative
  - Note: always in parentheses!
  - Note: cases are separated by commas

Set Notation

- Usable in both *case expressions* / *case statements* and in membership tests
- Without set notation:

```
if X = 'A' or else X = 'B' or else X = 'C' then
```

- With set notation:

```
if X in 'A' | 'B' | 'C' then
```

- Also allowed for opposite membership test: `if X not in ...`

Quantified Expressions

Range-based Form

- Based on the usual *for loop* syntax over a range

```
for J in T'Range loop
   T (J) := 0;
end loop;
pragma Assert (for all J in T'Range => T(J) = 0);
```

- Universally quantified expression
  `(for all J in A .. B => Property)`
  - Express that property holds for **all** values in the range
  - True if the range is empty (∀ in logic)
  - At runtime, executed as a loop which stops at first value where the property is not satisfied
- Existentially quantified expression
  `(for some J in A .. B => Property)`
  - Express that property holds for **at least one** value in the range
  - False if the range is empty (∃ in logic)
  - At runtime, executed as a loop which stops at first value where the property is satisfied

Array-based Form

- Based on the *for loop* syntax over an array

```
for E of T loop
   E := 0;
end loop;
pragma Assert (for all E of T => E = 0);
```

- Counterparts of range-based forms
  - Universally quantified expression
    `(for all E of T => Property)`
  - Existentially quantified expression
    `(for some E of T => Property)`
- Note: always in **parentheses**!

Range-based Vs Array-based Forms

- Array-based form only possible if `Property` does **not** refer to the **index**
- Example: array `T` is sorted

```
(for all J in T'Range =>
  (if J /= T'First then T(J-1) <= T(J)))
```

or (better for proof to avoid the need for induction)

```
(for all J in T'Range =>
  (for all K in T'Range =>
    (if J < K then T(J) <= T(K))))
```

General Iteration Mechanism

- **Based** on the `Iterable` aspect on a type
  - **Not the same** as the standard Ada mechanism!
  - **Simpler** mechanism adopted for the SPARK formal containers

```
type Container is private with
  Iterable => (First       => First,
               Next        => Next,
               Has_Element => Element
               Element     => Element);
```

- *Iteration over positions* uses `for .. in` syntax
- Uses cursor type with `First`, `Next` and `Has_Element`
- Function `Element` is **not** required
- *Iteration over components* uses `for .. of` syntax
  - Based on the previous iteration
  - Function `Element` retrieves the **component** for a given cursor

Iteration Over Formal Containers

- **Generic** units compatible with SPARK
  - The API is slightly different from standard Ada containers
  - Available in the SPARK Library
- Available for **all** formal containers:
  - vectors
  - doubly linked lists
  - sets (hashed and ordered)
  - maps (hashed and ordered)
- Iteration over positions
  - Access to **component** through function `Element`
  - For maps, access to **key** through function `Key`
- Iteration over components
  - For maps, really an iteration over **keys**
    - Use another function `Element` to get **component**

Iteration Over Formal Vectors

- Only formal container to have 3 iteration mechanisms
- Range-based iteration (using `-gnatX` for dot-notation)

```
for J in V.First_Index .. V.Last_Index loop
   V.Replace_Element (J, 0);
end loop;
pragma Assert
  (for all J in V.First_Index .. V.Last_Index => V.Component (J) = 0);
```

- Iteration over positions

```
for J in V loop
   V.Replace_Element (J, 0);
end loop;
pragma Assert (for all J in V => V.Element (J) = 0);
```

- Iteration over components (**no update**!)

```
for E of V loop
   pragma Assert (E = 0);
end loop;
pragma Assert (for all E of V => E = 0);
```

New Aggregate Expressions

Delta Aggregates

Ada 2022

- Express the value of a **modified** composite object (record or array)

```
(Rec with delta Comp1 => Val1, Comp2 => Val2)
(Arr with delta 1 => True, 42 => False)
```

- Typically used to relate input and output **values** of parameters
  - Combines delta aggregate with use of attribute `'Old`

```
procedure P (Rec : in out T)
  with Post => Rec = (Rec'Old with delta Comp1 => Val1,
                                         Comp2 => Val2);
```

- With array object:
  - Avoids the introduction of **explicit** quantifiers
  - Can have **overlapping** and **dynamic** choices (values or ranges)

Extension of Delta Aggregates

GNAT Extension

- GNAT extension allowed using either
  - switch `-gnatX0`, or
  - pragma `Extensions_Allowed (All)`
- Choice can be a subcomponent of the record or array

```
(Rec with delta Comp1.Sub1 => Val1,
                Comp2.Sub2.Subsub => Val2)
(Arr with delta (1).Sub => True,
                (42).Subarr(4) => False)
```

Iterated Component Associations

Ada 2022

- Express the **value** of an array aggregate depending on index
- Example: the *identity* function

```
(for J in T'Range => J)
```

- This is a *component association*
  - Can be used in **any** aggregate
  - Can be mixed with regular component associations `Idx => Val`

Container Aggregates

Ada 2022

- Available for all functional and formal containers
- Vectors, lists and sets use the positional syntax:

```
V : Vector := [1, 2, 3];
L : List := [1, 2, 3];
S : Set := [1, 2, 3];
```

- Maps use the named syntax:

```
M : Map := [1 => 8, 4 => 3, 42 => 127];
```

- General mechanism using the `Container_Aggregates` annotation
  - Three predefined patterns `Predefined_Sequences`, `Predefined_Sets` and `Predefined_Maps` require specific API (used for functional containers)
  - `From_Model` only requires `Model` function returning the above (used for formal containers)
  - consistency checked by GNATprove

Structuring Expressions

Declare Expressions

Ada 2022

- Convenient shorthand for **repeated** subexpression
  - Only constants and renamings allowed
  - Typically used in **postconditions**

```
function Find (T : Table; R : Integer) return Integer
  with Post =>
    (declare
       Res : constant Integer := Find'Result;
     begin
       Res >= 0 and then
       (if Res /= 0 then T (Res) = R));
```

Expression Functions

- Convenient shorthand for **repeated** subexpression
  - Somewhat similar goal as declare expressions
  - But visible in a **larger** scope
- Simple query functions used in contracts

```
function Is_Sorted (T : Table) return Boolean is
  (for all J in T'Range =>
     (for all K in T'Range => (if J < K then T(J) <= T(K))));
```

- Above is equivalent to having a **postcondition**
  - But no subprogram body to add in the body unit

```
function Is_Sorted (T : Table) return Boolean
  with Post => Is_Sorted'Result = (for all J in T'Range => ...);
```

- Pre and postconditions can be specified **after** the expression

```
function Is_Sorted (T : Table) return Boolean is (...)
  with Pre => T'Length > 0;
```

Use of Expression Functions

- Expression functions can be declared in a package spec and used in **contracts**
  - It can even be declared **after** its use in contracts!
- For queries over objects of a `private` type
  - Function **spec** is declared in the **public** part
  - **Expression function** is declared in the **private** part

```
package P is
  type T is private;
  function Value (X : T) return Integer;
private
  type T is new Integer;
  function Value (X : T) return Integer is (Integer (X));
end;
```

  - GNATprove uses the **implicit postcondition** to prove client units

Lab

Specification Language Lab

- Find the `070_specification_language` sub-directory in `source`
    - You can copy it locally, or work with it in-place
- In that directory, open the project `lab.gpr` in GNAT Studio
    - Or, on the command-line, do `gnatstudio -P lab.gpr`
- Unfold the source code directory (.) in the project pane

Richer Expressions

- Find and open the files `basics.ads` and `basics.adb` in GNAT Studio
    - After each modification, check that the code is still proved by **GNATprove**
- Use a *declare expression* to introduce names `X_Old` for `X'Old` and `Y_Old` for `Y'Old` in the postcondition of `Swap`
- Use *delta aggregates* to state the new value of `R` in the postcondition of `Bump_Rec`
    - Hint: use an *if expression* testing the value of the discriminant
- Use a *quantified expression* to state that all values in array `T` are preserved after the call to `Swap_Table`, except for those at indexes `I` and `J`
    - Hint: use a membership test for "being different from `I` and `J`"
    - Hint: notice that `T'Old(K)` may be allowed even if `T(K)'Old` is not

Expression Functions

- Define an expression function `Value_Rec_Is_One` to express the condition in the postcondition of `Init_Rec`
- Use `Value_Rec_Is_One` in the postcondition of `Init_Rec`
  - Check that the code is still proved
- Keep the declaration of `Value_Rec_Is_One` in the spec file, but move the expression function in the body file.
  - Is the code still proved?
- Turn the expression function of `Value_Rec_Is_One` into a regular function body.
  - Is the code still proved?
- Add a postcondition to the declaration of `Value_Rec_Is_One` into a regular function body.
  - Is the code proved again?
- Discuss these with the course instructor.

All Together

- Define a function `Constant_Value` that returns `True` if an array `T` has value `Value` between indexes `Start` and `Stop`
    - Hint: add a precondition to exclude incorrect parameter values
- Use `Constant_Value` in the postcondition of `Init_Table` to express that the table has value zero at all indexes except the first and last ones.
- Check that the code is still proved.

Summary

Specification Language

- Rich **specification language** in SPARK
  - Conditional expressions
  - Quantified expressions
  - New forms of aggregates
  - Structuring expressions
- Expression functions are handled **specially** in proof
  - Implicit postcondition given by their expression
- Expression functions define **queries** on private types
  - Function spec declared in the visible part
  - Expression function given in the private part
  - Preserves abstraction for user
  - Gives enough details for proof

Subprogram Contracts

Introduction

Programming by Contract

- Pioneered by programming language **Eiffel** in the 80's
  - Since then adopted in Ada, .NET
  - Also being discussed for C++, Rust
  - Available as libraries for many languages
- The *contract* of a subprogram defines:
  - What a caller guarantees to the subprogram (the precondition)
  - What the subprogram guarantees to its caller (the postcondition)
- A contract should include **all** the necessary information
  - Completes the API
  - Caller should **not** rely on **implementation details**
  - Typically parts of the contract are in English

Contracts in SPARK

- Preconditions and postconditions added in Ada 2012
  - Using the aspect syntax for `Pre` and `Post`
  - Already in GNAT since 2008 as pragmas
- Language support goes much **beyond** contracts-as-a-library
  - Ability to relate pre-state and post-state with attribute `Old`
  - **Fine-grained** control over execution

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

- GNATprove analysis based on contracts
  - Precondition should be sufficient to prove subprogram **itself**
  - Postcondition should be sufficient to prove **its callers**
  - ...at all levels of software assurance beyond Bronze!
- SPARK contracts by cases, for callbacks, for OOP, etc.

Frame Condition

Quiz - Stating the Obvious

What is the problem with this postcondition?

```
type Pair is record
   X, Y : Integer;
end record;

procedure Set_X (P : in out Pair; Value : Integer)
  with Post => P.X = Value;
```

Quiz - Stating the Obvious

What is the problem with this postcondition?

```
type Pair is record
   X, Y : Integer;
end record;

procedure Set_X (P : in out Pair; Value : Integer)
  with Post => P.X = Value;
```

- The postcondition does not say that the value of `Y` is preserved!
- As a result, nothing is known about `Y` after calling `Set_X`

```
P : Pair := Pair'(X => 1, Y => 2);
P.Set_X (42);
pragma Assert (P.Y = 2); -- unproved
```

Frame Condition - Records

- Simpler solution is to state which components are **preserved**

```
procedure Set_X (P : in out Pair; Value : Integer)
  with Post => P.X = Value and P.Y = P.Y'Old;
```

- Or with a **delta aggregate**

```
procedure Set_X (P : in out Pair; Value : Integer)
  with Post => P = (P'Old with delta X => Value);
```

- In both cases, value of `Y` is known to be preserved

Frame Condition - Arrays

- Use universal quantification to denote components preserved

```
procedure Swap_Table (T : in out Table; I, J : Index)
  with Post =>
    (for all K in T'Range =>
      (if K not in I | J then T (K) = T'Old (K)));
```

- Or with a delta aggregate

```
procedure Swap_Table (T : in out Table; I, J : Index)
  with Post =>
    T = (T'Old with delta I => T(J)'Old, J => T(I)'Old);
```

- In both cases, value of `T(K)` is known to be preserved for `K` different from `I` and `J`

Frame Condition - Conditions

- Any variable may be preserved conditionally
  - That applies also to scalar variables

```
procedure Zero_If (X : in out Integer; Cond : Boolean)
  with Post => (if Cond then X = 0);
```

- The preservation case needs to be **explicited**

```
procedure Zero_If (X : in out Integer; Cond : Boolean)
  with Post => (if Cond then X = 0 else X = X'Old);
```

- *Frame condition* is **all** the parts of objects that may be preserved
  - Bounded by user-defined or generated **data dependencies**
  - Anything else needs to be stated **explicitly**

Frame Condition - Bounds and Discriminants

- Some parts of objects **cannot** be changed by a call
  - Array bounds
  - Discriminants of constrained records
- Special handling in GNATprove to preserve them

```
type Rec (Disc : Boolean) is record ...

procedure Change (T : in out Table; R : in out Rec)
  with Post =>
    T'First = T'First'Old          -- redundant
    and then T'Last = T'Last'Old   -- redundant
    and then R.Disc = R.Disc'Old;  -- redundant
```

Frame Condition - Private Types

- Direct access to value or components not possible
- Simpler solution: define **query functions**
  - **Hide** access to value or components

```
type Pair is private;
function Get_Y (P : Pair) return Integer;
procedure Set_X (P : in out Pair; Value : Integer)
  with Post => P.Get_Y = P.Get_Y'Old;
```

- More comprehensive solution: define **model functions**
  - Create a visible **model** of the value

```
type Pair is private;
type Pair_Model is record X, Y : Integer; end record;
function Model (P : Pair) return Pair_Model;
procedure Set_X (P : in out Pair; Value : Integer)
  with Post => P.Model = (P.Model'Old with delta X => Value);
```

Attribute Old

- Dynamic semantics is to make a copy at subprogram entry
  - Forbidden on `limited` types
- Evaluation for the copy may raise runtime errors
  - Not allowed by default inside *potentially unevaluated expressions*
    - Unless prefix is a variable

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

- Use `pragma Unevaluated_Use_Of_Old (Allow)` to allow
  - GNATprove **checks** that this is safe

Special Cases for Attribute `Old`

- Simple component access `X.C'Old` equivalent to `X'Old.C`
  - Although one may be more efficient at runtime
- Function call in the prefix of `Old` is evaluated at subprogram entry
  - Value of **globals** is the one at subprogram entry
  - Not the same as calling the function on parameters with `Old`

```
function F (X : Integer) return Integer
  with Global => Glob;

procedure P (X : in out Integer)
  with Post =>
    F (X'Old) = 0 and then
    F (X)'Old = 0;
```

Contracts by Cases

Contract Cases (1/2)

- Some contracts are best expressed by cases
  - Inspired by *Parnas Tables*
- SPARK defines aspect `Contract_Cases`
  - Syntax of named aggregate
  - Each case consists of a guard and a consequence
- Example from SPARK tutorial

```
Contract_Cases =>
  (A(1) = Val                               => ...
   Value_Found_In_Range (A, Val, 2, 10)     => ...
   (for all J in Arr'Range => A(J) /= Val) => ...
```

Contract Cases (2/2)

- GNATPROVE checks that **each** case holds
  - When guard is enabled on entry, consequence holds on exit
  - Note: guards are evaluated **on entry**
  - Attributes `Old` and `Result` allowed in consequence
- GNATPROVE checks that cases are **disjoint** and **complete**
  - All inputs allowed by the precondition are covered by a single case
- When enabled at runtime:
  - Runtime check that exactly one guard holds on entry
  - Runtime check that the corresponding consequence hold on exit

Exceptional Cases

- Needed when exception propagation is expected

```
-- Constraint error in specific case
Exceptional_Cases =>
   (Constraint_Error => Status = Error);

-- All exceptions (most general form)
Exceptional_Cases => (others => True);
```

- Different exceptions can be grouped by cases

```
Exceptional_Cases =>
  (Constraint_Error | Numerical_Error => Post1,
   Program_Error                      => Post2);
```

- GNATprove checks that **each** case holds
  - When exception is raised, consequence holds on exit
  - Attribute `Old` allowed in consequence
- No runtime effect

Contracts and Refinement

What's Refinement?

- *Refinement* = relation between two representations
  - An *abstract* representation
  - A *concrete* representation
- Concrete behaviors are **included** in abstract behaviors
  - Analysis on the abstract representation
  - Findings are valid on the concrete one
- SPARK uses refinement
  - For analysis of **callbacks**
  - For analysis of **dispatching calls** in OOP
    - aka Liskov Substitution Principle (LSP)
- Generics do not follow refinement in SPARK
  - Reminder: instantiations are analyzed instead

Contracts on Callbacks

- Contracts can be defined on access-to-subprogram types
  - Only precondition and postcondition

```
type Update_Proc is access procedure (X : in out Natural)
with
  Pre  => Precond (X),
  Post => Postcond (X'Old, X);
```

- GNATPROVE checks refinement on **actual** subprograms

```
Callback : Update_Proc := Proc'Access;
```

  - **Precondition** of `Proc` should be **weaker** than `Precond(X)`
  - **Postcondition** of `Proc` should be **stronger** than `Postcond(X'Old, X)`
  - Data **dependencies** should be `null`
    - **No** use of globals
- GNATPROVE uses contract of `Update_Proc` when `Callback` is called

Contracts for OOP

- Inherited contracts can be defined on dispatching subprograms

```
type Object is tagged record ...
procedure Proc (X : in out Object) with
  Pre'Class  => Precond (X),
  Post'Class => Postcond (X'Old, X);
```

- GNATprove checks refinement on **overriding** subprograms

```
type Derived is new Object with record ...
procedure Proc (X : in out Derived) with ...
```

  - **Precondition** of `Proc` should be **weaker** than `Precond(X)`
  - **Postcondition** of `Proc` should be **stronger** than `Postcond(X'Old, X)`
  - Data **dependencies** should be the **same**

- GNATprove uses contract of `Proc` in `Object` when `Proc` is called with static type `Object`
  - Dynamic type might be `Derived`

Preventing Unsoundness

Quiz - Unsoundness

What's wrong with the following contract?

```
function Half (Value : Integer) return Integer
  with Post => Value = 2 * Half'Result;
```

Quiz - Unsoundness

What's wrong with the following contract?

```
function Half (Value : Integer) return Integer
  with Post => Value = 2 * Half'Result;
```

- The postcondition is false when `Value` is odd
- GNATprove generates an inconsistent axiom for `Half`
  - It says that any integer is equal to twice another integer
  - This can be used by provers to deduce `False`
  - **Anything** can be proved from `False`
    - As if the code was dead code

Unfeasible Contracts

- All contracts **should** be feasible
  - There exists a correct implementation
  - This includes absence of runtime errors
- Contract of `Double` also leads to **unsoundness**
  - The postcondition is false when `Value` is too large

```
function Double (Value : Integer) return Integer
  with Post => Double'Result = 2 * Value;
```

- GNATprove implements defense in depth
  - Axiom only generated for functions (not procedures)
  - Function **sandboxing** adds a guard to the axiom
    - Unless switch `--function-sandboxing=off` is used
  - Switch `--proof-warnings=on` can detect inconsistencies
  - Proof of subprogram will detect contract unfeasibility
    - **Except** when subprogram does not terminate

Non-terminating Functions

What's wrong with the following code?

```
function Half (Value : Integer) return Integer is
begin
   if True then
      return Half (Value);
   else
      return 0;
   end if;
end Half;
```

Non-terminating Functions

What's wrong with the following code?

```
function Half (Value : Integer) return Integer is
begin
   if True then
      return Half (Value);
   else
      return 0;
   end if;
end Half;
```

- Function `Half` does not terminate
- GNATprove proves the postcondition of `Half`!
  - Because that program point is unreachable (dead code)
- GNATprove does not generate an axiom for `Half`
  - Because function may not terminate
  - `info: function contract not available for proof`
  - Info message issued when using switch `--info`

Terminating Functions

- Functions should **always** terminate
- Specific contract to require proof of termination of procedures

```
procedure P
  with Always_Terminates => Condition;
```

- Flow analysis proves termination in **simple cases**
  - No (mutually) recursive calls
  - Only bounded loops
- **Proof** used to prove termination in remaining cases
  - Based on subprogram variant for recursive subprograms
  - Based on loop variant for unbounded loops

Subprogram Variants

- Specifies measure on recursive calls
  - Either increases or decreases strictly

```
function Half (Value : Integer) return Integer
  Subprogram_Variant =>
    (Increases => (if Value > 0 then -Value else Value)),
is
begin
   if Value in -1 .. 1 then
      return 0;
   elsif Value > 1 then
      return 1 + Half (Value - 2);
   else
      return -1 + Half (Value + 2);
   end if;
end Half;
```

- More complex cases use lexicographic order

```
Subprogram_Variant => (Decreases => Integer'Max(Value, 0),
                       Increases => Integer'Min(Value, 0)),
```

Quiz

Quiz - Frame Condition

Which statement is correct?

A. The frame condition is easily overlooked.
B. The frame condition is generated by GNATprove.
C. Delta aggregates are only used in frame conditions.
D. Attribute `Old` is illegal after **`and then`** or **`or else`**.

Quiz - Frame Condition

Which statement is correct?

A. ***The frame condition is easily overlooked.***
B. The frame condition is generated by GNATprove.
C. Delta aggregates are only used in frame conditions.
D. Attribute `Old` is illegal after **`and then`** or **`or else`**.

Explanations

A. Correct
B. Only part of the frame condition is generated.
C. No, but they are particularly useful in frame conditions.
D. Use pragma `Unevaluated_Use_Of_Old (Allow)`.

Quiz - Unsoundness

Which statement is correct?

A. All functions terminate by definition in SPARK.
B. An inconsistent axiom may be caused only by a non-terminating function.
C. The only protection against unsoundness is reviews.
D. A proved terminating subprogram cannot lead to unsoundness.

Quiz - Unsoundness

Which statement is correct?

A. All functions terminate by definition in SPARK.
B. An inconsistent axiom may be caused only by a non-terminating function.
C. The only protection against unsoundness is reviews.
D. ***A proved terminating subprogram cannot lead to unsoundness.***

Explanations

A. No, recursion and infinite loops may cause non-termination.
B. The contract may be unfeasible if the function is not proved.
C. GNATprove has multiple defenses against inconsistent axioms.
D. Correct

Summary

Subprogram Contracts

- Functional contracts given by
  - The precondition with aspect `Pre`
  - The postcondition with aspect `Post`
  - The contract cases with aspect `Contract_Cases`
  - The exceptional cases with aspect `Exceptional_Cases`
- Postcondition may be imprecise
  - In particular, **frame condition** might be missing
  - This may prevent **proof of callers**
- Function contracts may lead to unsoundness
  - If contract is unfeasible
  - If function does not terminate
  - Prove functions **and** their termination!

Type Contracts

Introduction

Range Constraints

Ada 83

- Scalar ranges gives **tighter** bounds to scalar types
  - Integer types: signed, modular
  - Real types: floating-point, fixed-point

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

- Also in standard subtypes `Natural` and `Positive`
- Range constraint also for enumeration and array types

```
subtype Week_Day is Day range Monday .. Friday;

type Index is range 1 .. 100;
type Table is array (Index range <>) of Integer;
subtype Table_10 is Table (1 .. 10);
```

Discriminant Constraints

Ada 83

- Record discriminants can be **specialized** to specific values
- Formal bounded containers from SPARK Library

```
type Vector (Capacity : Capacity_Range) is record ...
My_Vec : Vector (10);
```

- Discriminant without default cannot be changed
  - Needs to be defined at variable declaration
- Discriminant with default can be changed
  - If variable `Var` declared with unconstrained type
  - Then `Var'Constrained = False`

Richer Type Contracts

Ada 2012

- Predicates and invariants added in Ada 2012
  - Using the aspect syntax for `Predicate` and `Type_Invariant`
- Language support goes **much beyond** contracts-as-a-library
  - Constraint expressed once and verified *everywhere*
  - Fine-grain control over execution

```
pragma Assertion_Policy (Predicate => Check);
pragma Assertion_Policy (Type_Invariant => Ignore);
```

- GNATprove analysis based on contracts
  - Predicates and invariants assumed on subprogram inputs
  - Predicates and invariants proved on subprogram outputs
  - ...at all levels of software assurance beyond Bronze!

Type Predicates

What Is a Type Predicate?

- Boolean property that should **always hold** for objects of the type
  - Name of the type used to refer to an object of the type
  - Direct use of component names also allowed
- Can be specified on a type or subtype

```
type Non_Zero is new Integer
  with Predicate => Non_Zero /= 0;

subtype Even is Integer
  with Predicate => Even mod 2 = 0;
```

- Type predicate can be static or dynamic
  - Aspect `Predicate` can be `Static_Predicate` or `Dynamic_Predicate`

```
type Non_Zero is new Integer
  with Static_Predicate => Non_Zero /= 0;

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

- Like a type constraint, part of membership test `X in T`

Static Vs Dynamic Predicate

- **Static** predicates are **more restricted**
  - Boolean combination of comparisons with **static** values
  - Usable mostly on scalar and enumeration types
  - That does **not** mean statically checked by the compiler
- **Dynamic** predicates are **arbitrary** boolean expressions
  - Applicable to array and record types
- Types with static predicates are allowed in more contexts
  - Used as range in a *for loop*
  - Used as choice in *case statement* or *case expression*
- Aspect `Predicate` is GNAT name for:
  - `Static_Predicate` if predicate is static
  - `Dynamic_Predicate` otherwise

Useful Static Predicates

- Scalar ranges with **holes**

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

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

- Enumeration of scalar values

```
type Serial_Baud_Rate is range 110 .. 1200
  with Static_Predicate =>
    Serial_Baud_Rate in 110 | 300 | 600 | 1200;
```

- Enumeration ranges with holes

```
subtype Weekend is Day
  with Static_Predicate => Day in Saturday | Sunday;
```

Useful Dynamic Predicates (1/2)

- Array types with **fixed lower bound**

```
type Message is new String
  with Dynamic_Predicate => Message'First = 1;
```

  - Also possible with GNAT extension

```
type Message is new String(1 .. <>);
```

- Record with capacity discriminant and size component

```
type Bounded_String (Capacity : Positive) is record
   Value  : String (1 .. Capacity);
   Length : Natural := 0;
end record
  with Dynamic_Predicate => Length in 0 .. Capacity;
```

Useful Dynamic Predicates (2/2)

- Array type with ordered content

```
type Table is array (Index) of Integer
  with Dynamic_Predicate =>
    (for all K in Table'Range =>
      (K = Table'First or else Table(K-1) <= Table(K)));
```

- Record type with relationship **between** components

```
type Bundle is record
   X, Y : Integer;
   CRC  : Unsigned_32;
end record
  with Dynamic_Predicate => CRC = Math.CRC32 (X, Y);
```

- Scalar type with arbitrary property

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

Restrictions in Usage

- Type with predicate `T` not allowed for some usages
  - As an array index

    ```
    type Table is array (T) of Integer; -- Illegal
    ```
  - As a slice

    ```
    Var := Param(T); -- Illegal
    ```
  - As prefix of attributes `Range`, `First`, and `Last`
    - Because they reflect only range constraints, not predicates
    - Use instead attributes `First_Valid` and `Last_Valid`
    - Not allowed on type with dynamic predicate
- Type with dynamic predicate further restricted
  - Not allowed as range in a *for loop*
  - Not allowed as choice in *case statement* or *case expression*
- Special aspect `Ghost_Predicate` for referring to ghost entities
  - Type cannot be used in membership tests

Dynamic Checking of Predicates

- Dynamic checks inserted by GNAT
  - When using switch `-gnata`
  - Or pragma `Assertion_Policy (Predicate => Check)`
- Placement of checks **similar** as for type constraints
  - On assignment and initialization
  - On conversion `T(...)` and qualification `T'(...)`
  - On parameter passing in a call
- No checks where not needed
  - On uninitialized objects
  - On references to an object
- No checks where that would be too expensive
  - On assigning a part of the object

Static Checking of Predicates

- Static checks performed by GNATprove
  - Always (independent of the choice of switches or pragmas)
- Placement of checks as for dynamic checks
  - Plus assignment on part of the object
  - GNATprove checks objects **always** satisfy their predicate
- No checks only where not needed
  - On uninitialized objects
  - On references to an object
- GNATprove can assume that all initialized objects satisfy their type constraints and predicates

Beware Recursion in Predicates

- Infinite recursion when calling inside the predicate a function taking the type with predicate as parameter type

```
type Nat is new Integer
  with Predicate => Above_Zero (Nat);
function Above_Zero (X : Nat) return Boolean is (X >= 0);
```

```
warning: predicate check includes a call to "Above_Zero"
  that requires a predicate check
warning: this will result in infinite recursion
warning: use an explicit subtype of "Nat" to carry the predicat
high: infinite recursion might occur
```

- Fix by **inlining the property** or introducing a **subtype**

```
type Int is new Integer;
function Above_Zero (X : Int) return Boolean is (X >= 0);
subtype Nat is Int with Predicate => Above_Zero (Nat);
```

Type Invariants

What Is a Type Invariant?

- Boolean property that should always hold of objects of the type
  - ...**outside** of its unit
  - Same use of name of the type and component names as in predicates
- Can only be specified on the completion of a private type (in SPARK)

```
package Bank is
  type Account is private;
  type Currency is delta 0.01 digits 12;
  ...
private
  type Account is ... with
    Type_Invariant => Consistent_Balance (Account);
```

- **Not** part of membership test `X in T`

Dynamic Checking of Type Invariants

- Dynamic checks inserted by GNAT
  - When using switch `-gnata`
  - Or pragma `Assertion_Policy (Type_Invariant => Check)`
- Placement of checks on the creation of values of type `T`
  - Note: that applies to objects with a part of type `T`
  - On default initial value
  - On type conversion `T(...)`
  - On parameter passing after a call to a *boundary subprogram*
    - i.e. call to a subprogram in the public spec of the package
- No checks where not needed
  - On assignment and initialization
  - On qualification `T'(...)`
  - On references to an object
  - On **internal** assignment or call
- No checks where this is impossible for the compiler
  - On **global** variables of type `T`
  - On parts of objects under components of **access** type

Static Checking of Type Invariants

- Static checks performed by GNATprove
  - **Always** where needed (independent of the choice of switches or pragmas)
- Placement of checks as for dynamic checks
  - **Plus** global variables and objects under access types
  - On each call to external subprogram from inside the unit
    - This avoids so-called *reentrancy problems*
  - GNATprove checks objects **always** satisfy their invariant outside of their unit
- No checks only where not needed
- GNATprove can assume that all inputs to *boundary subprograms* and all objects of the type outside the unit satisfy their type invariants
  - Type invariant is used both for proof of unit itself and in other units
  - An expression function deferred to the body can be used to perform an abstraction

Beware Recursion in Type Invariants

- Infinite recursion when calling inside the type invariant a *boundary function* taking the type with invariant as parameter type

```
package Bank is
   type Account is private;
   function Consistent_Balance (A : Account) return Boolean;
private
   type Account is ... with
     Type_Invariant => Consistent_Balance (Account);
```

```
high: cannot call boundary subprogram for type in its own invariant
```

- Fix by declaring the function in the **private** part of the spec

```
private
   type Account is ... with
     Type_Invariant => Consistent_Balance (Account);
   function Consistent_Balance (A : Account) return Boolean
     is (...);
```

Lab

Type Contracts Lab

- Find the `090_type_contracts` sub-directory in `source`
  - You can copy it locally, or work with it in-place
- In that directory, open the project `lab.gpr` in GNAT Studio
  - Or, on the command-line, do `gnatstudio -P lab.gpr`
- Unfold the source code directory (.) in the project pane

Type Predicates

- Find and open the files `basics.ads` and `basics.adb` in GNAT Studio
- Run GNATprove to prove the unit
  - Look at unproved predicate checks, can you explain them?
  - Does it make a difference that `Swap_Pair` is public and `Bump_Pair` is private?
- Fix the predicate check failure in `Bump_Pair`
  - Hint: use an aggregate assignment
- Fix the predicate check failure in `Swap_Pair` by using a base type without predicate for `Pair`

Type Invariants

- Run GNATprove to prove the unit
  - Look at unproved invariant checks, can you explain them?
  - Does it make a difference that `Swap_Triplet` is public and `Bump_Triplet` is private?
- Fix the invariant check failure on the default value for `Triplet`
- Fix the invariant check failure in `Swap_Triplet`
  - Hint: the intent is for the value of all components to rotate
- Fix the invariant check failure in `Bump_And_Swap_Triplet`
  - Hint: look also at `Bump_Triplet`
  - Hint: you will need to add a postcondition to `Bump_Triplet`

All Together

- Run GNATprove to prove the unit and display all proved checks
- Can you explain the presence of predicate checks and invariant checks?
  - How about the absence of checks in `Bump_And_Swap_Pair`?
  - How about the checks in `Bump_And_Swap_Triplet`?

Summary

Type Contracts

- Type contracts given by
  - Type constraints (range and discriminant constraints)
  - Type predicates with aspect `Predicate`
  - Type invariants with aspect `Type_Invariant`
- Type predicates are static or dynamic
  - Special aspects `Static_Predicate` and `Dynamic_Predicate`
  - Slightly different use cases
- Type invariants define an abstraction on private types
  - Always hold on objects outside their unit
  - Can be violated inside the unit

Advanced Proof

Introduction

Proof So Far

- Variables follow data initialization policy
  - Flow analysis deals with initialization
  - Arrays must be initialized by aggregates
  - Variables cannot be partially/conditionally initialized
- Loop-free code
  - Strongest Postcondition calculus does not deal with loops
    - At least, not without a little help
- How do we deal with the following program?

```
procedure Init_Table (T : out Table) is
begin
   for J in T'Range loop
      T(J) := 0;
   end loop;
end Init_Table;
```

Going Beyond Basic Proof

- Relaxed initialization
  - Ability to partially initialize variables
  - Proof deals with initialization of such variables
- Loop pragmas
  - Specialized pragmas to deal with loops in proof
  - Loop invariants provide the necessary help
  - Loop variants deal with loop termination
- SPARK formal containers
  - Dealing with loops over vectors, lists, sets and maps

Relaxed Initialization

Limitations of the Initialization Policy

- Objects must be fully initialized when read
  - Forces useless initialization of unread components
- Arrays must be initialized from an aggregate
  - Otherwise flow analysis cannot check initialization
  - Except in some special cases when a heuristic works
    - e.g. fully initialize an array with a *for loop*
- All outputs must be fully initialized when returning
  - Forces useless initialization of unread outputs

Specifying Relaxed Initialization

- Aspect `Relaxed_Initialization` can be used on objects, types and subprograms

```
type Rec is record ... end record
  with Relaxed_Initialization;
X : Integer with Relaxed_Initialization;
procedure Update (A : in out Arr)
  with Relaxed_Initialization => A;
```

- Corresponding objects (variables, components) have relaxed initialization
  - Flow analysis does not check (full) initialization
  - Instead, proof checks (partial) initialization when read
  - Not applicable to scalar parameter or scalar function result

Specifying Initialized Parts

- Ghost attribute `Initialized` is used to specify initialized objects

```
pragma Assert (R'Initialized);
```

- Or initialization of parts of objects

```
pragma Assert (R.C'Initialized);
```

- Attribute executed like `Valid_Scalars`
  - All scalar subcomponents are dynamically checked to be valid values of their type

Relaxed Initialization and Predicates

- Ghost attribute `Initialized` cannot be used in predicate
  - Rationale: predicate is part of membership tests

* Use instead special `Ghost_Predicate`

- Membership tests are not allowed for such types
- Otherwise subject to same rules as other predicates

```
type Stack is record
   Top     : Index;
   Content : Content_Table;
end record
   with Ghost_Predicate =>
  Content (1 .. Top)'Initialized;
```

Verifying Relaxed Initialization

- Contracts (postcondition, predicate) may refer to `Initialized`

```
procedure Update (R : in out Rec) with
  Post => R'Initialized;
```

- Any read of an object requires its initialization
- Loop invariant may need to state what part of an array is initialized

```
for J in Arr'Range loop
  Arr(J) := ...
  pragma Loop_Invariant
    (Arr(Arr'First .. J)'Initialized;
end loop;
```

Loops

Unrolling Loops

- GNATprove can unroll loops when:
  - Loop is of the form `for J in A .. B loop`
  - Number of iterations is less than 20
  - The only local variables declared in the loop are scalars
- Confirming message issued when using switch `--info`

```
info: unrolling loop
```

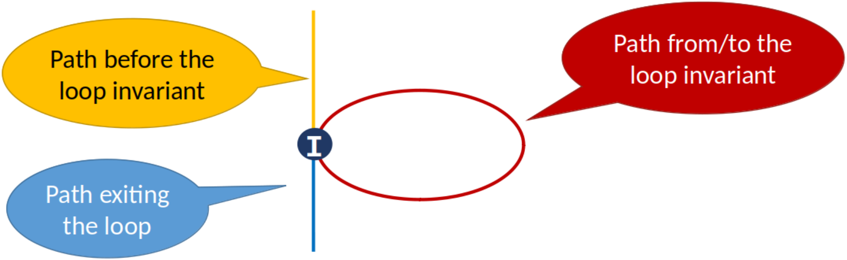
- Strongest Postcondition calculus can deal with unrolled loop
  - But size of code might become large
  - Especially on nested loops
- Loop unrolling can be prevented
  - Globally with switch `--no-loop-unrolling`
  - On a specific loop with a loop invariant

Loop Invariants

- A *loop invariant* is a special assertion
  - Placed inside loops
  - Executed like an assertion at runtime
  - Interpreted specially in proof
  - Slightly different from classical Hoare loop invariant
- Dynamic checks inserted by GNAT
  - When using switch `-gnata`
  - Or pragma `Assertion_Policy (Loop_Invariant => Check)`
- Multiple loop invariants are allowed
  - Must be grouped
  - Same as conjunction of conditions using **`and`**
- Placement anywhere in the top-level sequence of statements
  - Typically at the beginning or end of the loop
  - Can be inside the statements of a *declare block*
  - Default loop invariant of `True` at beginning of the loop

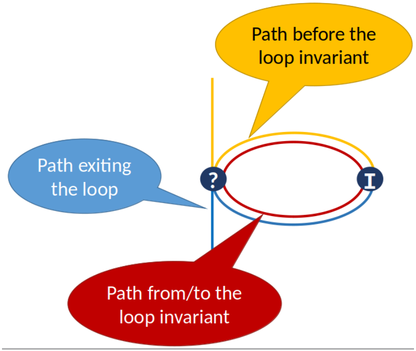
Loop Invariants in Proof

- The loop invariant acts as a cut point for the SP calculus
  - Establish it at the beginning of the loop
  - Check that it is preserved by one iteration
  - Assume it to check the remaining of the program



Placement of Loop Invariants

- Proof reasons around the *virtual loop*
  - Starting from the loop invariant
  - Ending at the loop invariant



Four Properties of a Good Loop Invariant

- These four properties should be established in this order
- [INIT] - It should hold in the first iteration of the loop
  - GNATPROVE generates a loop invariant initialization check
- [INSIDE] - It should allow proving absence of run-time errors and local assertions inside the loop
- [AFTER] - It should allow proving absence of run-time errors, local assertions and the subprogram postcondition after the loop
- [PRESERVE] - It should be preserved by the loop
  - GNATPROVE generates a loop invariant preservation check

Summarizing Mutations

- Analysis of arbitrary loop iteration in coarse context
  - All information on modified variables is lost
  - Except information preserved in the loop invariant
- Example: initialization loop

```
procedure Init_Table (T : out Table)
with
  Post => (for all J in T'Range => T(J) = 0);

procedure Init_Table (T : out Table) is
begin
   for J in T'Range loop
      T(J) := 0;
      pragma Loop_Invariant
        (for all K in T'First .. J => T(K) = 0);
   end loop;
end Init_Table;
```

Accumulating Information

- Analysis of arbitrary loop iteration in coarse context
  - All information accumulated on variables is lost
  - Except information preserved in the loop invariant
- Example: search loop

```
procedure Search_Table (T : Table; Found : out Boolean)
with
  Post => Found = (for some J in T'Range => T(J) = 0);

procedure Search_Table (T : Table; Found : out Boolean) is
begin
   for J in T'Range loop
      if T(J) = 0 then
         return True;
      end if;
      pragma Loop_Invariant
        (for all K in T'First .. J => T(K) /= 0);
   end loop;
   return False;
end Search_Table;
```

Attribute Loop_Entry

- Attribute `Loop_Entry` used to refer to the value of a variable on entry to the loop

```
procedure Bump_Table (T : in out Table) is
begin
   for J in T'Range loop
      T(J) := T(J) + 1;
      pragma Loop_Invariant
        (for all K in T'First .. J => T(K) = T'Loop_Entry(K) + 1);
   end loop;
end Bump_Table;
```

- Similar to attribute `Old` which is usable only inside postconditions
  - In many cases, `X'Loop_Entry` is also value on subprogram entry
  - Same limitations as for attribute `Old`
    - Use `pragma Unevaluated_Use_Of_Old (Allow)` if needed
- Use `X'Loop_Entry(Loop_Name)` for value of `X` on entry to loop not directly enclosing

Loop Frame Condition (1/2)

- Reminder: analysis of arbitrary loop iteration in coarse context
  - All information on modified variables is lost
  - Except information preserved in the loop invariant
- This is true for the *loop frame condition*
  - Variables that are not modified
  - Parts of modified variables that are preserved
  - Similar to frame condition on subprogram calls
- GNATPROVE generates part of the frame condition
  - Variables that are not modified, or only on paths that exit the loop
  - Components of records that are not modified
  - Components of arrays that are not modified
    - When the array is only assigned at the current loop index

Loop Frame Condition (2/2)

- In other cases, explicit frame condition might be needed
- Typically use attribute `Loop_Entry`

```
procedure Bump_Table (T : in out Table) is
begin
   for J in T'Range loop
      T(J) := T(J) + 1;
      pragma Loop_Invariant
        (for all K in J .. T'Last =>
           (if K > J then T(K) = T'Loop_Entry(K)));
   end loop;
end Bump_Table;
```

Classical Loop Invariants

- Known best loop invariants for some loops
  - Initialization loops - initialize the collection
  - Mapping loops - map each component of the collection
  - Validation loops - check each component of the collection
  - Counting loops - count components with a property
  - Search loops - search component with a property
  - Maximize loops - search component that maximizes a property
  - Update loops - update each component of the collection
- SPARK User's Guide gives detailed loop invariants
  - See section *7.9.2 Loop Examples*
  - Loops on arrays or formal containers

Quiz: Non-terminating Loops

What's wrong with the following code?

```
loop
   null;
end loop;
pragma Assert (False);
```

Quiz: Non-terminating Loops

What's wrong with the following code?

```
loop
   null;
end loop;
pragma Assert (False);
```

- Loop does not terminate
- GNATPROVE proves the assertion of `False`!
  - Because that program point is unreachable (dead code)
- GNATPROVE implements defense in depth
  - Non-terminating loop causes enclosing subprogram to also not terminate
  - Switch `--proof-warnings=on` can detect dead code
  - Proof of loop termination based on loop variants

Loop Variants (1/2)

- A *loop variant* is a special assertion
  - Placed inside loops
  - Executed specially at runtime
  - Interpreted specially in proof
- Dynamic checks inserted by GNAT
  - When using switch `-gnata`
  - Or pragma `Assertion_Policy (Loop_Variant => Check)`
  - Check that expression varies as indicated at each iteration
- Only one loop variant is needed to prove loop termination
  - And only on *while loop* or *plain loop*, not on *for loop*
- Same placement as for loop invariants
  - Must be grouped if both presents

Loop Variants (2/2)

- Same syntax as subprogram variants

```
procedure Bump_Table (T : in out Table) is
   J : Index'Base := T'First;
begin
   while J <= T'Last loop
      T(J) := T(J) + 1;
      J := J + 1;
      pragma Loop_Variant (Increases => J);
   end loop;
end Bump_Table;
```

- Could also use `(Decreases => -J)`
- Same loop variant could be placed anywhere in the loop here
  - Because check between two successive evaluations of the variant
  - The loop invariant must be modified to reflect current values

Formal Containers

Formal Containers in SPARKlib

- Available from SPARK Library
  - Distributed with SPARK Pro
  - Copy `sparklib.gpr` or `sparklib_light.gpr` locally
  - Set value of `Object_Dir` in the copied project file
  - To use, add `with "sparklib[_light]";` in your project file
- Reminder: four kinds of formal containers
  - vectors
  - doubly linked lists
  - sets (hashed and ordered)
  - maps (hashed and ordered)
- All available in bounded and unbounded versions
- All generics that need to be instantiated
  - Only their spec is in SPARK
  - Their implementation is not proved

Bounded Formal Containers

- Bounded version for light and embedded runtimes
- Under `SPARK.Containers.Formal.<name>`
- Use discriminated record
  - Discriminant `Capacity` fixes maximum size
- Component type must have known size (*definite* type)
- Container type itself is definite
  - Bounded container can be component of another formal container

Unbounded Formal Containers

- Unbounded version for full runtimes
- Under `SPARK.Containers.Formal.Unbounded_<name>`
- Use dynamic memory allocation
  - For each component in the container
  - For growing the container
- Use controlled types for dynamic memory reclamation
- Component type may have unknown size (*indefinite* type)
- Container type itself is definite
  - Unbounded container can be component of another formal container

Loops Over Formal Containers

- Same as for quantified expressions
- Range-based iteration (only for vectors)

```
for J in V.First_Index .. V.Last_Index loop
   V.Replace_Element (J, 0);
end loop;
```

- Iteration over positions

```
for J in V loop
   V.Replace_Element (J, 0);
end loop;
```

- Iteration over components (no update!)

```
for E of V loop
   pragma Assert (E = 0);
end loop;
```

Loop Invariants Over Formal Containers

- Range-based iteration (only for vectors)
  - Use scalar index `J` to access vector at `V.Element (J)`
- Iteration over positions
  - For vectors, same as range-based iteration (cursor is index)
  - Otherwise, need to reason about formal model
    - Functional model of the container
    - Mapping from cursors to positions
    - Sequence of components/keys of the container
- Iteration over components
  - Impossible to access previous components
  - Use iteration over positions instead

Formal Model of Formal Containers

- Defined in local package `Formal_Model`
  - Based on functional containers (also part of SPARKlib)
    - Immutable containers to represent mathematical one
  - Used in contracts of formal containers API
- Functional model of the container
  - Given by function `Model`
  - Returns a different type
    - A sequence of components for formal lists
    - A set of components for formal sets
    - A map from keys to components for maps
- Mapping from cursors to positions
  - Given by function `Positions`
  - Positions in the iteration sequence
- Sequence of components/keys of the container
  - Corresponds to the iteration sequence
  - Given by different functions
    - `Model` for lists
    - `Elements` for sets
    - `Keys` for maps

Difficulties with Loops Over Formal Containers

- GNATprove does not unroll such loops
- GNATprove does not generate a frame condition
  - Contrary to loops over arrays
  - Need to explicitly state the frame condition using attribute `Loop_Entry`
- Container structure may be modified in the loop
  - When inserting or deleting components
  - In general, need to know position of corresponding cursor
    - Relative to current cursor: e.g. previous/next cursor
    - Otherwise difficult with hashed sets/maps

Functional Containers

- Available from SPARK Library
- Five kinds of functional containers
  - infinite sequences
  - vectors
  - sets
  - multisets
  - maps
- Simple containers close to mathematical structures
  - No bounds on cardinality
  - No cursors for iteration
  - No order of components in sets and maps
  - Functional: cannot modify them, rather create a new one
- They are easy to handle for proof
  - Often used as models for more complex structures
- They are executable but might be inefficient

Lab

Advanced Proof Lab

- Find the `010_advanced_proof` sub-directory in `source`
  - You can copy it locally, or work with it in-place
- Copy locally **sparklib.gpr** from your SPARK install and set `Object_Dir`
- In that directory, open the project `lab.gpr` in GNAT STUDIO
  - Or, on the command-line, do `gnatstudio -P lab.gpr`
- Unfold the source code directory (.) in the project pane

Array Initialization Loop

- Find and open the files `loop_init.ads` and `loop_init.adb` in GNAT Studio
- Run GNATprove to prove the subprogram `Init_Table`
  - Can you explain why `Init_Table` is proved?
  - Confirm this by rerunning GNATprove with switch `--info`
- Change the type `Table` to be an unconstrained array:

```
type Table is array (Index range <>) of Integer;
```

- Run GNATprove to prove the subprogram `Init_Table`
  - Can you explain why the postcondition is not proved?
  - Confirm this by rerunning GNATprove with switch `--info`
- Add a loop invariant in `Init_Table`.
  - Hint: take inspiration in the postcondition.
  - Subprogram `Init_Table` should be proved except for initialization checks.
- Mark parameter `T` as having relaxed initialization.
  - Rerun GNATprove.
  - Add the necessary loop invariant to complete the proof of `Init_Table`.

Array Mapping Loop

- Run GNATPROVE to prove the subprogram `Bump_Table`
- Add a loop invariant in `Bump_Table`.
  - Hint: use attribute `Loop_Entry`
  - Can you prove the subprogram without a loop frame condition?
- Change the assignment inside the loop into
  `T(J + 0) := T (J) + 1;`
  - Can you still prove the subprogram without a loop frame condition?
  - Discuss this with the course instructor.
  - Complete the loop invariant with a frame condition to prove `Bump_Table`

Formal Container Loops

- Run GNATprove to prove the subprogram `Init_Vector`
- Add a loop invariant in `Init_Vector`
  - Hint: you need to state that `V.Last_Index` is preserved
- Run GNATprove to prove the subprogram `Init_List`
- Add a loop invariant in `Init_List`
  - Hint: the position of cursor `Cu` in `L` is `Positions (L).Get (Cu)`
  - Hint: the sequence of components for `L` is `Model (L)`

Summary

Advanced Proof

- Use relaxed initialization when needed
  - Some variables are partially initialized
  - Some array variables are initialized in a loop
  - More annotations are needed with ghost attribute `Initialized`
- Proof of loops requires more work
  - Add loop invariants to prove correction
  - Take special care of the loop frame condition
  - Add loop variants to prove termination
- Formal containers
  - Generics for vectors, lists, sets and maps
  - Available in all runtime libraries
  - Proof of code using formal containers uses formal models

Advanced Flow Analysis

Introduction

Data and Information Flow Analysis

- Data flow analysis
  - Models the variables used by a subprogram
  - Enforces data initialization policy
  - Detects reads of uninitialized data
- Data dependencies can be specified
  - Introduced by aspect `Global`
- Information flow analysis
  - Models the flow of information from inputs to outputs
  - Can be very useful for security analysis
- Flow dependencies can be specified
  - Introduced by aspect `Depends`

Information Flow Analysis

Direct and Indirect Flows

- A direct flow occurs when assigning `A` to `B`

```
B := A;
```

- An indirect flow occurs when assigning `B` conditioned on `A`

```
if A then
   B := ...
end if;
```

- A direct flow can be masquerading as indirect flow

```
if A then
   B := True;
else
   B := False;
end if;
```

- GNATPROVE handle both flows together in flow analysis

Self-Dependency on Array Assignment

- Flow analysis is not value-dependent
- Assigning an array component or slice preserves part of the original value

```
type T is array (1 .. 2) of Boolean;
A : T := ...

A (1) := True;
-- intermediate value of A seen as dependent on
-- original value
A (2) := False;
-- final value of A seen as dependent on original value
```

- This holds also for slices

```
A (1 .. 2) := (True, False);
-- final value of A seen as dependent on original value
```

Flow Dependency Contracts

Basic Data Dependency Contracts

- Introduced by aspect `Depends`
- Optional, but must be complete if specified
- Describes how outputs depend on inputs

```
procedure Proc
with
  Depends => (X => (X, Y),
              Z => V);
```

- Not very interesting for functions which have only their result as output

```
function Func (X : Integer)
with
  Depends => (Func'Result => (X, Y, Z));
```

Some Outputs May Appear As Inputs

- Parts of outputs are in fact inputs:
  - Bounds of arrays
  - Discriminants of records
  - Tags of tagged records
- These output objects will appear as inputs in `Depends` when bounds/discriminants/tags not implied by the object subtype

```
procedure Proc (Tab : out Table)
with
  Global => (Output => Glob),
  Depends => (Tab  => Tab,
              Glob => Glob);
```

Special Cases

- Some outputs may depend on no input
  - Typically when initializing data to some constant value
  - Thus, output depends on *null*

```
procedure Init (T : out Table)
with
  Depends => (T => null);
```

- Some inputs may not flow into any output
  - Typically when effect hidden from analysis
  - Or input used only for debug
  - Also the case for global variables of mode `Proof_In`
  - Must be last line of flow dependencies

```
procedure Debug (T : Table)
with
  Depends => (null => T);
```

Special Notation

- Outputs can also be grouped

```
procedure Init (T1, T2 : out Table)
with
  Depends => ((T1, T2) => null);
```

- Symbol + indicates a self-dependency

```
procedure Update (T : in out Table)
with
  Depends => (T => +null);  -- same as (T => T)
```

- Most useful with grouped outputs

```
procedure Update (T1, T2 : in out Table)
with
  Depends => ((T1, T2) => +null);
             -- same as (T1 => T1, T2 => T2)
```

Automatic Generation

From Data Dependencies

- Data dependencies may be specified or generated
- If flow dependencies are not specified, they are generated
  - All outputs depend on all inputs
  - All globals of mode `Proof_In` have no effect on outputs
- This is a correct over-approximation of actual flow dependencies
  - This might be too imprecise for analysis of callers
  - In that case, add explicit flow dependencies

From Flow Dependencies

- If only flow dependencies are specified
- Data dependencies are generated
  - All variables only on the left-hand side are outputs
  - All variables only on the right-hand side are inputs
  - All other variables are both inputs and outputs
- This is the exact data dependencies consistent with flow dependencies
  - Except some globals of mode `Proof_In` may be classified as inputs

Lab

Advanced Flow Analysis Lab

- Find the `110_advanced_flow_analysis` sub-directory in `source`
  - You can copy it locally, or work with it in-place
- In that directory, open the project `lab.gpr` in GNAT Studio
  - Or, on the command-line, do `gnatstudio -P lab.gpr`
- Unfold the source code directory (.) in the project pane

Flow Dependencies

- Find and open the files `basics.ads` and `basics.adb` in GNAT Studio
- Run GNATprove in flow analysis mode
- Add flow dependency contracts to all subprograms except `Strange_Init_Rec` and `Strange_Init_Table`
    - Rerun GNATprove in flow analysis mode
    - Discuss the correct flow dependencies of `Init_Table` with the instructor.

Imprecise Flow Dependencies

- Copy the flow dependencies of `Init_Rec` and `Init_Table` for respectively `Strange_Init_Rec` and `Strange_Init_Table`
- Run GNATprove in flow analysis mode
  - Understand the error messages and add the suggested dependencies.
- Run GNATprove in flow analysis mode
  - Do you understand the reason for the check messages?
  - Either adapt the flow dependencies or justify the messages with pragma `Annotate`

Summary

Advanced Flow Analysis

- Flow dependencies can be specified
  - This can be important for security
- Flow analysis detects:
  - Violation of flow dependency contracts (`Depends`)
  - Inconsistency between data and flow dependency contracts
- Flow analysis is imprecise
  - On value-dependent flows
  - On array assignment to index/slice

Pointer Programs

Introduction

Absence of Interferences

- Flow analysis rejects aliasing
  - Between two parameters
  - Between a parameter and a global variable
  - ... when that may lead to interferences
- Interferences when one of the variables is written
- Many features avoid direct use of pointers
  - Array types
  - By-reference parameter passing mode
  - Address specifications `X : Integer with Address => ...`
  - Generics (avoid C-style `void*` genericity)
- What about pointers?

Pointers and Aliasing

- Pointers introduce aliasing
  - This violates SPARK principle of absence of interferences
- Rust programming language popularized *ownership*
  - Only one pointer (the *owner*) at any time has read-write access
  - Assigning a pointer transfers its ownership
- Work on ownership in SPARK started in 2017
  - First version released in SPARK Pro 20
  - Detection of memory leaks in SPARK Pro 21
  - Support for all access types in SPARK Pro 22
  - SPARK libraries for aliasing in SPARK Pro 23

Ownership Checking

Access Types in Ada

- Access-to-variable vs access-to-constant types

```
AV : access Integer;
AC : access constant Integer;
```

  - AV can be used to modify the integer, AC cannot

- Named vs anonymous access types

```
type Acc is access Integer;
AN : Acc;
AA : access Integer;
```

  - Convenience in Ada to save the introduction of a type name

- Pool-specific vs general access types

```
type PS_Acc is access Integer;
type G_Acc is access all Integer;
```

  - Type PS_Acc can only point to the heap, GS_Acc can point to the heap and stack.

- Accessibility levels prevent escaping pointers to the stack
- Not null access types forbid use of value **null**

Access Types in SPARK

- Named pool-specific access-to-variable types: subject to ownership

```
type PS_Int_Acc is access Integer;
```

- Named access-to-constant types: aliasing allowed, deallocation forbidden

```
type Cst_Int_Acc is access constant Integer;
```

- Named general access-to-variable types: subject to ownership, deallocation forbidden

```
type Gen_Int_Acc is access all Integer;
```

- Anonymous access-to-object types: for borrowing and observing

```
X : access Cell := ...
X : access constant Cell := ...
```

Memory Ownership Policy

- A chunk of memory has a single *owner*
- Assigning a pointer *moves* its ownership
- Only the owner can both read and write the memory

```
X := new Integer'(1);
--  X has the ownership of the cell
Y := X;
--  The ownership is moved to Y
Y.all := Y.all + 1;
--  Y can access and modify the data
pragma Assert (X.all = 1);
--  Error: X can no longer access the data
```

- Ownership policy ensures absence of interferences

Model of Pointers in SPARK

- Pointers are seen as records in analysis
  - Both for flow analysis and proof
  - This is possible thanks to absence of interferences

```
type Int_Acc is access Integer;
X : Int_Acc := new Integer'(42);
```

is treated like:

```
type Int_Acc (Nul : Boolean := False) is record
  case Nul is
    when True  => null;
    when False => Content : Integer;
  end case;
end record;
X : Int_Acc := Int_Acc'(Nul => False, Content => 42);
```

- Value of pointer itself is not modelled
  - This is an intentional limitation to
    - Allow allocators in expressions
    - Allow dellocation in functions
  - Equality of pointers is not supported (only with `null`)

Borrowing and Observing

- Borrowing is temporary read-write access
  - either through a declaration

    ```
    X : access Cell := Current_Cell.Next;
    ```
  - or through a call (access type can be named or anonymous)

    ```
    procedure Update_Cell (X : access Cell);
    Update_Cell (Current_Cell.Next);
    ```
- In-out parameter of access type is *moved* on entry and return
- Observing is temporary read-only access
  - either through a declaration

    ```
    X : access constant Cell := Current_Cell.Next;
    ```
  - or through a call

    ```
    procedure Read_Cell (X : access constant Cell);
    Read_Cell (Current_Cell.Next);
    ```

Access to Constant Data

- Data is constant all the way down
  - Data designated by the pointer is constant
  - Pointers in that data inherit the same property
  - This is specific to SPARK: in Ada only designated data is constant
- Also applies to constants and input parameters of composite types containing pointers
  - Different from constants and input parameters of access-to-variable type
- Aliasing is allowed

Access to Data on the Stack

- Use attribute `Access` on local variable
  - Not allowed on global variable which would remain visible
  - Result of general access type with `access all` syntax
- `Constant'Access` of access-to-constant type
- `Variable'Access` of access-to-variable type
- Variable is *moved* and cannot be referenced anymore

Attributes `Old` and `Loop_Entry`

- Attributes `Old` and `Loop_Entry` not applicable to pointers
  - Implicit copy on subprogram/loop entry would violate ownership
- Prefix of access type needs to be a call to an *allocating function*
  - Allocating function is a function returing an access-to-variable type

```
function Copy (X : Ptr) return Ptr
  with Post => Copy'Result.all = X.all;

procedure P (X : in out Ptr)
  with Post => Property (Copy (X)'Old);
```

Useful Tips

- No cycles or sharing inside mutable data structures
- Global objects can also be moved temporarily
  - Procedure must restore some value (or null) before returning
- Allocation function returns a new object of access-to-variable type
  - Similar to initialized allocator with `new T'(Value)`
  - Some special *traversal functions* give access to part of an object
- Deallocation procedure simply nullifies in-out access parameter

Loops and Predicted Values

Recursive Data Structures

- Pointers allow to build recursive data structures like lists

```
type List_Cell;
type List_Acc is access List_Cell;
type List_Cell is record
   Value : Integer;
   Next  : List_Acc;
end record;
```

- Traversing the data structure can use
  - Recursion, typically for specification functions
  - Loops otherwise

Pointers and Recursion

- No built-in quantified expression for recursive data structures
- Instead, use recursion to traverse the structure

```
function All_List_Zero
  (L : access constant List_Cell) return Boolean
is (L = null or else
     (L.Value = 0 and then All_List_Zero (L.Next)));
```

- Reminder: GNATPROVE protects against non-terminating recursive functions
  - No axioms generated for such functions
  - Need to prove termination of recursive functions
- Use special form of structural subprogram variant

```
function All_List_Zero ... with
  Subprogram_Variant => (Structural => L);
```

Pointers and Loops

- Procedure `Init_List_Zero` initializes `L`

```
procedure Init_List_Zero (L : access List_Cell)
  with Post => All_List_Zero (L);
```

- Initialization uses loop to traverse data structure

```
procedure Init_List_Zero (L : access List_Cell) is
   B : access List_Cell := L;
begin
   while B /= null loop
      B.Value := 0;
      B := B.Next;
   end loop;
end Init_List_Zero;
```

- Problem: how do we express that previous cells have value zero?
    - Cannot refer to value of `L` while borrowed

Predicted Values

- Special annotation `At_End_Borrow` on identity function
  - For proof, refers to value of argument at the end of the borrow
  - For execution, is simply the identity function

```
function At_End
  (L : access constant List_Cell)
  return access constant List_Cell
is (L)
with
  Ghost,
  Annotate => (GNATprove, At_End_Borrow);
```

- Loop invariant can refer to values at end of the borrow
  - Value of borrower at end of the borrow `At_End (B)`
  - Value of borrowed at end of the borrow `At_End (L)`

```
pragma Loop_Invariant
  (if All_List_Zero (At_End (B))
   then All_List_Zero (At_End (L)));
```

- Invariant proved using what is known now about the value at end
  - There is no look ahead
  - Loop invariant proved because values in L and not B are frozen to 0

SPARK Libraries

Pointers with Aliasing (1/2)

- SPARK Library defines two generics
  - `SPARK.Pointers.Pointers_With_Aliasing`
  - `SPARK.Pointers.Pointers_With_Aliasing_Separate_Memory`
  - Only generic parameter is any type `Object`
- Both allow aliasing pointers
  - Type `Pointer` is private
    - User code can copy such pointers freely
    - Ownership policy does not apply
  - All accesses through API check validity of pointer

Pointers with Aliasing (2/2)

- Shared API to create, free, access pointers

```
procedure Create (O : Object; P : out Pointer);
function Deref (P : Pointer) return Object;
procedure Assign (P : Pointer; O : Object);
procedure Dealloc (P : in out Pointer);
```

- Version in `Pointers_With_Aliasing_Separate_Memory` adds parameter

```
Memory : in out Memory_Type
```

  - To handle separate groups of pointers in different memories

- Use of pointers with aliasing is *possible* but *costly*
  - Need to maintain validity of pointers at all times
  - Need to maintain separation of pointers at all times
  - This comes for free with the ownership policy

Access-to-subprogram Values

Contracts on Access-to-subprogram Types

- Access-to-subprogram values not subject to ownership
- Only preconditions and postconditions are allowed

```
type Proc is access procedure (...)
with
  Pre  => ...
  Post => ...
```

- Very often using `not null access` (for parameters)
- Implicit **Global => null** on type
- GNATprove checks feasibility of contract
- Creating a value of access-to-subprogram type with attribute `Access`

```
procedure P (...);
Acc : Proc := P'Access;
```

- GNATprove checks conditions for refinement
  - Pre of type implies pre of subprogram
  - Post of subprogram implies post of type

Higher Order Specialization

- Higher order functions take an anonymous access-to-subprogram parameter
- Example of map:

```
function Map
  (A : Nat_Array;
   F : not null access function (N : Natural) return Natural)
   return Nat_Array;
```

- Function `F` above cannot read global variables
- Annotation `Higher_Order_Specialization` allowed on **Map**
  - Call to `Map (A, Func'Access)` specialized for `Func`
  - `Func` is allowed to read global variables
  - `Func` can have a precondition and postcondition
- Used in SPARK Higher Order Library
  - Associated lemmmas also use annotation `Higher_Order_Specialization`
  - Lemmas specialized when calls are specialized

Interrupt Handlers

- Handler can be called asynchronously outside SPARK program
  - But not called from SPARK code
- Handler declared with access-to-subprogram type
- Handler may read or write global data
- Annotation `Handler` on access-to-subprogram type

```
type No_Param_Proc is access procedure with
  Annotate => (GNATprove, Handler);
```

- Can take `Access` on subprogram that reads or writes global

```
procedure Reset with Global => ...
P : No_Param_Proc := Reset'Access;
```

Lab

Pointer Programs Lab

- Find the `120_pointer_programs` sub-directory in `source`
  - You can copy it locally, or work with it in-place
- Copy locally **sparklib.gpr** from your SPARK install and set `Object_Dir`
- In that directory, open the project `lab.gpr` in GNAT Studio
  - Or, on the command-line, do `gnatstudio -P lab.gpr`
- Unfold the source code directory (.) in the project pane

Swapping Pointers

- Find and open the files `pointers.ads` and `pointers.adb` in GNAT Studio
- Run GNATprove in flow analysis mode
- Fix the ownership error in `Swap_Ptr`
- Add postconditions to procedures `Swap` and `Swap_Ptr`
  - Hint: you cannot compare pointers in SPARK
  - Rerun GNATprove to prove these procedures

Allocation and Deallocation

- Run GNATprove to prove procedure `Realloc`
  - Understand the memory leak message and fix it.
  - Hint: you need to add a postcondition to `Dealloc`
- Understand what makes `Alloc` and `Dealloc` special
  - Discuss with the course instructor.

Recursion and Loops

- Review the rest of the code manipulating types `List_Cell` and `List_Acc`
  - Discuss with the course instructor.
- Run GNATPROVE to prove the complete unit.
- Add a loop invariant in procedure `Init_List_Zero`
  - The postcondition of `Init_List_Zero` should be proved
- Add a loop variant in procedure `Init_List_Zero`
  - First using the structural loop variant
  - Next using a numerical loop variant, by defining a recusrive function `Length`

```
function Length
  (L : access constant List_Cell) return Big_Natural;
```

Summary

Pointer Programs

- Pointers are supported in SPARK
  - All kinds of pointers are supported
  - Access-to-constant is all the way down
  - General access cannot be deallocated
- Ownership policy is key
  - Ensures absence of interferences
  - Constrains code and data structures
    - No cyclic data structures
- Loops require special reasoning
  - So-called promises peek at value after borrow
  - Useful in loop invariants

Auto-Active Proof

Introduction

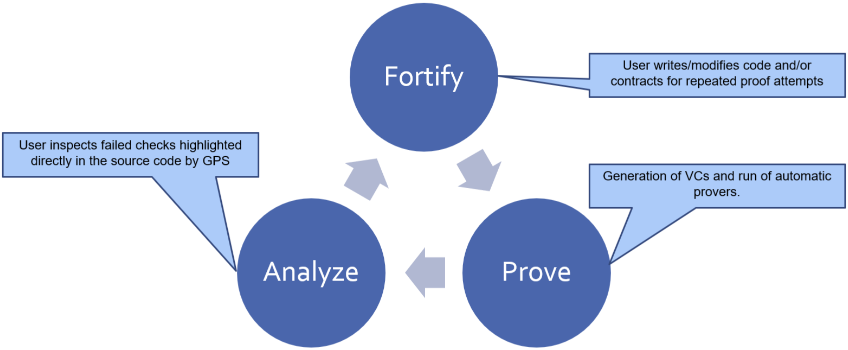
Not All Proofs Are Easy

- correct spec + correct code → proof?
- We saw already limitations of automatic provers:
  - Arithmetic - non-linear and mixed arithmetic
  - Quantifiers - existential quantifiers and induction
  - Proof context - may become too large
- *Auto-active proof* overcomes these limitations
  - Based on **automatic** provers
  - Using human **interaction**
- Akin to *developing the proof* like we develop code
  - Still much lower effort than required in proof assistants (Coq, Lean, Isabelle...)
  - Special code supporting the proof is called *ghost code*

Investigating Unproved Checks

- Maybe spec is incorrect? Maybe code is incorrect? Or both?
- Need to investigate unproved checks
  - Easiest way is to get runtime failure in spec or code
    - Test the code+spec with assertions enabled!
    - Then debug with the usual debugging tools
  - Increase the proof effort
    - More provers and time to attempt proof
  - Break down property to prove into easier ones
    - Add intermediate assertions
    - Extract proof of a property in a lemma
- Need to understand the messages output by GNATPROVE!
  - Tool tries to help you help it

The Proof Cycle



GNATprove Messages

Parts of a Check Message

- Messages adapted to usage with switch `--output=`
  - Message in colors with code excerpts in terminal
  - Message on one line in IDEs (further separated by IDE)
- Typical check message consists in multiple parts

```
file:line:col: severity: check "might fail"
  "cannot prove" this-part
  "e.g. when" counterexample
  "reason for check:" check-is-here-for-that-reason
  "possible fix:" this-or-that-could-fix-it
  continuation-message-with-another-source-location
```

Check Message Example

What is the problem with this code?

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

Check Message Example

What is the problem with this code?

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

```
incr.adb:3:11: high: overflow check might fail
  cannot prove upper bound for 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
```

Counterexamples

- A *counterexample* is input values that lead to check failure
- Different displays in a terminal and in IDEs
  - In GNAT Studio, GNATprove displays the full path
    - Magnify icon next to check message to display path
    - Values of variables displayed along the path
  - In terminal and other IDEs, GNATprove displays final values
    - Values of variables in the check expression
    - At the point where the check is failing
- Feature is activated with switch `--counterexamples=on`
  - Off by default at proof levels 0, 1
  - On by default at proof levels 2, 3, 4
- Automatic prover cvc5 is asked for a counterexample on unproved checks
  - Counterexample is re-checked twice by GNATprove
    - Once by simulating the execution interprocedurally
    - Once by simulating the execution intraprocedurally
  - Result of simulations allows to refine message
    - `high` message when execution is known to fail
    - message points at missing contracts otherwise

Possible Fix

- Suggestion of a possible way to fix the problem
  - This might not be the right way!
  - Based on heuristics and most likely reasons
- In general, suggest missing precondition or loop invariant
  - Because some variable in check is not constrained at all

```
possible fix: precondition of subprogram should mention Var
possible fix: precondition of subprogram should mention Var'Initialized
possible fix: add precondition (Expr in Integer) to subprogram
possible fix: loop should mention Var in a loop invariant
```

- Also suggests missing postcondition

```
possible fix: call should mention Var in a postcondition
possible fix: you should consider adding a postcondition to function
  or turning it into an expression function in its unit spec
```

- Other suggestions for arithmetic and representation

```
possible fix: use pragma Overflow_Mode or switch -gnato13
  or unit SPARK.Big_Integers
possible fix: overlaying object should have an Alignment
  representation clause
```

Continuation Messages

- Typically points to another relevant source location
- Specific instantiation for code in generics

```
in instantiation at...
```

- Specific call for code in inlined subprogram

```
in call inlined at...
```

- Specific contract when inherited

```
for inherited predicate at...
for inherited default initial condition at...
in inherited contract at...
```

- Original contract when inlined

```
in inlined expression function body at...
in inlined predicate at...
in default value at...
```

Information Messages

- Information messages about proved or justified checks
  - With switch `--report=all/provers/statistics`
  - Checks justified with pragma `Annotate`

```
file:line:col: check proved
file:line:col: check justified
```

- Information about analysis
  - With switch `--info`
  - Subprograms that are inlined or not
  - Loops that are unrolled or not
  - Function contracts not available for proof (termination)
  - Imprecise value for some attributes and functions

Increasing the Proof Effort

Control of the Proof Effort

- Automatic provers have different strengths
  - More provers = more likely to prove checks
  - From one prover to four (Alt-Ergo, COLIBRI, cvc5, Z3)
  - Use switch `--provers` e.g. `--provers=all`
- Automatic provers heuristically search for a proof
  - More time = more likely to prove checks
  - Time given in seconds (`--timeout`) or prover-specific steps (`--steps`)
- Default proof effort is minimal (one prover, 100 steps)
- Timeout vs steps
  - Timeout is best to bound the running time
  - Steps are useful for reproducible results across machines
    - Still use timeout to avoid runaway proofs

Proof Levels

- Switch `--level` bundles lower-level switches
  - `--level=0` uses 1 prover and 1sec timeout
  - `--level=1` uses 3 provers and 1sec timeout
  - `--level=2` uses 3 provers and 5sec timeout
  - `--level=3` uses 3 provers and 20sec timeout
  - `--level=4` uses 3 provers and 60sec timeout
- Level 2 is the recommended one to start
  - Activation of counterexamples also starts at level 2
- Levels do not use steps (`--steps=0`) and increase memory limit (`--memlimit`)
- Specific values for lower-level switches take precedence
  - e.g. `--level=2 --timeout=120 --steps=10000`

Running Proof Faster

- During development, run GNATprove on relevant part
  - On given file
    - With SPARK → Prove File in GNAT Studio
    - With task Prove file in Visual Studio Code
    - With `-u file` in terminal
  - On given subprogram, selected region of code, selected line of code
    - With corresponding menus in IDEs and switches in terminal
- Use parallelism with `-j` e.g. `-j0` for all cores
  - Proof faster on more powerful machines: more cores, more memory, faster clock
- Sharing session files by setting attribute `Proof_Dir` in project file
  - This also allows to simply replay proofs with `--replay`
- Sharing proof results via a cache
  - Can store database in a file, or connect to a Memcached server

Ghost Code

Intermediate Assertions

- Intermediate assertions can help provers

```
pragma Assert (Intermediate_Assertion_1);
pragma Assert (Intermediate_Assertion_2);
pragma Assert (Complex_Assertion);
```

- In addition, each assertion can be proven by different prover
- Intermediate assertions help prove each path separately

```
if Cond then
   pragma Assert (Assertion_1);
   return;
end if;

if Other_Cond then
   pragma Assert (Assertion_2);
else
   pragma Assert (Assertion_3);
end if;
```

- Intermediate assertions are essential to investigate unproved checks

Ghost Code

- *Ghost code* is code meant only for verification
  - Intermediate assertions can refer to ghost entities
  - Contracts can also refer to ghost entities
- Special aspect `Ghost` used to identify ghost entities
  - Ghost functions express properties used in contracts

    ```
    function Is_Valid (X : T) return Boolean is (...)
      with Ghost;
    procedure Proc (X : T) with Pre => Is_Valid (X);
    ```
  - Ghost variables hold intermediate values referred to in assertions

    ```
    X_Saved : constant T := X with Ghost;
    ...
    pragma Assert (X = 3 * X_Saved);
    ```
  - But also ghost types, procedures, packages
- Ghost statements are:
  - Calls to ghost procedures
  - Assignments to ghost variables

Compilation of Ghost Code

- Ghost code compiled by GNAT
  - When using switch `-gnata`
  - Or pragma `Assertion_Policy (Ghost => Check)`
- GNATprove checks that ghost code has no effect

```
X_Saved : constant T := X with Ghost;
...
X_Saved := X; -- ghost assignment
X := X_Saved; -- error
```

- Same behavior with or without ghost code
  - Proof using ghost code
  - Even if execution without ghost code

Ghost Functions

- Most common ghost entities
- Ghost functions express properties used in contracts
  - Typically as expression functions
  - Complete the existing API with queries only for verification
- Ghost functions can be very costly in running time
  - If objective is not to execute them!
  - Typically when creating models of the actual types
  - e.g. using SPARK functional containers (sets, maps, etc)
  - e.g. like it is done for SPARK formal containers

Ghost Variables

- Local ghost variable or constant
  - Typically to store intermediate values
    - e.g. value of variable at subprogram entry
  - Also used to build useful data structure supporting proof

```
procedure Sort (T : in out Table)
  with Post => Is_Permutation (T, T'Old)
is
  Permutation : Index_Array := (for J in T'Range => J)
    with Ghost;
begin
```

- Global ghost variable
  - Help specify and verify interprocedural properties
  - Maintain a model of a complex or private data structure
  - Specify properties over sequence of calls

Ghost Procedures

- Inlined local ghost procedure without contract
  - Used to group operations on ghost variables
  - Guarantees removal of all the code (e.g. loops, conditionals)
- Ghost procedure with contract and no effects
  - Also called *lemma*
  - Isolates the proof that the precondition implies the postcondition
  - Proof of lemma might be full automatic

```
procedure Lemma (X : T)
with
  Pre  => ...,
  Post => ...;
procedure Lemma (X : T) is null;
```

  - Lemma is used by calling it on relevant arguments

```
pragma Assert (precondition-of-lemma);
Lemma (Y);
-- postcondition of lemma known here
```

SPARK Lemma Library

- Part of SPARK Library in `SPARK.Lemmas.<unit>`
- Mostly non-linear arithmetic lemmas
  - Generics instantiated for standard numerical types
  - On signed and modular integer arithmetic

```
procedure Lemma_Div_Is_Monotonic
  (Val1  : Int;
   Val2  : Int;
   Denom : Pos)
with
  Global => null,
  Pre  => Val1 <= Val2,
  Post => Val1 / Denom <= Val2 / Denom;
```

  - On fixed-point arithmetic (specific to GNAT)
  - On floating-point arithmetic
    - Monotonicity of operations, conversions with integer, rounding

SPARK Higher Order Library

- Higher order functions and lemmas to express:
  - mapping a function over a collection
  - folding a computation over a collection
  - summing a quantity over a collection
  - counting matches over a collection
- Over arrays in `SPARK.Higher_Order(.Fold)`
  - Fold, sum and count over arrays and matrices
  - Defined as generics to be instantiated
- Over functional containers in `SPARK.Containers.Functional.*.Higher_Order`
  - Available for vectors, lists, sets, maps
  - Functions for mapping, filtering, summing, counting
  - Take access-to-function parameter to apply to all collection
  - Functions and lemmas use **Higher_Order_Specialization**

Automatic Instantiation

- By default, lemma only available where called explicitly
- Annotation **Automatic_Instantiation** available on lemmas
  - Declaration of lemma must follow function declaration
  - Axiom for lemma put in proof context for calls to the function
- Can be combined with **Higher_Order_Specialization**
  - Used in SPARK Higher Order Library

Dealing with Hard Proofs

Reducing the Proof Context

- Large proof context confuses provers
- Lemmas allow reducing the proof context to a minimum
  - Precondition of the lemma
  - Definition of constants, types and subprograms used
- Pragma `Assert_And_Cut`
  - State property used as cut-point for instructions that follow
  - All variables in context are havoc'ed
  - Proof context may still be large, but fewer ground terms
- SPARK Library `SPARK.Cut_Operations`
  - Functions `By` and `So` to chain assertions
  - `By (A, B)` requires proving `B`, then `A` from `B`, and leaves only `A` in proof context
  - `So (A, B)` requires proving `A`, then `B` from `A`, and leaves both in proof context
  - Note: `A and then B` requires proving separately `A` and `B`
- Annotation `Hide_Info` and `Unhide_Info` used to hide/expose expression function or private part of package

Triggering Provers

- SMT provers use *triggers* to instantiate axioms
  - A trigger is a ground term usually appearing in the axiom
  - E.g. GNATPROVE generates trigger `f args` for axiom defining function `f` on arguments `args`
- Annotation `Inline_For_Proof` avoids definition of axiom
  - Instead direct definition given for function
  - Applicable to expression function, or function with postcondition `F'Result = ...`
- Call to expression function is inlined when it is a conjunction
  - This facilitates proof in general
  - ... but it removes a potential trigger, making other proofs more difficult!
  - Disable such inlining with an explicit `Post => True`

Dealing with Equality

- Equality in SPARK $\neq$ logical equality
- Equality in SPARK on type `T` is:
  - The user-defined primitive equality if present
  - The predefined equality otherwise, based on the equality of components:
    - Using the primitive equality on record subcomponents
    - Using the predefined equality on other subcomponents
- Predefined equality on arrays ignores value of bounds
- In general, `A = B` does not imply `F (A) = F (B)`
  - Possible to state a lemma proving this property
  - Or use annotation `Logical_Equal` on equality function
    - GNATprove checks that this is sound

Computing with Provers

- Provers not a good fit for computing values
- Proving properties on large constants can be hard
    - E.g. to check validity of configuration data
- Use ghost code to prove intermediate steps
    - Loops without loop invariants of up to 20 iterations are unrolled
    - Calls to local subprograms without contract are inlined
    - Proof by induction using loops with loop invariants
    - Define lemmas for shared proofs
- Alternative is to execute these assertions at runtime

Lab

Auto-active Proof Lab

- Find the `130_autoactive_proof` sub-directory in `source`
    - You can copy it locally, or work with it in-place
- Copy locally **sparklib.gpr** from your SPARK install and set `Object_Dir`
- In that directory, open the project `lab.gpr` in GNAT Studio
    - Or, on the command-line, do `gnatstudio -P lab.gpr`
- Unfold the source code directory (.) in the project pane

Selection Sort

- Find and open the files `sort_types.ads`, `sort.ads` and `sort.adb` in GNAT Studio
  - Study the specification of procedure `Selection_Sort`. Is it a full functional specification?
  - Study the implementation of procedure `Selection_Sort`. Does it implement selection sort algorithm?
- Add a full functional contract to procedure `Swap` and prove it
- Add a full functional contract to procedure `Index_Of_Minimum` and prove it
- Start by proving that `Values` is sorted when returning from procedure `Selection_Sort`
  - Add a loop invariant to procedure `Selection_Sort`
- Then prove that the output value of `Values` is a permutation of its input value
  - Hint: you need to update global ghost variable `Permutation`
- Run GNATprove to prove the file

Selection Sort - Variations

- Find the `13_autoactive_proof` sub-directory in `answers`
  - It contains two sub-directories `answer1` and `answer2`
- In directory `answer1`, open the project `lab.gpr` in GNAT STUDIO
  - This solution follows the specification you worked on. Study it.
  - Run GNATPROVE to prove the file
- In directory `answer2`, open the project `lab.gpr` in GNAT STUDIO
  - This is another solution following a different specification for permutations. It uses multisets from the SPARK Library. Study it.
  - Run GNATPROVE to prove the file
- Compare the two solutions
  - Which specification is more readable to you?
  - Which proof is easier for you?

Further Readings

- The second solution is based on the example in subsection "A Concrete Example: a Sort Algorithm" of section 7.9.3.2 of the SPARK User's Guide on "Manual Proof Using User Lemmas".
  - Read it and discuss with the course instructor.
- The blog post https://blog.adacore.com/i-cant-believe-that-i-can-prove-that-it-can-sort presents 18 useful tips in the context of the proof of another sorting algorithm.
  - Read it and discuss with the course instructor.

Summary

Auto-active Proof

- Not all proofs are easy
- Understand tool messages
  - Messages guide you to help the tool
  - Many useful parts in a message
- Auto-active proof needed for harder proofs
  - Intermediate assertions
  - Ghost code for specification and verification
  - Lemmas to separately prove properties
- Ghost code has no effect
  - Compiler can ignore it or compile it

State Abstraction

Introduction

Subprogram Contracts and Information Hiding

- Subprogram contracts expose variables and types
  - In preconditions with aspect `Pre`
  - In postconditions with aspect `Post`
- Variables and types mentioned directly need to be visible
- Information hiding forbids exposing variables and types
  - Global variables in the private part or body
  - Use of private types for parameters
- Solution is to use (ghost) query functions

```
type T is private;
function Get_Int (X : T) return Integer;
function Get_Glob return Integer;

procedure Proc (X : in out T)
with
  Pre  => Get_Int (X) /= Get_Glob;
  Post => Get_Int (X) = Get_Glob;
private
type T is ...    -- returned by Get_Int
Glob : Integer; -- returned by Get_Glob
```

Dependency Contracts and Information Hiding

- Dependency contracts expose variables
    - In data dependencies with aspect `Global`
    - In flow dependencies with aspect `Depends`
- These variables need to be visible
- Information hiding forbids exposing variables
- Solution is to use *state abstraction*
    - Names that denote one or more global variables
    - They represent all the *hidden state* of the package

Abstract States

Abstract State

- Abstract state declared with aspect `Abstract_State`
  - On the package spec

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

- More than one abstract state is possible

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

- The number of abstract states is a choice
  - More abstract states make the contracts more precise
  - ...but expose more details
  - ...that may not be useful for callers

State Refinement

- *State refinement* maps each abstract to variables
  - All hidden variables must be constituents of an abstract state
  - This includes variables in the private part and in the body
- Refined state declared with aspect `Refined_State`
  - On the package body

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

- More than one abstract state is possible

```
package body Stack with
  Refined_State => (Top_State => Top,
                    Content_State => Content)
is ...
```

State in the Private Part

- Private part of package is visible when body is not
  - From client code that only sees the package spec
  - State refinement is not visible in that case
  - What is the abstract state for variables in the private part?
    - This is a problem for flow analysis
- Partial refinement declared with aspect `Part_Of`
  - On variables in the private part
  - Even when only one abstract state declared

```
package Stack with
  Abstract_State => The_Stack
is ...
private
   Content : T       with Part_Of => The_Stack;
   Top     : Natural with Part_Of => The_Stack;
end Stack;
```

- When package body is present, confirmation in `Refined_State`

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

Additional States

Nested Packages

- State of package P includes state of nested packages N
  - N may have visible state (variables in the public part, abstract states)
  - N may have hidden state (variables in the private part of body)
  - If N is visible
    - Its visible state is visible for P too
    - As are its own abstract states
    - Its hidden state is a constituent of its own abstract states
  - If N is hidden
    - Its visible state is a constituent of P's abstract states
    - As are its own abstract states
    - Its hidden state is a constituent of its own abstract states

```
package P with Abstract_State => State is
  package Visible_Nested with
    Abstract_State => Visible_State is
    ...
end P;
package body P with
  Refined_State => (State => Hidden_Nested.Hidden_State)
is
  package Hidden_Nested with
    Abstract_State => Hidden_State is
```

Child Packages

- State of package `P` includes state of private child package `P.Priv`
  - Its visible state is a constituent of `P`'s abstract states
  - As are its own abstract states
  - Its hidden state is a constituent of its own abstract states
- The visible state of private child packages should have `Part_Of`
- The state of public child packages is not concerned

```
package P with Abstract_State => State is ...

private package P.Priv with
   Abstract_State => (Visible_State with Part_Of => State)
is
    Var : T with Part_Of => State;
    ...

package body P with
  Refined_State => (State => (P.Priv.Visible_State,
                              P.Priv.Var, ...
```

Constants with Variable Input

- Constants are not part of the package state usually
  - Same for named numbers

```
package P is
   C : constant Integer := 42;
   N : constant := 42;
```

- Some constants are part of the package state
  - When initialized from variables, directly or not
  - They participate in information flow
  - These are *constants with variable input*

```
package body Stack with
  Refined_State => (The_Stack => (Content, Top, Max))
is
  Max     : constant Natural := External_Variable;
  Content : Component_Array (1 .. Max);
  Top     : Natural;
  --  Max has variable input. It must appear as a
  --  constituent of The_Stack
```

Dependency Contracts

Data Dependencies

- Abstract states are used in `Global` contracts
    - Abstract state represents all its constituents
    - Mode is the aggregate of all modes of constituents
        - As if the abstract state was a record with constituents as components

```
package Stack with
   Abstract_State => (Top_State, Content_State)
is
   procedure Pop  (E : out Component) with
     Global => (Input  => Content_State,
                In_Out => Top_State);
```

```
package Stack with
  Abstract_State => The_Stack
is
   procedure Pop  (E : out Component) with
     Global => (In_Out => The_Stack);
```

Flow Dependencies

- Abstract states are used in Depends contracts

```
package Stack with
   Abstract_State => (Top_State, Content_State)
is
   procedure Pop  (E : out Component) with
     Depends => (Top_State => Top_State,
                 E         => (Content_State, Top_State));

package Stack with
   Abstract_State => The_Stack
is
    procedure Pop  (E : out Component) with
      Depends => ((The_Stack, E) => The_Stack);
```

Dependency Refinement

- Inside the body, one can specify refined dependencies
  - Referring to constituents instead of abstract states
  - With aspects for refined dependencies on the subprogram body
    - Aspect `Refined_Global` for data dependencies
    - Aspect `Refined_Depends` for flow dependencies
- GNATprove verifies these specifications when present
- GNATprove generates those refined contracts otherwise
  - More precise flow analysis inside the unit

Package Initialization

Data Dependencies of a Package

- The *package elaboration* executes code
  - For all declarations in the package spec
  - For all declarations in the package body
  - And the statements at the end of the package body
- Only package state can be written during package elaboration
  - A package cannot write the state of another package in SPARK
- Aspect `Initializes` specifies state initialized during elaboration
  - If present, must be complete, including visible and hidden state
  - Otherwise, GNATprove generates it
  - Similar to the outputs of mode `Output` for the package elaboration

```
package Stack with
   Abstract_State => The_Stack,
   Initializes    => The_Stack
is
   -- Flow analysis verifies that Top and Content are
   -- initialized at package elaboration.
```

Flow Dependencies of a Package

- Initialization of package state can depend on other packages
  - This dependency needs to be specified in aspect `Initializes`
  - If no such aspect, GNATPROVE also generates these dependencies
  - Similar to the `Depends` aspect for the package elaboration

```
package P with
   Initializes => (V1, V2 => External_Variable)
is
   V1 : Integer := 0;
   V2 : Integer := External_Variable;
end P;
-- The association for V1 is omitted, it does not
-- depend on any external state.
```

Lab

State Abstraction Lab

- Find the `140_state_abstraction` sub-directory in `source`
  - You can copy it locally, or work with it in-place
- In that directory, open the project `lab.gpr` in GNAT Studio
  - Or, on the command-line, do `gnatstudio -P lab.gpr`
- Unfold the source code directory (.) in the project pane

Abstract State

- Define an abstract state called `The_State` to hold all of the state of package `Basics`
- Move all the state of package `Basics` into its private part with suitable aspects `Part_Of`
- Define the state refinement in the package body
- Run GNATprove in flow analysis mode

Dependency Contracts

- Update the data dependency and flow dependency contracts to use `The_State`
- Run GNATPROVE in flow analysis mode
  - There should be no check messages, only a warning:

    ```
    no procedure exists that can initialize abstract state
    ```
- Add a procedure `Init_The_State` that initializes all of the state
  - The body of this procedure can simply call `Init_The_Rec` and `Init_The_Table`
  - Do you understand how GNATPROVE checks that this is correct?

Summary

State Abstraction

- Abstract state represents hidden state of a package
  - Variables in the private part or body
  - Visible state of nested packages (variables and abstract states)
  - Visible state of private child packages
  - Constants with variable input
- Each abstract state must be refined into constituents
  - Annotation `Part_Of` needed on declarations in the private part
- Dependency contracts use abstract states to refer to hidden state
- Initialization at elaboration specified with aspect `Initializes`
  - This concerns both visible and hidden state
  - This replaces aspects `Global` and `Depends` for package elaboration

SPARK Boundary

Introduction

Modelling the System

- Special variables used to interact with the system
  - Usually marked as volatile for the compiler
  - This prevents compiler optimizations
- GNATprove needs to model these interactions
  - Both in flow analysis and proof
  - Distinction between different kinds of interactions
- This modelling is used as assumptions by GNATprove
  - These assumptions need to be reviewed

Integrating SPARK Code

- Not all the program is in SPARK usually
  - The Operating System (if present) is rarely in SPARK
  - Some services (logging, input/output) may not be in SPARK
  - Only a core part may be in SPARK
- User needs to specify the boundary of SPARK code
- GNATPROVE needs to model interactions with non-SPARK code
- GNAT needs to compile SPARK and non-SPARK code together

System Boundary

Volatile Variables (1/2)

- Volatile variable is identified by aspect `Volatile`
  - Either on the variable or its type
  - Aspect `Atomic` implies `Volatile`
- GNATprove assumes that volatile variable may change value
  - Each read gives a different value
  - Even if read is preceded by a write

```
Var  : Integer := 42 with Volatile;
Val1 : Integer := Var;
Val2 : Integer := Var;
pragma Assert (Val1 = 42);   -- unprovable
pragma Assert (Val1 = Val2); -- unprovable
```

Volatile Variables (2/2)

- Volatile variable typically has its address specified

```
Var : T with
  Volatile,
  Address =>
    System.Storage_Elements.To_Address (16#CAFECAFE#);
```

- A volatile variable can only occur in a *non-interfering context*
  - On either side of an assignment
    - As whole variable or as prefix when accessing a component
  - But not as part of a more complex expression

```
Var := Var + 1; -- illegal

Tmp : Integer := Var;
Var := Tmp + 1; -- legal
```

Volatility Properties

- Four different properties of volatile variables in SPARK
  - `Async_Readers` - asynchronous reader may read the variable
  - `Async_Writers` - asynchronous write may write to the variable
  - `Effective_Reads` - reading the variable changes its value
  - `Effective_Writes` - writing the variable changes its value
- Each is a Boolean aspect of volatile variables
  - By default a volatile variable has all four set to `True`
  - When one or more are set explicitly, others default to `False`

Volatility Properties - Examples

- A sensor (program input) has aspect
  - `Async_Writers => True`
- An actuator (program output) has aspect
  - `Async_Readers => True`
- A machine register (single data) has aspects
  - `Effective_Reads => False`
  - `Effective_Writes => False`
- A serial port (stream of data) has aspects
  - `Effective_Reads => True`
  - `Effective_Writes => True`

Volatile Functions

- Some volatile variables can be read in functions
    - When `Async_Writers` and `Effective_Reads` are set to `False`
    - These correspond to program outputs
- *Volatile functions* can read volatile inputs
    - When `Async_Writers` is set to `True`
    - Function needs to have the aspect `Volatile_Function`
- Functions (even volatile ones) cannot read some volatile variables
    - When `Effective_Reads` is set to `True`
    - A read is a side-effect, which is forbidding in SPARK functions
    - Unless the function has aspect `Side_Effects`
- A call to a volatile function must appear in a non-interfering context
    - Same as a read of a volatile variable

External State

- Abstract state may have volatile variables as constituents
  - Abstract state needs to have aspect `External`
- An external state is subject to the four volatility properties
  - All volatility properties set to `True` by default
  - Specific properties can be specified like for volatile variables
  - An external state with `Prop` set to `False` can only have
    - Non-volatile constituents
    - Volatile constituents with `Prop` set to `False`
- Special case for external state always initialized
  - An external state with `Async_Writers` set to `True`
  - The asynchronous writer is responsible for initialization

Effect of Volatility on Flow Analysis

- A variable with `Effective_Reads` set to `True`
  - Has its value influenced by conditions on branches where read happens

```
Var : Integer := 42 with Volatile, Effective_Reads;
if Cond then
   Val := Var;
end if;
-- value of Var here depends on Cond
```

- A variable with `Effective_Writes` set to `True`
  - Never triggers a warning on unused assignment

```
Var : Integer := 42 with Volatile, Effective_Writes;
Var := 1; -- previous assignment is not useless
```

Effect of Volatility on Proof

- A variable is *effectively volatile for reading* if
  - It has `Async_Writers` set to `True`
  - Or it has `Effective_Reads` set to `True`
- The value of such a variable is never known
- Same for external state with these volatility properties

```
Var : Integer := 42 with Volatile, Async_Readers;
pragma Assert (Var = 42); -- proved

Var : Integer := 42 with Volatile, Async_Writers;
Val : Integer := Var;
pragma Assert (Val = 42); -- unprovable
```

Software Boundary

Identifying SPARK Code

- SPARK code is identified by pragma/aspect `SPARK_Mode` with value `On`
- Other values: `Off` or `Auto`
  - `Off` to exclude code
  - `Auto` to include only SPARK-compatible declarations (not bodies)
- Default is `On` when using `SPARK_Mode` without value
- Default is `Auto` when `SPARK_Mode` not specified
  - `Auto` can only be used explicitly in configuration pragmas

Sections with SPARK_Mode

- Subprograms can have 1 or 2 sections: spec and body
  - `SPARK_Mode` can be `On` for spec then `On` or `Off` for body
- Packages can have between 1 and 4 sections:
  - package spec visible and private parts, package body declarations and statements
  - `SPARK_Mode` can be `On` for some sections then `On` or `Off` for the remaining sections
- `SPARK_Mode` **cannot** be `Off` for a section
  - Then `On` for a following section
  - Or `On` inside the section

Inheritance for SPARK_Mode on Subprogram

- Value of `SPARK_Mode` inherited inside subprogram body
  - Nested subprogram or package can have `SPARK_Mode` with value `Off`
- Value for subprogram spec **not** inherited for subprogram body

Inheritance for SPARK_Mode on Package

- Value On of SPARK_Mode inherited inside package spec/body
  - Nested subprogram or package can have SPARK_Mode with value Off
- Value Off of SPARK_Mode inherited inside package spec/body
- Value Auto of SPARK_Mode inherited inside package spec/body
  - Nested subprogram or package can have SPARK_Mode with value On or Off
- Value for package spec visible part inherited in private part
- Value for package body declarations inherited for body statements
- Value for package spec **not** inherited for package body

Syntax for SPARK_Mode

- Aspect on declarations (pragma is also possible)
- Pragma in other cases

```
pragma SPARK_Mode; -- library-level pragma

with Lib; use Lib;

package P
  with SPARK_Mode -- aspect on declaration
is
   ...
   procedure Proc
     with SPARK_Mode => Off; -- aspect on declaration
   ...
private
   pragma SPARK_Mode (Off); -- pragma for private part
   ...
end P;
```

Generics and `SPARK_Mode`

- Remember: only generic instances are analyzed
- If generic spec/body has no value of `SPARK_Mode`
  - Each instance spec/body inherites value from context
  - As if the instantiation was replaced by the instance spec and body
- If generic spec/body has `SPARK_Mode` with value `On`
  - Each instance spec/body has `SPARK_Mode` with value `On`
  - Unless context has value `Off`, which takes precedence
    - Remember: `SPARK_Mode` **cannot** be `Off` then `On`
- If generic spec/body has `SPARK_Mode` with value `Off`
  - Each instance spec/body has `SPARK_Mode` with value `Off`
- Value of library-level pragma inside generic file **not** inherited in instance

Typical Use Cases

- Unit fully in SPARK
  - Spec and body both have `SPARK_Mode` with value `On`
- Spec only in SPARK
  - Spec has `SPARK_Mode` with value `On`
  - Body has no `SPARK_Mode` or with value `Off`
- Package spec is partly in SPARK
  - Visible part of spec has `SPARK_Mode` with value `On`
  - Private part of spec has `SPARK_Mode` with value `Off`
  - Body has no `SPARK_Mode` or with value `Off`
- Package is partly in SPARK
  - Spec and body both have `SPARK_Mode` with value `On`
  - Some subprograms inside have `SPARK_Mode` with value `Off` on spec and body

Multiple Levels of Use (1/2)

- Level 1: `SPARK_Mode` as a configuration pragma
- `SPARK_Mode` can be specified in a global/local configuration pragmas file
  - Configuration pragmas file referenced in the GNAT project file
  - Only for `SPARK_Mode` with value `On`
- `SPARK_Mode` can be specified as library-level pragma in a file
  - Initial pragmas in a file before with/use clauses
  - Takes precedence over value in configuration pragmas file
  - Typically for `SPARK_Mode` with value `On` or `Off`
  - Can be used with explicit value `Auto`
    - Useful when configuration pragmas file has value `On`

Multiple Levels of Use (2/2)

- Level 2: `SPARK_Mode` as a program unit pragma
- `SPARK_Mode` can be specified on top-level subprogram or package
  - Takes precedence over value in library-level pragmas
  - Only for `SPARK_Mode` with value `On` or `Off`
- `SPARK_Mode` can be specified on nested subprogram or package
  - Takes precedence over inherited value from context
  - Only for `SPARK_Mode` with value `On` or `Off`

Integrating SPARK and Ada Code

- SPARK code has `SPARK_Mode` with value `On`
- Ada code has no `SPARK_Mode` or with value `Off`
- GNAT compiles all code together
- Contracts on Ada subprograms must be correct
    - As if the subprogram was implemented in SPARK
    - Precondition must prevent RTE in subprogram (for Silver level and above)
    - Postcondition must be respected by subprogram
    - Data dependencies must be either generated or accurate
        - This may require introducing abstract states for Ada units

Integrating SPARK and C Code (1/2)

- GNAT data layout follows C ABI by default
  - Representation clauses may change the default
  - Aspect `Pack` forces data packing
- Subprograms used across the boundary
  - Must have aspect `Convention => C`
  - Must be marked with aspect `Import` or `Export`
  - Must have their C name given in aspect `External_Name`
- Parameters of these subprograms
  - Ada mode `in out` → C pointer
  - Ada record/array → C pointer
  - Ada scalar → C scalar

Integrating SPARK and C Code (2/2)

- Standard library units
  - `Interfaces` defines fixed-size scalar types
  - `Interfaces.C` defines C standard scalar types
  - `Interfaces.C.Strings` defines character and string conversion functions between Ada and C
- SPARK Library units
  - `SPARK.C.Strings` defines wrapper on `Interfaces.C.Strings` for mutable strings based on ownership
  - `SPARK.C.Constant_Strings` defines wrapper on `Interfaces.C.Strings` for read-only strings (aliasing **is** allowed)

Integrating SPARK and Other Programming Languages

- Based on integration of Ada with other languages
  - Standard support for COBOL and Fortran
  - GNAT specific backends for Java and .NET
  - Based on C integration for C++, Rust, Python...
- C-Based Integration
  - Same as for integrating with C code on both sides
  - Use same external name (no mangling)
- Thin binding and thick binding
  - *Thin binding* matches closely constructs at C level
  - *Thick binding* matches SPARK semantics
  - It is common to have both
    - Thin binding may be auto-generated (e.g. using `gcc -fdump-ada-spec`)
    - Thick binding defines wrappers around thin binding

Integrating with Main Procedure Not in Ada

- GNAT compiler generates startup and closing code
  - Procedure `adainit` calls elaboration code
  - Procedure `adafinal` calls finalization code
  - These are generated in the file generated by GNATBIND
- When using a main procedure not in Ada
  - Main procedure should declare `adainit` and `adafinal`

    ```
    extern void adainit (void);
    extern void adafinal (void);
    ```
  - Main procedure should call `adainit` and `adafinal`
- When generating a stand-alone library
  - Specify interface units with `Library_Interface` in project file
  - GNAT then generates library initialization code
    - This code is executed at library loading (depends on platform support)

Modelling an API

- API may be modelled in SPARK
  - Implementation may be in Ada, C, Rust...
  - Implementation may be in the Operating System
- Relevant global data should be modelled
  - As abstract states when not accessed concurrently
  - As external states when accessed concurrently
- API subprogram contracts model actual behavior
  - Data dependencies must reflect effects on global data
  - Functional contracts can model underlying automatons
    - Possibly defining ghost query functions, e.g. `Is_Open` for a file
    - Ghost function may be marked `Import` when not implementable

Modelling an API - Example

- Standard unit `Ada.Text_IO` is modelled in SPARK
  - Subprograms can be called in SPARK code
  - File system is not precisely modelled

```
package Ada.Text_IO with
  SPARK_Mode,
  Abstract_State => File_System,
  Initializes    => File_System,
is
   type File_Type is limited private with
     Default_Initial_Condition => (not Is_Open (File_Type));

   procedure Create (File : in out File_Type; ...)
   with
     Pre      => not Is_Open (File),
     Post     => Is_Open (File) and then ...
     Global   => (In_Out => File_System),
 Exceptional_Cases =>
   (Name_Error | Use_Error => Standard.True);

   function Is_Open (File : File_Type) return Boolean with
     Global => null;
```

Modelling an API to Manage a Resource

- Managing a resource may require
  - Preventing aliasing of the resource
    - e.g. with limited type as in `Ada.Text_IO.File_Type`
  - Requiring release of the resource
    - e.g. free memory, close file or socket, ...
- GNATprove can force ownership on a type
  - With `Annotate => (GNATprove, Ownership)`
    - On a private type
    - When private part of package has `SPARK_Mode` with value `Off`
  - Assignment transfers ownership of object
    - Similar to treatment of pointers in SPARK
    - GNATprove checks absence of aliasing
  - Possibility to specify a reclamation function, predicate, or value
    - GNATprove checks absence of resource leaks

Modelling an API to Manage a Resource - Example

```
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 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;
```

Assumptions

Quiz - Implicit Assumptions

Is the following code correct?

```
package Random_Numbers
  with SPARK_Mode
is
   function Random (From, To : Integer) return Integer
     with Post => Random'Result in From .. To;
private
   pragma SPARK_Mode (Off);
   ...
```

Quiz - Implicit Assumptions

Is the following code correct?

```
package Random_Numbers
  with SPARK_Mode
is
   function Random (From, To : Integer) return Integer
     with Post => Random'Result in From .. To;
private
   pragma SPARK_Mode (Off);
   ...
```

- No - GNATprove assumes that `Random` is a mathematical function
  - An abstract state should be added in package `Random_Numbers`
  - `Random` should be a procedure
  - A data dependency contract should be added for reads/writes to this abstract state
- No - GNATprove assumes that the postcondition of `Random` is always satisfied, even when `From > To`
  - A precondition `From <= To` should be added
  - The implementation must satisfy the postcondition

Tool Assumptions

- Results of flow analysis and proof are valid under assumptions
  - About the system behavior as modelled in SPARK
  - About parts of the code not in SPARK
  - About the hardware platform
- All assumptions should be reviewed and validated
  - Complete list in SPARK User's Guide section 7.3.7
- Common assumptions whether or not complete program in SPARK
- Additional assumptions
  - When only part of the program in SPARK
  - When GNATprove never called with all bodies available
  - When code not compiled with GNAT

Lab

SPARK Boundary Lab

- Find the `150_spark_boundary` sub-directory in `source`
  - You can copy it locally, or work with it in-place
- In that directory, open the project `lab.gpr` in GNAT Studio
  - Or, on the command-line, do `gnatstudio -P lab.gpr`
- Unfold the source code directory (.) in the project pane

System Boundary

- Find and open the files `alarm.ads` and `alarm.adb` in GNAT Studio
- Run GNATprove on the unit
  - Check that you understand the error messages.
- Specify correct volatility properties for `Temperature` and `Status`
  - `Temperature` is an input register
  - `Status` is an output port
- Rerun GNATprove on the unit
  - Fix the SPARK violations in the implementation
  - Hint: you need to mark one of the functions as a volatile function
- Add an external state `State` with both `Temperature` and `Status` as constituents
  - What is the problem?
- Add separate external states with suitable volatile properties for `Temperature` and `Status`
  - The unit should be fully proved
- Review warnings and mark variables with aspect `Warnings => Off`

Software Boundary

- Find and open the files `random_numbers.ads` and `random_numbers.adb` in GNAT Studio
- Run GNATprove on the unit
  - Check that you understand the error message.
- Add aspect `SPARK_Mode` to the package body with value `Off`
- Run GNATprove on the unit
  - Check that there are no messages.
  - Is the spec compatible with SPARK?
- Complete the spec so that it is compatible with SPARK

Integration with Other Programming Languages

- Find and open the file `main.adb` in GNAT Studio
- Run GNATprove on the unit
  - Fix the warnings with suitable annotations on the declaration of `Swap`
- Add a suitable postcondition on `Swap`
  - Check that you can prove after the call that the values of `X` and `Y` have been swapped
  - Hint: add a suitable assertion
- Compile the code of `main.adb`

```
gcc -c main.adb
```

Integration with C

- Compile a C implementation for swap in `swap.c`, link it with the SPARK code, and run the executable

```
gcc -c swap.c
gnatbind main
gnatlink main swap.o
./main
```

- Or declare the main and languages used in the project file

```
for Main use ("main.adb");
for Languages use ("Ada", "C");
```

  and build the project with GPRbuild

- What assumptions did you make on the C implementation?
  - Discuss these with the course instructor.

Integration with Rust

- Compile a Rust implementation for swap in `swap.rs`, link it with the SPARK code, and run the executable

```
rustc --crate-type=lib --emit=obj swap.rs
gnatbind main
gnatlink main swap.o
./main
```

- Or build a Rust library with cargo and link that library with the SPARK code
- What assumptions did you make on the Rust implementation?
  - Discuss these with the course instructor.

Summary

SPARK Boundary

- System (hardware, OS) can be modelled in SPARK
    - Using volatile variables and external states
    - With precise volatility properties
- SPARK software boundary defined by aspect/pragma `SPARK_Mode`
    - Fine-grain integration of SPARK and non-SPARK code is possible
- Integration with other programming languages
    - Easiest between SPARK and Ada
    - Easy between SPARK and C
    - Usually based on C integration for other languages
- Formal verification is based on assumptions
    - Assumptions at the boundary need to be reviewed