4: SPARK: Language & Tools for Static Verification

This section provides

- a quick intro for the
  - SPARK language and the
  - SPARK toolset
- and a couple of simple examples.

Read: Sections 1–5 of the SPARK 2014 User's Guide ⟨https://docs.adacore.com/spark2014-docs/html/ug/⟩,

What is “SPARK”

ask Wikipedia

. . .

Computer Science:

Cisco Spark (application) , a collaboration application and platform now part of the Webex Teams application

Spark (application) , a mobile email application for iOS devices by Readdle

SPARK (programming language)

Spark (software) , a web application framework.

Spark (cellular automaton) , a type of pattern in Conway’s Game of Life and related rules.

Spark (XMPP client) , an instant messaging client.

Apache Spark , a cluster computing framework

. . .

→ SPARK (programming language)

from Wikipedia, again

SPARK is a formally defined computer programming language based on the Ada programming language, intended for the development of high integrity software used in systems where predictable and highly reliable operation is essential. It facilitates the development of applications that demand safety, security, or business integrity.

The entry isn't wrong but incomplete. SPARK is both

- a programming language (actually, a subset of Ada)
- and a toolset to support static verification.

Without the toolset, nobody would care abouit the language.

History

- Pre-Ada: SPADE: **S**outhampton **P**rogram **A**nalysis **D**evelopment **E**nvironment (for Pascal)
- Ada: SPARK **SP**ADE **A**da **K**e**R**nel
  - Following the revisions of Ada: SPARK 83, SPARK 95, SPARK 05
- Ada 2012 / SPARK 2014: a very significant change ($\rightarrow$ next slide)

From SPARK 05 to SPARK 2014

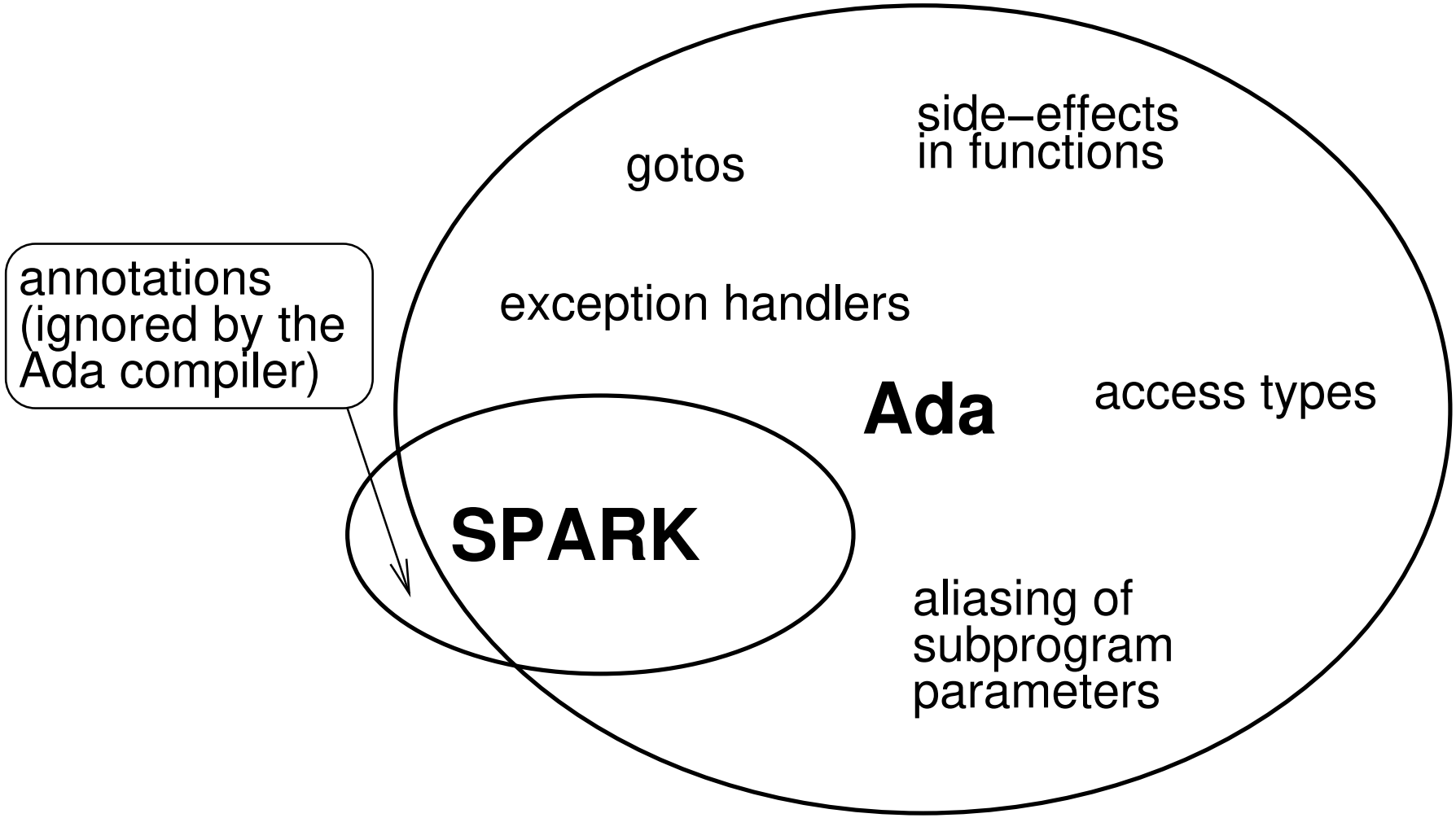
SPARK 05:

```
procedure Swap(X, Y: in out Integer);
--# derives X from Y \& Y from X;
--# post (X = Y~) and (Y = X~);
```

SPARK 2014:

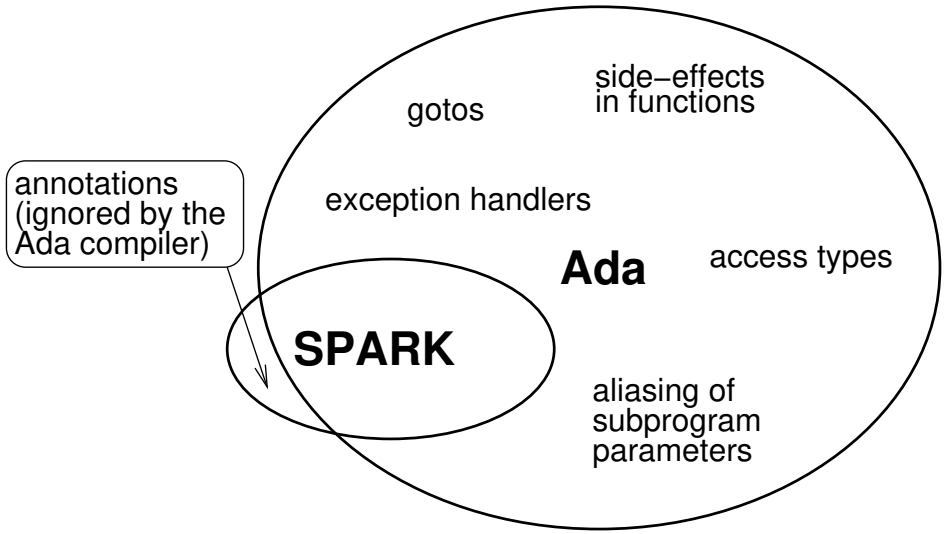
```
procedure Swap (X : in out Integer;
                Y : in out Integer) with
   Global  => null,
   Depends => (X => Y, Y => X),
   Post    => ((X = Y'Old) and (Y=X'Old));
```

The Relationship between SPARK and Ada



No Access Types! No Gotos!

- access types
  - worry about memory safety
  - extremely difficult to analyze (heap = one huge array)
- goto
  - bad style: “goto considered harmful”
  - difficult to analyze (especially when going backward)



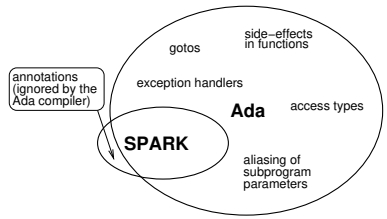
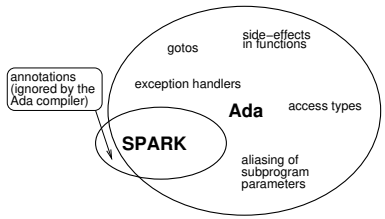
Why no side-effects in functions?

- ```
  X := 10;
  Y := Fun(0);
  Put(X); -- This prints 0. But why?
  ```

- ```
  function Fun(N: Integer) return Integer is
  begin
     X := N mod 2; -- OK in Ada, prohibited in SPARK.
     return N;
  end;
  ```

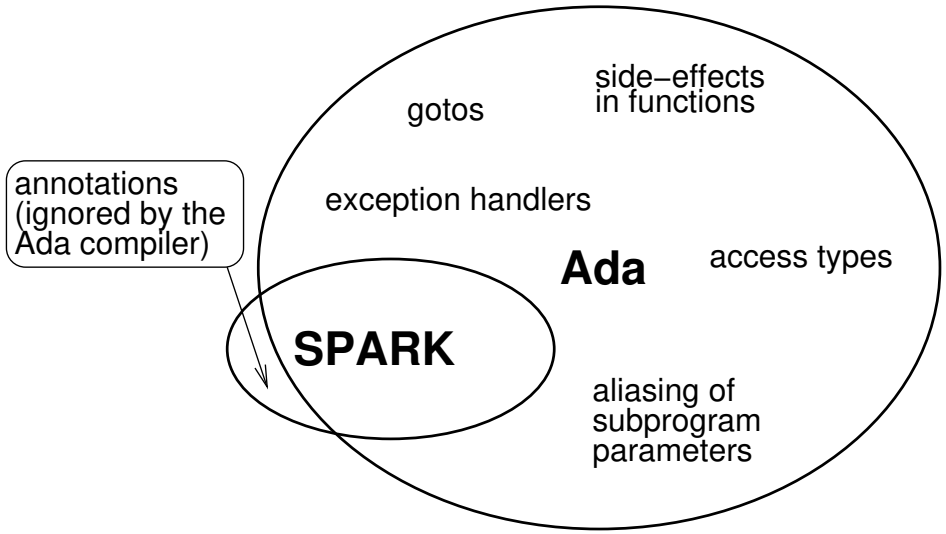
- ```
  X := 10;
  Z := Fun(1) + Fun(2);
  Put(X); -- Does this print 0 or 1?
  ```

- Both 0 and 1 possible: order of expression evaluation not defined.


  ```

Why no aliasing of subprogram parameters?

What is that, at all?

- ```
  type Math_Int is ...; -- unbounded integers
  procedure Divide(X, Y: Math_Int;
                   Result, Remainder out Math_Int);
  ...
  Divide (A, B, A, A); -- OK in Ada, prohbited in SPARK.
  ```

- The semantic of calling “Divide(A,B,A,A)” is undefined in Ada:
  - Subprograms can copy their parameters forward and backward. Then, the result of calling “Divide (A, B, A, A);” should be either Result or Remainder – or perhaps a mixture of both.
  - Subprograms can use pointers to their parameters. Then, after calling “Divide (A, B, A, A);” the value of A is almost impossible to predict.


  ```

Why no exception handlers?

- complex control flow difficult to analyze (as discussed before)
- also:
  - early programming languages did not have exceptions and did crash unexpectedly (e.g., “division by 0”) ($\rightarrow$ next slide)
  - exceptions as an approach to “tame” crashes (improved reliability)
  - SPARK: static analysis to prove that nothing unexpected happens (improves reliability more) ($\rightarrow$ next slide +1)



Early Languages (Here: C)

from the textbook:

```
fptr = fopen("~/comments.txt",
             "a");
// open for append

if(fptr == NULL)
{
  error_handling()
};

if ((100*done) / full > 50) {
  fprintf(fptr, ``> 50 %'');
  // store comment in file
  // hope that full is never 0!
}

fclose(fptr);
```

lazy programmers' time bomb

```
fptr = fopen("~/comments.txt",
             "a");
// open for append

// forgot the error handling!

if ((100*done) / full > 50) {
  fprintf(fptr, ``> 50 %'');
  // store comment in file
}

// if fopen above failed:
//   crash -- here we come!
// (same if full=0)

fclose(fptr);
```

The Same in SPARK

```
procedure Append(File: File_of_Item; It: Item) with
  Pre => Valid(File);
```

textbook-like:

```
File := Open("~/comments.txt",
             Append_Mode);

if Valid(File) then
   if (100*Done)/Full > 50 then
      Append(File, Comment);
   end if;
   Close(File)
else
   Error_Handling;
end if;
```

(Must prove Full>0.)

SPARK catches the time-bomb:

```
File := Open("~/comments.txt",
             Append_Mode);

if (100*Done)/Full > 50 then
   Append(File, Comment);
end if;
-- SPARK proof fails:
--    "cannot prove
--     precondition!''

Close(File)
```

(Even if Full>0 proven.)

Mixing Ada and SPARK

```
pragma SPARK_Mode(on);
package X is
   type T is ...;

   procedure P(...)
   with
      Pre  => ...,
      Post => ...;

   function F(...) return ...
   with
      Pre  => ...,
      Post => ...;
end X;
```

```
pragma Spark_Mode(Off);
package body X is
   -- SPARK doesn't check body

   procedure P(...) is
   begin
      ...
   end P;

   function F(...) return ... is
   begin
      ...
   end F;
end X;
```

- SPARK will check compilation units **with**ing X
- **assuming** (without proof!) the body of X keeps its promises

The SPARK toolset

1. SPARK checks confirmity with SPARK rules and restrictions (no gotos no side-effects in functions, no exception handlers, no access types, no aliasing, ...).
2. SPARK performs flow analysis. It generates information about
   - global variables,
   - and dependencies.

   If flow specifications are given (**Global**, **Depends**), SPARK verifies that the implementation conforms with the specification.
   If no flow specifications are given, SPARK derives the flow information from the implementation.
3. And SPARK performs the functional analysis: SPARK
   - proves that no exceptions are raised,
   - proves that all calls to subprogram satisfy the subprograms' preconditions,
   - proves the postcondition (if given) *(partial correctness)*,
   - and proves the "loop variants" (if given) *(termination)*.

   For the proofs, SPARK assumes that the precondition holds.

Subprogram Contracts

beyond Ada 2012

```
procedure Decr_Abs (X : in out Integer) with
  Pre  => X /= 0,
  Post => (if X'Old < 0 then X = X'Old + 1
           elsif X'Old > 0 then X = X'Old - 1);
           -- else the result is unspecified
```

```
procedure Decr_Abs(X: in out Integer) with Contract_Cases
  => (X < 0 => X = X'Old + 1,
      X > 0 => X = X'Old - 1,
      X = 0 => True -- can do anything if X = 0,
                    -- but must specify this case, else
                    -- "contract cases might not be complete"
     );
```

- sometimes easier to read or maintain
- must be disjoint and complete (SPARK checks this)

To use SPARK, you shuold have a gnat project file

Here is one for homework problems: p.gpr

```
project P is
  for Source_Dirs use (".");

  package Compiler is
    for Default_Switches ("Ada")
    use ("-gnat12", "-gnato13");
  end Compiler;

  package Prove is
   for Switches use
   ("--report=fail",
    "--proof=progressive",
    "--warnings=continue",
    "--timeout=5");
  end Prove;

end P;
```

What does the project file do?

```
project P is
  for Source_Dirs use (".");

  package Compiler is
    for Default_Switches ("Ada")
    use ("-gnat12", "-gnato13");
  end Compiler;

...
```

- check/compile in the current directory
- `-gnat12`: use Ada 2012
- (`-gnato13`: later in this chapter)

What does the project file do? (2)

```
package Prove is
 for Switches use
 ("--report=fail",
  "--proof=progressive",
  "--warnings=continue",
  "--timeout=5");
end Prove;

end P;
```

- `--report=fail`: only report failures to prove checks (no success stories, no statistics)
- `--proof=progressive`: start with one formula per check; split into paths when needed
- `--timeout=5`: after 5 seconds, prover gives up (default: 1 sec.)

Also: `--mode=flow` and `--mode=prove` (default: `--mode=all`).

4.1: First Examples

spec (data flow, precondition):

```
1  procedure What
2    (A:    in      Integer;
3     B,C: in out Integer;
4     D:       out Integer) with

6    Global =>
7      null,
8
9    Depends =>
10     (B => null;
11      C => (A,B,C),
12      D => A),

14   Pre =>
15     A+1 in Integer and
16     (B+C) in Integer;
```

implementation:

```
1  procedure What
2    (A:    in      Integer;
3     B,C: in out Integer;
4     D:       out Integer) is
5    T: Integer := A;
6  begin
7     if T < 1 then
8        T := 1;
9     end if;
10    D := T+1;
11    C := B+C;
12    while C < 0 loop
13       C := C + T;
14    end loop;
15    B := 1;
16 end What;
```

Example: Linear Search

The basic specification for linear search is the following:

in: Array A, value V (no precondition)

out: Index I with A(I)=V (this is the postcondition)

```
type Arr is array (Positive range <>) of Item);
A: Arr(First, Last) := Read_Array(From_Somewhere);

procedure Lin_Search (Value: Item; Result: out Positive) with
  Post => A(Result)=Value;
```

Anything wrong with this specification? Yes, it is incomplete:

1. What shall we output if no such I exists?
   - **1.1** Raise an exception, of no J with A(J)=V exists.
   - **1.2** New precondition: Require the existence of J with A(J)=V.
   - **1.3** Define special value “none” not in the range of A.
     Output I=“none” if no J with A(J)=V exists.
2. If there are several such indices I, which will we output?

Fixing Problem 1.

1.1 Raise an exception: not permitted in SPARK.

```
procedure Lin_Search (Value: Item; Result: out Positive) with
  Pre  => (for some I in A'Range => A(I)=Value),
  Post => A(Result)=Value;
```

1.2 Require the existence of an index J with A(J)=V.

```
No_Such_Index: Natural := 0; -- no positive value

procedure Lin_Search (Value: Item; Result: out Natural) with
  Post =>
  (if (for all I in A'Range => A(I)/=Value) then
      Result = No_Such_Index
   else A(Result)=Value);
```

1.3 Define special value “none” to handle the case that no such index exists. (I prefer this!)

Dealing with problem 2.

2. If there are several such indices I, which will we output?

2.1 The first index I with A(I)=V.

2.2 The last index I with A(I)=V.

2.3 The first prime index I with A(I)=V,
and the last index I with A(I)=V if no such prime index exists.

⋮ …

2.∗ We don't care. (I prefer this!)

First Implementation

```
procedure Lin_Search (Value: Item; Result: out Positive) is
begin
   Result := A'First;
   A(Result) := Value;
end Lin_Search;
```

This matches the contract! But it is hardly what you would expect.

Oh, and what happens when A is empty?

Revised Specification

global flow specification

```
No_Such_Index: Natural := 0; -- no positive value

procedure Lin_Search (Value: Item; Result: out Natural) with
  Global => in A, -- read from global A,
                  -- don't write to any gloabl
  Post =>
  (if (for all I in A'Range => A(I)/=Value) then
      Result = No_Such_Index
   else A(Result)=Value);
```

The flow annotation "**Global** => A" makes sure that

1. A is the only global variable used, and
2. A is read from, but never written to.

Revised Specification

rewrite Lin_Search as a function

```
function Lin_Search (A: Arr; Value: Item) return Natural with
  Post =>
  (if (for all I in A'Range => A(I)/=Value) then
      Lin_Search'Result = No_Such_Index
   else A(Lin_Search'Result)=Value);
```

Why did we specify Lin_Search as a **procedure** with a **Global** flow annotation at all?

→ Because I needed an example for writing a **Global** flow annotation.

For the problem at hand, the **function** is by far better!

Second Implementation

```
function Linear_Search (A: Arr; Value: Item) Natural is
  Idx: Positive := A'First;
begin
  while Idx <= A'Last loop
     if A(Idx) = Value then
       return Idx;
     end if;
   end loop;
   return No_Such_Index;
end Linear_Search;
```

- This implementation would pass static verification.
- Nevertheless, it hardly ever does, what we want it to do!
- Why? (Surprised?)

Third Implementation: Loop Variant

```
function Linear_Search (A: Arr; Value: Item) Natural is
  Idx: Positive := A'First;
begin
   while Idx <= A'Last loop
     pragma Loop_Variant(Decreases => A'Last-Idx);
       -- this loop does not run forever!
     if A(Idx) = Value then
       return Idx;
     end if;
     Idx := Idx + 1;
   end loop;
   return No_Such_Index;
end Linear_Search;
```

- This fixes the main problem from the previoius code fragment.
- It still poor design.
- And in certain cases, it might raise a Constraint_Error. (When?)

Fourth Implementation: For-Loop

```
function Lin_Search (A: Arr; Value: Item) return Natural is
begin
   for I in A'Range loop
      if A(I) = Value then
         return I;
      end if;
   end loop;
   return No_Such_Index;
end Lin_Search;
```

A **for**-loop in Ada always terminates (no need for a loop variant).

But for the correctness proof, we still need a **loop invariant**:

- it must hold in first iteration,
- **if** it did hold in the previous iteration,
  then it also holds in the curren one, and
- it should allow us to prove the (missing part of the) postcondition.

($\rightarrow$Black- or whiteboard!)

4.2: Example: Seven Swaps

- task: swap two integer variables
- provide six flawed implementations and
- use `gnatprove` from the command line to find flaws
- our main program “test.adb” looks like this:

```
with Swap_It; use Swap_It;

procedure Test is
   X, Y: Integer;
begin
   X := 0; Y := 1;
   Swap(X, Y);
   if X = Y then
      raise Program_Error;
   end if;
end Test;
```

First Swap

```
package Swap_It is
   procedure Swap (X : in out Integer;
                   Y : in out Integer) with
     Global => null,
     Depends => (X => Y, Y => X);

end Swap_It;
```

```
package body Swap_It is

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

end Swap_It;
```

Analyzing First Swap

```
> gnatprove -Pswapper --mode=flow
Phase 1 of 2: frame condition computation ...
Phase 2 of 2: translation to intermediate language ...
swap_it.ads:2:20: ineffective import "X"
swap_it.ads:5:11: "X" missing from null dependency
swap_it.ads:5:31: "Y" depends on "Y"
swap_it.ads:5:31: "Y" does not depend on "X"
```

Flaw Found!

Second Swap

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

That looks allright – the additions and subtractions seem to cancel out.

Analyzing Second Swap

```
> gnatprove -Pswapper --mode=flow
Phase 1 of 2: frame condition computation ...
Phase 2 of 2: translation to intermediate language ...
swap_it.ads:5:23: "X" depends on "X"
swap_it.ads:5:31: "Y" depends on "Y"
```

Wait a minute: "`X := X + Y - Y`" cancels out, but the result is `X`, not `Y`!

Third Swap

```
procedure Swap (X : in out Integer;
                Y : in out Integer)
is
begin
   X := X + Y;
   Y := X - Y; -- = (X'Old + Y)      - Y      = X'Old
   X := X - Y; -- = (X'Old + Y'Old) - X'Old  = Y'Old
end Swap;
```

This time, the program logic appears to be OK (see comments)

Third Swap: `gnatprove` is not convinced

```
> gnatprove -Pswapper --mode=flow
Phase 1 of 2: frame condition computation ...
Phase 2 of 2: translation to intermediate language ...
swap_it.ads:5:23: "X" depends on "X"
swap_it.ads:5:31: "Y" depends on "Y"
```

This doesn't mean our program is wrong. It means, `gnatprove` cannot verify it is correct!

Flow analysis doesn't "know" the nature of operations, such as additions and subtractions cancelling out!

Is our program correct?

Third Swap: A Real Flaw

```
> gnatprove -Pswapper --mode=prove
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: translation to intermediate language ...
...
Phase 3 of 3: generation and proof of VCs ...
analyzing Swap3.Swap, 3 checks
swap_it.adb:7:14: overflow check not proved
swap_it.adb:8:14: overflow check not proved
swap_it.adb:9:14: overflow check not proved
```

Ouch! Additions and subtractions can overflow!

Swap Number Four

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

This is clearly wrong, and may overflow!

Fourth Swap

```
> gnatprove -Pswapper --mode=flow
Phase 1 of 2: frame condition computation ...
Phase 2 of 2: translation to intermediate language ...
```

So the information flow appears to be OK, now.

```
> gnatprove -Pswapper --mode=prove
...
swap_it.adb:6:23: overflow check not proved
```

But `X + 1` can overflow.

Swap Five: Fix the Overflow

```
procedure Swap (X : in out Integer;
                Y : in out Integer) with
  Global => null,
  Depends => (X => Y, Y => X),
  Pre     => X < Integer'Last;
```

```
> gnatprove -Pswapper --mode=flow
Phase 1 of 2: frame condition computation ...
Phase 2 of 2: translation to intermediate language ...
...

> gnatprove -Pswapper --mode=prove
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: translation to intermediate language ...
...
Phase 3 of 3: generation and proof of VCs ...
analyzing Swap_It.Swap, 1 checks
```

Why does `gnatprove` not find the flaw?

```
procedure Swap (X : in out Integer;
                Y : in out Integer) is
   -- same as swap four
   T: Integer := X + 1;
begin
   X := Y;
   Y := T;
end Swap;
```

If `X = 3` and `Y = 13`, what does `Swap(X, Y)` provide?

Why doesn't `gnatprove` find the flaw?

Swap Six: Specify, what you really want!

```
procedure Swap (X : in out Integer;
                Y : in out Integer) with
  Global  => null,
  Depends => (X => Y, Y => X),
  Pre     => X < Integer'Last,
  Post    => ((X = Y'Old) and (Y=X'Old));
```

(The implementation is the same as for swap four and five.)

Will `gnatprove` tell us what is wrong?

Swap Six

```
> gnatprove -Pswapper --mode=flow
Phase 1 of 2: frame condition computation ...
Phase 2 of 2: translation to intermediate language ...
...

> gnatprove -Pswapper --mode=prove
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: translation to intermediate language ...
,,,
Phase 3 of 3: generation and proof of VCs ...
analyzing Swap_It.Swap, 2 checks
swap_it.ads:7:22: postcondition not proved
analyzing precondition for Swap_It.Swap, 0 checks
```

Postcondition not proved!

Swap Seven: All is well, Finally!

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

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

Confirmed by `gnatprove`: This is correct!

4.3: Example: Math

for a moment, remove `-gnato13` from the switches

```
project Adder is
  for Source_Dirs use (".");

  package Compiler is
    for Default_Switches ("Ada") use ("-gnat12"
                                      --, "-gnato13");
  end Compiler;

  package Prove is
   for Switches use
   ("--report=fail",
    "--proof=progressive",
    "--warnings=continue",
    "--timeout=5");
  end Prove;

end P;
```

Specification and Implementation

```
pragma Spark_Mode (On);
package Add is

   function Sum(X: Integer; Y: Integer) return Integer with
      Pre  => X + Y in Integer,
      Post => Sum'Result = X + Y;

end Add;
```

```
pragma Spark_Mode (On);
package body Add is

   function Sum(X: Integer; Y: Integer) return Integer is
   begin
      return X + Y;
   end Sum;

end Add;
```

Flow Check `gnatprove`

```
> gnatprove -Padder --mode=flow
Phase 1 of 2: frame condition computation ...
Phase 2 of 2: translation to intermediate language ...
```

No surprise: No errors!

Proving with gnatprove

```
> gnatprove -Padder --mode=prove
...
add.ads:5:17: warning: overflow check might fail
```

- gnatprove could not disprove an overflow
- but the problem is in the precondition, not in the implementation

```
pragma Spark_Mode (On);
package Add is

   function Sum(X: Integer; Y: Integer) return Integer with

      Pre  => X + Y in Integer,

      Post => Sum'Result = X + Y;

end Add;
```

The Source of the Problem

```
function Sum(X: Integer; Y: Integer) return Integer with
   Pre  => X + Y in Integer,
   Post => Sum'Result = X + Y;
```

Catch 22: Before we could possibly check "`X+Y in Integer`", we'd have to add X and Y, and if they are not in Integer, this would overflow.

Possible fix: Rewrite the "offending" precondition:

```
function Sum(X: Integer; Y: Integer) return Integer with
   Pre  => (if X > 0 then
               Integer'Last-X >= Y
            else
               Integer'First-X <= Y);
   Post => Sum'Result = X + Y;
```

A Different Fix

. . . without rewriting the precondition

`-gnato13`:

Tell `gnat` and `gnatprove` to assume **unbounded mathematical integers** in assertions (including pre- and postconditions).

```
for Default_Switches ("Ada") use ("-gnat12", "-gnato13");
```

```
> gnatprove -Padder
Phase 1 of 3: frame condition computation ...
Phase 2 of 3: analysis and translation to intermediate
Phase 3 of 3: generation and proof of VCs ...
analyzing Add, 0 checks
analyzing Add.Sum, 2 checks
```

Spec of a Simple Math Package

```
pragma Spark_Mode(On);
package Math is

   function SQRT(X: Natural) return Natural with
     Contract_Cases =>
       ( X <= 1 => SQRT'Result = X,
         X > 1  => SQRT'Result > 1 and then
                   X/SQRT'Result <= SQRT'Result and then
                   X/(SQRT'Result-1) > (SQRT'Result-1)    );

   function Log_2(X: Positive) return Natural with
     Post => 2**Log_2'Result <= X and then
             2**Log_2'Result > X/2);

end Math;
```

Observe that the postcondition of SQRT is written as a complete set of disjoint Contract_Cases.

Implementation of SQRT

```
function SQRT(X: Natural) return Natural is
   R: Positive;
begin
   if X <= 2 then
      return X;
   else
      R := 2;
      while X/R > R loop
         pragma Loop_Variant(Increases => R);
         R := R + 1;
      end loop;
      return R;
   end if;
end SQRT;
```

Implementation of SQRT

```
function SQRT(X: Natural) return Natural is
   R: Positive;
begin
   if X <= 2 then
      return X;
   else
      R := 2;
      while X/R > R loop
         pragma Loop_Variant(Increases => R);
         R := R + 1;
      end loop;
      return R;
   end if;
end SQRT;
```

- To prove program termination, we did provied a **loop variant**.
- In the case of SQRT, we habe been exceptionally lucky: SPARK did automatically find a **loop invariant**, with which it managed to prove the postcontition.
- Most of the time, we must supply the **loop invariant** manually.
- To understand more about choosing **loop variant**s, **loop invariant**s, and how to get the proof done, we will have to introduce some theory.