SPARK 2014: Flow Analysis

Claire Dross and Martyn Pike

University.adacore.com

Copyright © AdaCore

Flow Analysis – What does it do?

- **Models the variables used by a subprogram**
  - Global and scope variables
  - Local variables
  - Formal parameters

- **Models how information flows through the statements in the subprogram**
  - from initial values of variables
  - to final values of variables

Copyright © AdaCore

Flow Analysis – Errors Detected – Uninitialized Variables

- **Use of uninitialized variables**
  - GNATprove requires initialization of variables prior to them being read
  - Flow analysis will report every violation of this requirement on SPARK code

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

Copyright © AdaCore

Flow Analysis – Errors Detected – Ineffective Statements

- **Warning on ineffective statements and unused variables**
  - An ineffective statement has no effect on any output variable
  - The presence of ineffective statements and unused variables reduces the quality and the maintainability of the code
  - They often indicate errors in the code

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

Tmp : T;

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

Copyright © AdaCore

Flow Analysis – Errors Detected – Incorrect Parameter Mode

- **Detection of incorrect mode of parameters (in, out, or in out)**

Initial value read	Updated on some path	Updated on every path	Parameter mode
X			**in**
X	X	or X	**in out**
	X		**in out**
		X	**out**

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

Copyright © AdaCore

Flow Analysis – Additional Verifications – Global Contracts

- **Global contracts state the global variables accessed or modified by a subprogram**
  - Variables are global to a subprogram if they are defined outside of its scope (at library level or in enclosing units for nested subprograms)
- **They are checked by flow analysis when present**
  - Flow analysis makes sure they are complete (no global variable missing) and correct

```
X : Natural := 0;

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

Copyright © AdaCore

Flow Analysis – Additional Verifications – Global Contracts

- **Global contracts are part of the specifications**
  - Like parameters, they have a mode. It can be Input, Ouput, In_Out, or Proof_In (for global variables only referenced in assertions)
  - Default mode is Input
  - ‘null’ can be used to specify that no global variable is referenced

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

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

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

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

Copyright © AdaCore

Flow Analysis – Additional Verifications – Depends Contracts

- **Depends contracts announce dependencies between outputs and inputs of subprograms**
  - Takes into account both parameters and global variables
  - Useful in particular for checking security properties
- **Flow analysis checks bodies of subprograms against Depends contracts when specified**
  - When a Depends contract is specified for a subprogram, it should be complete (relate every output to all its inputs) and correct.

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

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

Copyright © AdaCore

Flow Analysis – Additional Verifications – Depends Contracts

- **Depends contracts are part of the specifications**
  - ‘+’ indicates a dependency of a variable on its own initial value
  - ‘null’ can be used to state that an output depends on no input
  - or that an input does not affect any output

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

function Get_Value_Of_X return Natural with
  Depends => (Get_Value_Of_X’Result => X);    -- result depends on X

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

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

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

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

Copyright © AdaCore

Flow Analysis – Shortcomings – Modularity

- **Flow analysis, and in particular detection of uninitialized variables, is done modularly on a per subprogram basis**
  - global and parameter inputs should be initialized prior to any subprogram call.
  - global and parameter outputs should be initialized prior to subprogram return

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

Copyright © AdaCore

Flow Analysis – Shortcomings – Composite Types

- **Flow analysis treats array objects as single, entire objects**
  - Changing one element of an array object preserves the values of the other elements
  - In general, there is no way for flow analysis to determine whether a sequence of assignments to an array has updated all the elements of the array or a subset of them
  - So initializing an array with a sequence of statements will result in a flow message

```
for I in A’Range loop
   A (I) := 0;
end loop;
-- flow analysis does not know that A is initialized

A := (others => 0);
-- flow analysis knows that A is initialized
```

Copyright © AdaCore

Flow Analysis – Shortcomings – Composite Types

- **Flow analysis tracks record fields separately inside a subprogram**
  - Initialization and dependencies are treated in a more fine-grained manner
- **Record variables are treated as entire variables when taken as input and output of subprograms**

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

R : Rec;

R.F1 := 0;
R.F2 := 0;
--  R is initialized
```

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

R.F1 := 0;
Init_F2 (R);
-- R should be initialized
-- before this call
```

Copyright © AdaCore

Flow Analysis – Shortcomings – Value Dependency

- **Flow analysis is not value dependent**
  - It only reasons in terms of control flow

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

-- Flow analysis does not
-- know that R is initialized
```

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


-- Flow analysis knows that R
-- is initialized
```

Copyright © AdaCore

Flow Analysis – Shortcomings – Contract Computation

- **When not specified for a subprogram, Global and Depends contracts are computed**
  - Computed Global contracts are used to check initialization of variables
  - Computed contracts of callees are used to check user-written contracts of callers
- **Sometimes, computed contracts are not precise enough**
  - Global variable may have mode In_Out instead of Ouput
  - Depends contracts always assume that all outputs depend on all inputs.

Copyright © AdaCore

Quiz

Copyright © AdaCore

? Is this correct? 1/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

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

Copyright © AdaCore

Is this correct? 1/10

NO

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

Though there clearly are legal uses of the function Search_Array, flow analysis will complain here that Result is not initialized on every path.

It is up to the user then to make sure that Result's value will not be read when Found is false.

Copyright © AdaCore

Is this correct? 2/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

```
Not_Found : exception;

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

Copyright © AdaCore

? Is this correct? 2/10

✓ YES

```
Not_Found : exception;

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

Even if the out parameter Result of Search_Array is not initialized on the path where the exception is raised, flow analysis won't emit any message here. On the other hand, GNATprove will attempt to show that this exception cannot be raised at runtime, that is, that Search_Array is always called on an array A containing the value E.

Copyright © AdaCore

Is this correct? 3/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

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

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

Copyright © AdaCore

Is this correct? 3/10

✓ YES

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

procedure Search_Array (A      :     Array_Of_Positives;
                        E      :     Positive;
                        Result : out Search_Result) is
begin
  for I in A’Range loop
    if A (I) = E then
      Result := (Found   => True,
                 Content => I);
      return;
    end if;
  end loop;
  Result := (Found => False);
end Search_Array;
```

Here flow analysis can make sure that the appropriate record components are initialized depending on the value of Found.

Copyright © AdaCore

? Is this correct? 4/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

```
function Size_Of_Biggest_Increasing_Sequence return Natural is
  Max         : Natural;
  End_Of_Seq  : Boolean;
  Size_Of_Seq : Natural;
  Beginning   : Integer;
  procedure Test_Index (Current_Index : Integer) is
  begin
    if A (Current_Index) >= Max then
      Max := A (Current_Index);
      End_Of_Seq := False;
    else
      Max := 0;
      End_Of_Seq := True;
      Size_Of_Seq := Current_Index - Beginning;
      Beginning := Current_Index;
    end if;
  end Test_Index;
begin
  for I in A’Range loop
    Test_Index (I);
    ...
```

Copyright © AdaCore

Is this correct? 4/10

NO

```
function Size_Of_Biggest_Increasing_Sequence return Natural is
  Max          : Natural;
  End_Of_Seq   : Boolean;
  Size_Of_Seq  : Natural;
  Beginning    : Integer;
  procedure Test_Index (Current_Index : Integer) is
  begin
    if A (Current_Index) >= Max then
      Max := A (Current_Index);
      End_Of_Seq := False;
    else
      Max := 0;
      End_Of_Seq := True;
      Size_Of_Seq := Current_Index - Beginning;
      Beginning := Current_Index;
    end if;
  end Test_Index;
begin
  for I in A'Range loop
    Test_Index (I);
    ...
```

Max and Beginning should be initialized before the call, as they are read in Test_Index. Flow analysis will also report non initialization of Size_Of_Seq, as it may still be uninitialized after the procedure return.

Copyright © AdaCore

Is this correct? 5/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

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

procedure Init (A : out Permutation) is
begin
   for I in A'Range loop
      A (I) := I;
   end loop;
end Init;

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

Copyright © AdaCore

Is this correct? 5/10

✓ YES

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

procedure Init (A : out Permutation) is
begin
  for I in A'Range loop
    A (I) := I;
  end loop;
end Init;

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

This program is correct. Flow analysis will not be able to verify initialization of A in Init though, as it is done in a loop.

Copyright © AdaCore

Is this correct? 6/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

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

procedure Init (A : in out Permutation) is
begin
   for I in A'Range loop
      A (I) := I;
   end loop;
end Init;

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

Copyright © AdaCore

Is this correct? 6/10

NO

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

procedure Init (A : in out Permutation) is
begin
  for I in A'Range loop
    A (I) := I;
  end loop;
end Init;

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

It may be tempting to change the parameter mode of the A argument of Init to in out to avoid the previous flow analysis message. It should not be done though. Otherwise, the A argument of Init should be initialized before every call…

Copyright © AdaCore

Is this correct? 7/10

✓ YES
(click on the check icon)

✗ NO
(click on the error location(s))

```
Increment : constant Natural := 10;

procedure Incr_Step_Function (A : in out Array_Of_Positives) is
  Threshold : Positive := Positive’Last;
  procedure Incr_Until_Threshold (I : Integer) with
    Global => (Input  => Threshold,
               In_Out => A);

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

begin
  for I in A’Range loop
    ...
    Incr_Until_Threshold (I);
  end loop;
end Incr_Step_Function;
```

Copyright © AdaCore

Is this correct? 7/10

✓ YES

```
Increment : constant Natural := 10;

procedure Incr_Step_Function (A : in out Array_Of_Positives) is
  Threshold : Positive := Positive’Last;
  procedure Incr_Until_Threshold (I : Integer) with
    Global => (Input  => Threshold,
               In_Out => A);

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

begin
  for I in A’Range loop
    ...
    Incr_Until_Threshold (I);
  end loop;
end Incr_Step_Function;
```

A and Threshold are global to Incr_Until_Threshold. Increment is a constant so it should not be mentioned in Global contracts

Copyright © AdaCore

? Is this correct? 8/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

```
Max         : Natural := 0;
End_Of_Seq  : Boolean;
Size_Of_Seq : Natural := 0;
Beginning   : Integer := A’First - 1;
procedure Test_Index (Current_Index : Integer) with
  Global => (In_Out => (Beginning, Max, Size_Of_Seq),
             Output => End_Of_Seq,
             Input  => Current_Index);

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

Copyright © AdaCore

Is this correct? 8/10

NO

```
Max         : Natural := 0;
End_Of_Seq  : Boolean;
Size_Of_Seq : Natural := 0;
Beginning   : Integer := A’First - 1;
procedure Test_Index (Current_Index : Integer) with
  Global => (In_Out => (Beginning, Max, Size_Of_Seq),
             Output => End_Of_Seq,
             Input  => Current_Index);

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

Current_Index is a parameter of Test_Index, it should not be referenced as a global variable. On the other hand, A should appear as an Input in the Global contract.

Copyright © AdaCore

Is this correct? 9/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

```
Max          : Natural := 0;
End_Of_Seq   : Boolean;
Size_Of_Seq  : Natural := 0;
Beginning    : Integer := A'First - 1;
procedure Test_Index (Current_Index : Integer) with
  Depends => ((Max, End_Of_Seq)        => (A, Current_Index, Max),
              (Size_Of_Seq, Beginning) =>
                              +(A, Current_Index, Max, Beginning))

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

Copyright © AdaCore

? Is this correct? 9/10

✓ YES

```
Max          : Natural := 0;
End_Of_Seq   : Boolean;
Size_Of_Seq  : Natural := 0;
Beginning    : Integer := A'First - 1;
procedure Test_Index (Current_Index : Integer) with
  Depends => ((Max, End_Of_Seq)        => (A, Current_Index, Max),
              (Size_Of_Seq, Beginning) =>
                            +(A, Current_Index, Max, Beginning))

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

Every output depends on A, Current_Index and Max as they appear in the condition of the if statement. Since Size_Of_Seq and Beginning may not be modified, they have an additional self dependency

Copyright © AdaCore

? Is this correct? 10/10

✓ **YES**
(click on the check icon)

✗ **NO**
(click on the error location(s))

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

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

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

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

Copyright © AdaCore

Is this correct? 10/10

✓ YES

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

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

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

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

This code is correct, but flow analysis cannot verify the Depends contract of Identity. Indeed, Swap has no user-specified Depends contract. As a consequence, flow analysis assumes that all outputs of Swap, that is X and Y, depend on all its inputs, that is both X and Y's initial values.

Copyright © AdaCore

cc BY SA

university.adacore.com

Copyright © AdaCore