SPARK verification features Continued

Paul Jackson

School of Informatics
University of Edinburgh

Formal Verification
Spring 2018

Executability of assertions

Virtually all SPARK assertions are executable.

Are issues with quantifiers:

- Each `for all` or `for some` quantifier is translated into a loop over the values in the range quantified over
- When ranges are finite, loops terminate
  - Ranges finite nearly always
  - An issue with `Universal_Integer` type, implemented with a *BigNum* package.

Executability makes run-time assertion checking feasible

- Compilers have flags to optionally add checking to object code
- Care needed because of possible performance issues

Ghost code

Ghost code is extra code added to SPARK programs that is only used for specification purposes.

Never affects normal function of programs

- SPARK language provides syntax identifying ghost code. SPARK tools check that normal code never uses ghost code

Does impact performance when run-time assertion checking enabled

Ghost variables

Using a ghost variable to capture the initial value of a parameter.

```
procedure Do_Something (X : in out T) is
  X_Init : constant T := X with Ghost;
begin
  Do_Some_Complex_Stuff (X);
  pragma Assert (Is_Correct (X_Init, X));
  --  It is OK to use X_Init inside an assertion.

  X := X_Init;
  --  Compilation error:
  --     Ghost entity cannot appear in this context
```

Ghost functions and procedures

Uses include

- Factoring out common expressions in contracts
- Abstracting state

```
type Queue is private;

function Get_Model (S : Queue) return Nat_Array with Ghost;
--  Returns an array as a model of a queue

procedure Push_Front (S : in out Queue; E : in Natural) with
  Pre  => Get_Model (S)'Length < Max,
  Post => Get_Model (S) = E & Get_Model (S)'Old;

procedure Pop_Back (S : in out Queue; E : out Natural) with
  Pre  => Get_Model (S)'Length > 0,
  Post => Get_Model (S) & E = Get_Model (S)'Old;
```

Verification case study 1

Verification of Selection sort

- Shows where SPARK verification starts needing major user guidance

```
package Sort with SPARK_Mode is

   -- Sorts the elements in the array Values in ascending order
   procedure Selection_Sort (Values : in out Nat_Array)
     with
       Post => Is_Perm (Values'Old, Values) and then
     (if Values'Length > 0 then
        (for all I in Values'First .. Values'Last - 1 =>
             Values (I) <= Values (I + 1)));
end Sort;
```

Verification case study 2

Definition of Is_Perm function

```
package Perm with SPARK_Mode, Ghost is
   subtype Nb_Occ is Integer range 0 .. 100;

   function Remove_Last (A : Nat_Array) return Nat_Array is
     (A (A'First .. A'Last - 1))
   with Pre  => A'Length > 0;

   function Occ (A : Nat_Array; E : Natural) return Nb_Occ is
     (if A'Length = 0 then 0
      elsif A (A'Last) = E then Occ (Remove_Last (A), E) + 1
      else Occ (Remove_Last (A), E))
   with
     Post => Occ'Result <= A'Length;

   function Is_Perm (A, B : Nat_Array) return Boolean is
     (for all E in Natural => Occ (A, E) = Occ (B, E));

end Perm;
```

Verification case study 3

```
procedure Selection_Sort (A : in out Nat_Array) is
  Smallest : Positive;
begin
  if A'Length = 0 then return; end if;

  for K in A'First .. A'Last - 1 loop
    Smallest := Index_Of_Minimum (A (K .. A'Last));

    if Smallest /= K then
      Swap (Values => A, X => K, Y => Smallest);
    end if;

    pragma Loop_Invariant
      (for all I in A'First .. K =>
        (for all J in I + 1 .. A'Last =>
            A (I) <= A (J)));
    pragma Loop_Invariant (Is_Perm (A'Loop_Entry, A));
  end loop;

end Selection_Sort;
```

Verification case study 4

Full info in GNAT*prove by Example* section of SPARK UG

- Definition of `Index_Of_Minimum` function
- Swap contract

```
procedure Swap (Values : in out Nat_Array;
                X      : in     Positive;
                Y      : in     Positive)
with
   Pre  => (X in Values'Range and then
            Y in Values'Range and then
            X /= Y),

   Post => Is_Perm (Values'Old, Values)
      and Values (X) = Values'Old (Y)
      and Values (Y) = Values'Old (X)
      and (for all Z in Values'Range =>
             (if Z /= X and Z /= Y
              then Values (Z) = Values'Old (Z)))
```

- Justification for Swap realising its specification
  - Pragma assertions provide hints to prover
  - Ghost loop helps establish `Is_Perm (Values'Old, Values)`