Grupo:
44
18/10/19

Práctica 1: Verificación Formal con SPARK 2014

PROGRAMACIÓN III

AMUSQUÍVAR POPPE, JOSÉ MARÍA
JESWANI TEJWANI, PRASHANT
MARÍN REYES, IGNACIO JOSÉ

Programación III
Informe de la Práctica 1 (v4)
Verificación Formal con SPARK 2014

1. Nombre y apellido de los miembros del equipo.

   Ignacio José Marín Reyes
   Prashant Jeswani Tejwani
   José María Amusquívar Poppe

2. Horario de laboratorio en que van a defender el trabajo realizado (por ejemplo, Martes 12:30-14:30, Miércoles 10:30-12:30, etc.)

   La defensa del trabajo se realizará en la hora de prácticas del grupo 44, grupo al que pertenecemos (Jueves 10:30-12:30).

3. Listado enumerado con el nombre y tipo (procedimiento/función) de los procedimientos y funciones verificados.

   1. Function alphaValue
   2. Function divVector
   3. Function randomVector
   4. Function reverseString
   5. Procedure scaleChanger
   6. Procedure sumVector

4. Tabla que muestra qué características de SPARK se han utilizado para verificar formalmente cada uno de los procedimientos y funciones.

   El número de cada columna se corresponde con el número asignado a los métodos en el apartado (2).

   En cada casilla solamente hay que marcar con 'X' (en el centro de la casilla) si la verificación de este método utiliza la característica de SPARK indicada en el margen derecho de esa fila y su valor no es NULL ni True.

   El uso de Contract_Cases es opcional.

	1	2	3	4	5	6
Global						X
Depends	X	X	X	X	X	X
Pre	X	X	X	X	X	X
Post	X	X	X	X	X	X
Contract_Cases						
'Result	X	X	X	X		
'Old					X	

	1	2	3	4	5	6
for all	X	X	X	X		X
for some						X

	1	2	3	4	5	6
Loop_Variant		X				
Loop_Invariant	X	X	X	X		X
‘Loop_Entry			X			

5. Número de tests unitarios hechos para comprobar cada procedimiento y función. De nuevo, el número de cada columna se corresponde con el número asignado a los métodos en el apartado (2).

	1	2	3	4	5	6
Número de tests	10	10	10	10	10	10

6. Cabecera completa (con su contrato) de cada uno de los procedimientos y funciones verificados formalmente (incluyendo el comentario que describe su comportamiento, y manteniendo la numeración del apartado (2)).

1. Function alphaValue

```
type T_String is array (Positive range <>) of Character;
type T_Array is array (Positive range <>) of Natural;

function alphaValue (InStr : T_String) return T_Array
--Function that receives a string as input parameter, and it returns a vector of natural
--numbers filled with the positions of each character included in the alphabet (a=1..z=26).
--If the character is not included in the alphabet, the position is filled with 0.

with
   Global      => null,
   Depends     => (alphaValue'Result => InStr),
   Pre         => InStr'Length >= 1 and (for all x in InStr'Range =>
                                          (Character'Pos(InStr(x)) in 0..255)),
   Post        => (alphaValue'Result'Length = InStr'Length and then(
                     for all i in alphaValue'Result'Range =>
                                 (if Character'Pos(InStr(i)) in 97..122 then
                                       alphaValue'Result(i) = Character'Pos(InStr(i)) - 96
                                 elsif Character'Pos(InStr(i)) in 65..90 then
                                       alphaValue'Result(i) = Character'Pos(InStr(i)) - 64
                                 else
                                       alphaValue'Result(i) = 0)));
```

2. Function divVector

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

function divVector (vectorInicial : T_Vector; n : Positive) return T_Vector
--Function that receives two input parameters: an array of natural numbers and a positive
--number. Function returns an array of natural numbers of the same length as the input
--array. It calculates the module of each position and the positive number passed as a
--parameter, if the module is 0 then that value is copied to the same position of the array to
--be returned, if it is not 0, a zero is copied.

  with
    Global  => null,
    Depends => (divVector'Result => (vectorInicial, n)),
    Pre     => (vectorInicial'Length > 0),
    Post    => (for all I in divVector'Result'Range =>
                                  (if vectorInicial(I) mod n = 0 then
                                        divVector'Result(I) = vectorInicial(I)
                                  else
                                        divVector'Result(I) = 0));
```

3. Function randomVector

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

function randomVector (In1 : Positive; In2 : Positive) return T_Vector
--Function that receives two positive numbers as input parameters (In1, In2), and it
--returns an array of positive numbers.
--Function creates an array whose length is the module of In1 and In2, and which is
--initialized to 1. Then, the multiplication of In1 and the position of the array is calculated,
--then this value is added to the value was stored in that position.

with
   Global  => null,
   Depends => (randomVector'Result => (In1, In2)),
   Pre     => (if In1 mod In2 /= 0 then
                        In1 <= Positive'Last / (In1 mod In2) and then
                  (for all i in 1..(In1 mod In2) => (In1*i <= Positive'Last - 1))),
   Post    => (for all I in randomVector'Result'Range =>
                                 randomVector'Result(I) = 1 + (In1 * I));
```

4. Function reverseString

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

function reverseString (InStr : T_String) return T_String
--Function that receives a string as input parameter, and it returns another
--string that is an inverted copy of the input string.

with
   Global  => null,
   Depends => (reverseString'Result => InStr),
   Pre     => InStr'Length > 0,
   Post    => (for all J in 0 .. InStr'Length -1 =>
                  reverseString'Result(reverseString'Result'First + J) = InStr(InStr'Last - J));
```

5. Procedure scaleChanger

```
procedure scaleChanger (InTemp : in out Integer; InScale : in Character)
  --Procedure that receives one input parameter: origin scale as an character. And it receives
  --one input output parameter: temperature origin as an integer.
  --Procedure transforms temperature from Celsius to Fahrenheit or vice versa. If the input
  --character is 'Cc' or 'Ff', the output temperature scale is the opposite. If another character
  --is read, the transformation is not performed.

    with
      Global   => null,
      Depends => (InTemp => (InTemp, InScale)),
      Pre        => (if (InScale = 'c' or InScale = 'C') and then InTemp >= 0 then
                          InTemp <= (Integer'Last/2) - 32
                     elsif (InScale = 'c' or InScale = 'C') and then InTemp < 0 then
                          InTemp >= (Integer'First/2)
                     elsif (InTemp < 0) then
                          InTemp >= Integer'First + 32),
      Post       => (if InScale = 'c' or InScale = 'C' then
                          InTemp = InTemp'Old * 2 + 32
                     elsif InScale = 'f' or InScale = 'F' then
                          InTemp = (InTemp'Old - 32) / 2
                     else
                          InTemp = Intemp'Old);
```

6. Procedure sumVector

```
    type T_Vector is array (Positive range <>) of Natural;
    max : Natural;
    min : Natural;

    procedure sumVector (v : in T_Vector; sum : out Natural)
    --Procedure that receives one input parameter: an array of natural numbers. And
    --it receives one output parameter: a natural number where the average is stored.
    --Procedure calculates the maxmimum and the minimum values of the array, then the
    --average of these two values is calculated, adding them and dividing said sum by two.

    with
      Global   => (Output => (max, min)),
      Depends => (sum => v, min => v, max => v),
      Pre        => (v'Length > 0),
      Post       => (if (max <= Natural'Last - min) then
                          sum = (max+min) / 2
                     else
                          sum = Natural'Last);
```

7. Cuerpo de cada uno de los procedimientos y funciones verificados formalmente.

1. Function alphaValue

```
function alphaValue (InStr : T_String) return T_Array is
    OutArray : T_Array(InStr'Range) := (others => 0);
begin

      for I in InStr'Range loop

         if Character'Pos(InStr(I)) in 65..90 then
            OutArray (I) := Character'Pos(InStr(I)) - 64;
         elsif Character'Pos(InStr(I)) in 97..122 then
            OutArray (I) := Character'Pos(InStr(I)) - 96;
         end if;

         pragma Loop_Invariant (for all j in OutArray'First..I =>
                                (if Character'Pos(InStr(j)) in 97..122 then
                                    OutArray(j) = Character'Pos(InStr(j)) - 96
                                 elsif Character'Pos(InStr(j)) in 65..90 then
                                    OutArray(j) = Character'Pos(InStr(j)) - 64
                                 else
                                    OutArray(j) = 0));
      end loop;
   return (OutArray);

end alphaValue;
```

2. Function divVector

```
function divVector (vectorInicial : T_Vector; n : Positive) return T_Vector is
   vectorFinal : T_Vector(vectorInicial'First..vectorInicial'Last) := (others => 0);
   I : Natural := vectorInicial'First;
begin

   while I < vectorInicial'Last loop

      if (vectorInicial(I) mod n) = 0 then
          vectorFinal(I) := vectorInicial(I);
      else
          vectorFinal(I) := 0;
      end if;

      pragma Loop_Variant(Increases => I);
      pragma Loop_Invariant(I in vectorInicial'First..vectorInicial'Last - 1);
      pragma Loop_Invariant(
            for all J in vectorInicial'First..I =>
              (if vectorInicial(J) mod n = 0 then
                   vectorFinal(J) = vectorInicial(J)
               else
                   vectorFinal(J) = 0));
      I := I + 1;
   end loop;
```

```
  if vectorInicial(vectorInicial'Last) mod n = 0 then
       vectorFinal(vectorFinal'Last) := vectorInicial(vectorInicial'Last);
   else
       vectorFinal(vectorFinal'Last) := 0;
   end if;

   return (vectorFinal);

end divVector;
```

3. Function randomVector

```
function randomVector (In1 : Positive; In2 : Positive) return T_Vector is
   vector : T_Vector(1..(In1 mod In2)) := (others => 1);
begin

   for I in vector'Range loop

      vector(I) := vector(I) + In1 * I;
      pragma Loop_Invariant (for all J in vector'First.. I =>
                              (vector(J) = vector'Loop_Entry(J) + (In1 * J)));

   end loop;
   return (vector);

end randomVector;
```

4. Function reverseString

```
function reverseString (InStr : T_String) return T_String is
   OutStr : T_String := InStr;
begin
   for I in 0 .. InStr'Length -1 loop

      OutStr (OutStr'First + I) :=  InStr(InStr'Last  - I);
      pragma Loop_Invariant (for all J in 0 .. I =>
                         OutStr(OutStr'First + J) = InStr(InStr'Last - J));
   end loop;
   return (OutStr);

end reverseString;
```

5. Procedure scaleChanger

```
procedure scaleChanger (InTemp : in out Integer; InScale : in Character) is
begin
```

```
        if InScale = 'C' or  InScale = 'c' then
              InTemp := (InTemp * 2) + 32;
        elsif InScale = 'f' or InScale = 'F' then
              InTemp := (InTemp - 32) / 2;
        end if;

end scaleChanger;
```

6. Procedure sumVector

```
procedure sumVector (v : in T_Vector; sum : out Natural) is
begin
  max := v (v'First);
  min := v (v'First);

  for i in v'Range loop

    if v(i) > max then
        max := v(i);
    elsif v(i) < min then
        min := v(i);
    end if;

    pragma Loop_Invariant (for all K in v'First..i => v(K) <= max);
    pragma Loop_Invariant (for some J in v'First..i => max = v(J));

    pragma Loop_Invariant (for all K in v'First..i => v(K) >= min);
    pragma Loop_Invariant (for some J in v'First..i => min = v(J));

  end loop;

  if (max <= Natural'Last - min) then
      sum := (max+min) / 2;
  else
      sum := Natural'Last;
  end if;

end sumVector;
```

Información adicional

8. (Opcional) Aclaraciones que consideren oportunas sobre cada uno de los ejercicios verificados formalmente.

Función o procedimiento	Nivel de verificación
AlphaValue	1
DivVector	1
RandomVector	0
ReverseString	1
ScaleChanger	0
SumVector	1