SPARK Functional Contracts
Objectives

• Understand the motivation for SPARK’s functional contract language (*proof contexts*)

• Be able to use SPARK’s annotations for
  – specifying pre- and post-conditions
  – specifying ranges
  – specifying universal and existential quantification

• Be aware of potential pitfalls associated with specifying functional contracts
Outline

• Simple pre- and post-assertions
• Simple assertions
• Contracts for SPARK functions
• Quantification and ranges
• Update notation for arrays and records
• Assertions
Procedure Contracts

- **Post-condition** describes functional properties of outputs & guarantees made to clients
  - **Pre-state** – state just before the procedure begins execution
  - **Post-state** – state just after the procedure completes execution

```plaintext
procedure Add(X: in Integer);
with Global => (In_Out Total),
Post => Total = Total'Old + X;
```

- Final value of Total equals initial value plus X (value of X cannot be changed in procedure)
- Total’Old indicates value of Total in the pre-state (before execution)
- Modes indicate that X and Total both have values in the pre-state; Total also has a value in the post-state
Procedure Contracts

• **Pre-conditions** state requirements/assumptions on clients

```plaintext
procedure Add(X: in Integer)
    with Global => (In_Out Total),
    Pre => X > 0,
    Post => Total = Total'Old + X;
```

*Constraints on X that the client must satisfy before calling procedure Add, and thus that Add may assume about X at the start of execution.*
Procedure Contracts

- **Contract cases** – case consists of a guard and a corresponding property the procedure guarantees clients.

```pascal
procedure Add_Threshold (X: in Integer)
with Global => (In_Out => Total,
               Input => Threshold),
Contract_Cases =>
  (X + Total <= Threshold => Total = Total'Old + X,
   X + Total > Threshold => Total = Threshold);
```

- Guards refer to values in the pre-state.
- Guards must be disjoint.
- Guards must be complete (cover all legal inputs).

**Guarantee made to clients:** Value of Total in the post-state depends on which guard is true in the pre-state.
Location of pre/post assertions

Usually:

- If a procedure/function is public, its pre-/post-assertion is stated in the package spec.
- If a procedure/function is private, its pre-/post-assertion is stated in the package body

```plaintext
package Adder
   with SPARK_Mode
is
   Total : Integer := 0;
   Threshold: constant Integer := 100000;
   procedure Add_THRESHOLD (X: in Integer)
      with Global => (In_Out => Total),
      Contract_Cases =>
      (X + Total <= Threshold
      => Total = Total'Old + X,
      X + Total > Threshold
      => Total = Threshold);
end Adder;
```

```plaintext
package body Adder
   with SPARK_Mode
is
   procedure Add (X: in Integer)
      with Global => (In_Out => Total),
      Pre => X > 0,
      Post => Total = Total'Old + X
   is begin
      Total := Total + X;
   end Add;

   procedure Add_THRESHOLD (X: in Integer)
   is begin
      if Total + X <= Threshold then
         Add(X);
      else
         Total := Threshold;
      end if;
   end Add_THRESHOLD;
end Adder;
```
Procedure Contracts – Pitfalls for proof

- adder.adb:10:22: warning: overflow check might fail
- adder.adb:15:16: warning: overflow check might fail
- adder.adb:16:10: warning: precondition might fail
Procedure Contracts – Pitfalls for proof

- adder.adb:10:22: warning: overflow check might fail
- adder.adb:15:16: warning: overflow check might fail
- adder.adb:16:10: info: precondition proved
Procedure Contracts – Proof of correctness

```ada
package Adder
with SPARK_Mode
is
  Total : Integer := 0;
  Threshold: constant Integer := 100000;

procedure AddThreshold (X: in Integer)
  with Global => (In_Out => Total),
  Pre => (X > 0 and then
          Total <= Integer'Last - X),
  Contract_Cases =>
    (X + Total <= Threshold
     => Total = Total'Old + X,
    X + Total > Threshold
     => Total = Threshold);

end Adder;
```

```ada
package body Adder
with SPARK_Mode
is
  procedure Add (X: in Integer)
    with Global => (In_Out => Total),
    Pre => (X > 0 and then
            Total <= Integer'Last - X),
    Post => Total = Total'Old + X
  is begin
    Total := Total + X;
    end Add;

  procedure AddThreshold (X: in Integer)
    is begin
      if Total + X <= Threshold then
        Add(X);
      end if;
```

Messages | Locations
---|---

```
- gnatprove (8 items)
  - adder.adb (4 items)
    9:17    info: postcondition proved
    11:22   info: overflow check proved
    16:16   info: overflow check proved
    17:10   info: precondition proved
  - adder.ads (4 items)
    13:11   info: disjoint contract cases proved
    13:11   info: complete contract cases proved
    15:17   info: contract case proved
    17:17   info: contract case proved
```
Function Contracts

• Post condition describes what client may assume of return value

• All parameters have (implicit) mode `in`

```haskell
function Max (X, Y : Integer) return Integer with Post =>
    ((X > Y and then Max'Result = X)
     or else
     (X <= Y and then Max'Result = Y));
```

*Describes the result in terms of the input values (which are constants in the body).*
Function Contracts – Pitfall for Proof

Since a “hole” in the post condition would allow any return result!
Function Contract – alternative “fixes”

function Max (X, Y : Integer) return Integer with
    Pre => X /= Y,
    Post =>
        ((X > Y and then Max'Result = X)
        or else
        (X <= Y and then Max'Result = Y));

V.S

function Max (X, Y : Integer) return Integer with Post =>
    ((X > Y and then Max'Result = X)
    or else
    (X <= Y and then Max'Result = Y));
Function Contracts

• Contract_Cases often more succinct than Post condition

```haskell
function Max (X, Y : Integer) return Integer
with Contract_Cases =>
  (X > Y => Max'Result = X,
   X <= Y => Max'Result = Y);
```

V.S

```haskell
function Max (X, Y : Integer) return Integer
with Post =>
  ((X > Y and then Max'Result = X)
   or else
   (X <= Y and then Max'Result = Y));
```
Function Expressions

```plaintext
function Inc(X: Integer) return X + 1;
```

- Implicit post condition:
  \[ \text{Post} \implies \text{Inc\'Result} = X + 1 \]
- May appear in a specification file
- Provides a useful abstraction mechanism for use in pre/post conditions
- Does not require proof
Ranges

• A common feature of SPARK programs

• Consider the declaration:
  - `subtype Index is Integer range 1 .. 100;`

• The range can be reference in a number of ways, e.g.,
  - `I in Index`
  - `I in Index range Index'First .. 50;`
  - `I in Index range 10 .. Index'Last-9;`

• The same notation can be used in contracts
Ranges in Contracts

- Simplifies pre-conditions
- Procedures can assume argument is in the range
Loop Invariant

**procedure** Divide (M, N : in Natural;
     Q, R : out Natural);

**is begin**
    R := M;
    Q := 0;
    **while** R > N **loop**
        **pragma Loop_Invariant** (M = Q * N + R);
        Q := Q + 1;
        R := R - N;
    **end loop**;
**end Divide**;

- Needed for (induction) proof of *partial correctness* (if the procedure terminates, then the post-condition is true in the post-state)
- Must be weak enough to be valid invariant, strong enough to imply post-condition on termination (and no RTEs in the loop)
Loop Variant

Procedure Divide (M, N : in Natural;
               Q, R : out Natural);

Is begin
    R := M; Q := 0;
    while R > N loop
        pragma Loop_Invariant (M = Q * N + R);
        pragma Loop_Variant (Decreases => R);
        Q := Q + 1; R := R - N;
    end loop;
end Divide;

• Permits (induction) proof of termination.
• Variant expressions strictly increase or decrease on each iteration
• If multiple variant expressions: check at each iteration that at least one is progressing in the specified direction (Increases or Decreases)
function Max_Index (A : T_Arr) return Positive with
  Pre  =>  (A'Length > 0),
  Post =>
  ((for all J in A'Range =>
      A (J) <= A (Max_Index'Result)) and
  (for all J in A'First .. Max_Index'Result - 1 =>
      A (J) < A (Max_Index'Result)));
Quantification in Pre/Post Assertion

```plaintext
function Max_Element (A : T_Arr)
  return T
is (A (Max_Index (A)))
with
  Pre  => (A'Length > 0),
  Post =>
    ((for all J in A'Range =>
      Max_Element'Result >= A (J))
    and
    (for some J in A'Range =>
      Max_Element'Result = A (J)));
```

**Universal quantification**

**Existential quantification**
function Max_Index (A : T_Arr) return Positive is
  Result : Positive := A'First;
begin
  for J in A'First .. A'Last loop
    if A (Result) < A (J) then
      Result := J;
    end if;
  pragma Loop_Invariant
    ((for all K in A'First .. J =>
      A (K) <= A (Result)) and
     (for all K in A'First .. Result - 1 =>
      A (K) < A (Result)));
  end loop;
  return Result;
end Max_Index;
Loop Invariant - Pitfall

- max.adb:7:29: range check proved
- max.adb:10:16: array index check might fail
- max.adb:11:23: range check proved
- max.adb:13:10: loop invariant initialization proved
- max.adb:13:10: loop invariant preservation proved
- max.adb:15:23: index check proved
- max.adb:15:32: array index check might fail

Loop Invariant too weak: loses info that Result is a legal array index
Post Condition - Pitfall

- max.adb:7:29: range check proved
- max.ads:17:23: array index check might fail
- max.ads:17:23: precondition proved
- max.ads:21:7: postcondition might fail

Max_Index post is too weak for proof of Max_Element: It needs to return a legal index
package Max with
  Spark_Mode is

  type T is new Integer;

  type T_Arr is array (Positive range <> ) of T;

  function Max_Index (A : T_Arr)
    return Positive
  with
    Pre => (A'Length > 0),
    Post =>
      ((Max_Index'Result in A'Range) and
      (for all J in A'Range =>
        A (J) <= A (Max_Index'Result)) and
      (for all J in A'First .. Max_Index'Result - 1 =>
        A (J) < A (Max_Index'Result)));

  function Max_Element (A : T_Arr)
    return T is (A (Max_Index (A)))
  with
    Pre => (A'Length > 0),
    Post =>
      ((for all J in A'Range =>
      Max_Element'Result >= A (J)) and
      (for some J in A'Range =>
        Max_Element'Result = A (J)));

end Max;

package body Max with
  Spark_Mode is

  function Max_Index (A : T_Arr)
    return Positive
  is
    Result : Positive := A'First;
    begin
      for J in A'First .. A'Last loop
        if A (Result) < A (J) then
          Result := J;
        end if;
        pragma Loop_Invariant
        (Result in A'Range and
          (for all K in A'First .. J =>
            A (K) <= A (Result)) and
          (for all K in A'First .. Result - 1 =>
            A (K) < A (Result)));
      end loop;
    return Result;
  end Max_Index;

end Max;
Update Notation

- Special notation for representing a composite object for which some components are changed.

\[
\text{procedure} \quad \text{Write}(\ I: \text{ in } \text{ Index}; \ V: \text{ in } \text{ Value}) \quad \text{is}
\]

\[
\quad \text{with Global} \Rightarrow (\text{In\_Out} \Rightarrow A), \\
\quad \text{Depends} \Rightarrow (A \Rightarrow A, I, V), \\
\quad \text{Post} \Rightarrow A = A'\text{Old}'\text{Update} \ (I \Rightarrow V);
\]

The array identical to the value of \(A\) in the pre-state \((A'\text{Old})\), except at index \(I\), where it has value \(V\).
type Index is range 1 .. 10;
type ArrType is array (Index) of Integer;

procedure Swap (I, J: in Index;
    A: in out ArrType)

    with Post =>
    A = A'Old'Update (I => A'Old (J),
                     J => A'Old (I));

The post-value of A equals the pre-value of A except at positions I and J; in the post-state, position I maps to the pre-value at position J and J maps to the pre-value at position I.
Update Notation

```haskell
type Point is
  record
    X_Coord, Y_Coord: Float;
  end record;

procedure UpdateX(P : in out Point, NewX: in Float)
with
  Post => P = P'Old'Update (X_Coord => NewX);
```

The post-value of P equals the pre-value except at field X_Coord, which maps to the value of NewX.
Position of Old

- **A’Old (I)**
  - Refers to value at position I (current value) in original value of A
  - Here, A is both imported/exported (not necessarily for I)

- **A’Old (I’Old)**
  - Refers to value at position I (original value) in original value of A
  - Here both A and I are imported/exported

- **R’Old.C**
  - Refers to the value of component C in the initial value of record R

- **P.V’Old (not P’Old.V)**
  - Refers to the initial value of global variable V from package P (P’Old doesn’t make sense because a package does not change state)
Assertions

- Assertions are specified in SPARK using

  \texttt{pragma Assert <bool\_exp>;}\

  \texttt{pragma Assert C \geq 0;}\

- May appear in the body of a procedure or function
- Dynamic check at run-time
- A proof obligation (only)

\textit{Boolean expression that programmer expects to be True}
Assert and Cut

- Like Assert, but also “cuts” the execution
- Replaces all knowledge of execution context with the cut assertion
- Reduces number of paths that need to be analyzed

```plaintext
procedure P is
    X : Integer;
begin
    -- complex computation that sets
    pragma Assert_And_Cut (X > 0);
    -- complex computation that uses X
end P;
```

$X > 0$ should be True AND is only context needed for remaining proof obligations
Acknowledgements

- The material in this lecture adapted from:
  - Jérôme Hugues’ Git Hub repository of Spark Examples