High Integrity Computing (F23PS2)

High Integrity Software Development (F29HD2)

The SPARK Approach (Part 1)

Andrew Ireland

Department of Computer Science
School of Mathematical and Computer Sciences
Heriot-Watt University
Edinburgh

Prelude

"It is not too late! I believe that by careful pruning of the Ada language, it is still possible to select a very powerful subset that would be reliable and efficient in implementation and safe and economic to use."

Professor Tony Hoare,
1980 ACM Turing Award Lecture

Overview

- Basic building blocks of a SPARK program.
- SPARK core annotations.

Basic Program Unit

- The basic building block of a program is the **procedure**.
- A procedure consists of a **declaration** part and a **statement** part:

```
procedure <program_name> (<parameter_list>) is
   <declarations>
begin
   <statements>
end <program_name>;
```

Note the use of `<...>` to denote a place-holder for actual program code.

Hello World

```
with Text_IO;
-- My first program (this is a comment)!
procedure Hello is
begin
   Text_IO.Put_Line("Hello WORLD!");
end Hello;
```

- Note that an empty parameter list **does not** contain brackets, *i.e.* `()`, as is the case with Java, C, C++.
- A program will typically use a existing library resource, *e.g.* such as the `Put_Line` procedure from the `Text_IO` library resource (more details later).
- Such library resources will typically contain many procedures, types, *etc* and therefore require a higher level of structuring ...

Building Larger Program Units

- This higher level of structuring, or abstraction, is provided by **packages**.
- A large number of so called library packages, such as `Text_IO` exist, however, the notion of a package provides a general mechanism for structuring large software systems.
- While procedures support programming-in-the-small, packages can be seen as supporting programming-in-the-large.
- A package has a **specification** and a **body**:
  - A package specification provides an interface, it tells a user **what** resources a package provides.
  - A package body defines **how** the resources are implemented, and is not visible to the user.

Package Specification

```
package <package_name> is
   <declarations>
end <package_name>;
```

- Here the declarations provide enough information to use a resource (subprogram) without revealing its implementation details.
- Note that package specifications and bodies can be defined within the same file, but “best practice” suggests keeping them in separate files, *i.e.* `foo.ads` for the package specification of `foo` and `foo.adb` for the package body of `foo`.

Package Body

```
package body <package_name> is
   <declarations>
end <package_name>;
```

- Here the declarations provide the implementation details of the resource (subprogram).
- Any constants, types, variables declared within the package body are not accessible from outside the package.

Package Body

- An optional statement part can be included within the package body, *i.e.*

```
package body <package_name> is
   <declarations>
begin
   <statements>
end <package_name>;
```

- If included, then the statement part is executed once when the program using the package starts.

Different Roles for Packages

- Packages can be used to group together types and constants.
- Packages can be used to group together logically related subprograms (procedures and functions).
- Packages can have “memory” and can therefore represent objects with state.
- Packages can be used to construct **abstract data types**.

Packages of Types and Constants

```
package Distances is
   subtype Dist is Integer range 1..Integer'Last;
   Edin_Glas: constant:=42;
   Glas_Stir: constant:=30;
   Stir_Edin: constant:=36;
end Distances;
```

Note: `Integer` is a predefined type; `subtype` is a mechanism for defining a subset of a more general concept; `:=` denotes assignment; `constant` declares a constant (more details later).

Packages of Subprograms

```
package Volumes is

   Pi: constant:=3.14159;

   function Box_Car_Vol(L: Float;
                        W: Float;
                        H :Float) return Float;

   function Tank_Car_Vol(L: Float;
                         R: Float) return Float;

end Volumes;
```

Packages of Subprograms

```
package body Volumes is

   function Box_Car_Vol(L: Float;
                        W: Float;
                        H :Float) return Float is
   begin
      return L * W * H;
   end Box_Car_Vol;

   function Tank_Car_Vol(L: Float;
                         R: Float) return Float is
   begin
      return Pi * R * R * L;
   end Tank_Car_Vol;

end Volumes;
```


A Simple Procedure

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

- X and Y are (formal) parameters of type Integer.
- T is a local variable of type Integer.
- The effective of the call Switch(A, B) is to switch the values of the actual parameters A and B.

Formal Parameters To Procedures

in : the value of the **actual** parameter is copied to the **formal** parameter. The **formal** parameter is treated as a constant. Note if not specified then **in** is the default mode.

in out : the **actual** parameter must be a variable that has a value at the time the procedure is called. The value can be used or changed within the procedure. At the end of the procedure call the value of the **formal** parameter is copied back to the **actual** parameter.

out : the **actual** parameter must be a variable. The value of the **actual** parameter at the time the procedure is called is ignored. At the end of the procedure call the value of the **formal** parameter is copied back to the **actual** parameter.

A Simple Function

```
function Int_Min(X, Y: in Integer) return Integer is
   Z: Integer;
begin
   if X > Y then
      Z:= Y;
   else
      Z:= X;
   end if;
   return Z;
end Int_Min;
```

- SPARK functions **can not** have side-effects:
  - function parameters are constrained to have **in** mode
  - modification of global state variables is prohibited (more details later).
- Note that a SPARK function can only have a single `return` statement that must appear as the last statement within the function body.

SPARK Core Annotations

- The annotation language enables the programmer to specify the intended behaviour of their code.
- Annotations should be written before coding begins.
- Annotations provide an alternative perspective on the code.
- The SPARK Examiner automatically checks the consistency between the core annotations and code.
- Once the coding is complete certain aspects of the core annotations will be redundant. However, during the maintenance cycle such redundancy provides an effective basis for consistency checking.

Input/Output Dependency Relations

- Variables that are used to communicate data to a subprogram are known as the **imported variables**.
- Variables that are used to communicate results after a subprogram call are known as the **exported variables**.
- The SPARK `derives` annotation allows the programmer to specify the intended relationship between the imported and exported variables.
- The SPARK Examiner automatically checks for consistency between the derives relationship and the actual code.

The `derives` Clause

```
procedure Switch(X, Y: in out Integer)
--# derives X from Y &
--#         Y from X;
is
    ...
end Switch;
```

- Note that Ada comments are proceeded by `'--'` and SPARK annotations are proceeded by `'--#'`. Because an annotation is a special form of comment its presence does not effect compilation.
- The above derives clauses should be read as:

  The exported value of X is derived from the imported value of Y and the exported value of Y is derived from the imported value of X.

Using Global Variables

- Imported and exported variables are not restricted to being parameters - **global variables** are also allowed.
- In SPARK to access a global variable within a subprogram requires the use of the `global` annotation.
- The SPARK Examiner automatically checks that the use of the `global` annotation is consistent with the code.
- Note that the SPARK Examiner will complain if any variables that are annotated as being `global` do not occur within a `derives` relation.

The `global` Clause

```
procedure Reset(Value: in Integer) is
--# global out Count;
--# derives Count from Value;
begin
   Count:=Value;
end Reset;
```

- The procedure `Reset` updates the global variable `Count` using the formal parameter `Value` – this is expressed via the `global` and `derives` annotations.
- Note that if a procedure appears as part of a package specification then the annotations appear only within the package specification and not the package body.

Notes on `derives` and `global`

- A `global` clause includes mode information, *i.e.* `in` and `out`, as appropriate.
- Multiple global clauses are expressed as follows:

```
--# global <mode> X_1, ..., X_l;
--#        <mode> Y_1, ..., Y_m;
--#         ...
--#        <mode> Z_1, ..., Z_n;
```

- Multiple derives clauses are expressed as follows:

```
--# derives W_1, ..., W_k from X_1, ..., X_l &
--#                     ...
--#         Y_1, ..., Y_m from Z_1, ..., Z_n;
```

- Note that if the value of an exported variable is **not** derived from an imported variable then the semi-colon should directly follow the `from` part of the clause, *e.g.*

```
--# derives X from ;
```

Summary

- Abstraction via packages.
- Package: package specification and package body.
- Subprograms: procedures and functions.
- Formal parameter modes.
- Imported and exported variables.
- Derives and global annotations.