Abstraction and Refinement in SPARK
Data Abstraction

• Important for managing complexity

• Abstract state machine
  – Encapsulates relevant history as a state
  – Provides an abstract view of some data using
    • Operations that cause state transitions
    • Functions that reveal properties of the state
  – Implementation details “hidden” from “clients”
Non Example: Missing data abstraction

package Adder ... is

    Threshold: constant Integer := 100000;
    subtype Amount is Integer range 1 .. Threshold;
    Total : Amount := 0;
    procedure Add_Threshold (X: in Integer);
    procedure Clear;

end Adder;

package body Adder

is

    procedure Add_Threshold (X: in Integer)
    is begin ...
    procedure Clear is begin ...

end Adder;

Not an ADT: Client can import Adder and directly manipulate Adder.Total
Example: Adder as an ADT

package Adder is
    Threshold: constant Integer := 100000;
    subtype Amount is Integer range 1 .. Threshold;
    function Get_Total return Amount;
    procedure Add (X: in Amount);
    procedure Clear;
end Adder;

package body Adder is
    Total: Amount := 0;
    procedure Add_Threshold (X: in Integer) is begin ...
    procedure Clear is begin ...
end Adder;
Refinement: Compositional proofs

- Client code cannot depend on the existence of variables declared in the package body
- Would like the same separation for proofs
- Refinement:
  - Declare *Abstract_State* for use in contracts in package specification
  - Package body refines the abstract state into a set of *concrete variables*
  - Decouples specification from implementation: provided changes to the body do not affect specification, no need to recompile/reanalyze clients
Ex: Adder as an ADT with data flow aspects

package Adder
with SPARK_Mode,
    Abstract_State => T,
    Initializes => T
is
    Threshold: constant Integer := 100000;
    subtype Amount is Integer range 1 .. Threshold;
    function Get_Total return Amount
        with Global => T;
    procedure Add (X: in Amount)
        with Global => (In_Out => T);
    procedure Clear
        with Global => (Output => T);
end Adder;

T refers to the state encapsulated by Adder

Adder’s state is initialized by elaboration of the package body

Adder’s state is a (global) input (used by Get_Total)

Adder’s state is both an input and output (used and defined)

Adder’s state is output, and not input (defined only)
package Adder
with SPARK_Mode,
    Abstract_State => T,
    Initializes => T
is
    Threshold: constant Integer := 100000;
    subtype Amount is Integer range 1 .. Threshold;
    function Get_Total return Amount
    with Global => T;
    procedure Add (X: in Amount)
    with Global => (In_Out => T);
    procedure Clear
    with Global => (Output => T);
end Adder;

Permits compositional data flow analysis of clients
package body Adder
   with SPARK_Mode, Refined_State => (T => Total)
is
   Total : Integer := 0;
   procedure Add (X: in Amount)
      with Refined_Global => (In_Out => Total)
   is begin
      if Total + X <= Threshold then
         Total := Total + X;
      else
         Total := Threshold;
      end if;
   end Add;
   function Get_Total return Amount
   is (Total) with Refined_Global => Total;
   procedure Clear
      with Refined_Global => (Output => Total)
   is begin
      Total := 0;
   end Clear;
end Adder;

Refinement definition:
value of $T$ is defined by the value of $Total$.

Refine the Global aspects:
written in terms of the concrete variable $Total$. 
package body Adder
   with SPARK_Mode, Refined_State => (T => Total)
is
   Total : Integer := 0;

   procedure Add (X: in Amount)
      with Refined_Global => (In_Out => Total)
   is begin
      if Total + X <= Threshold then
         Total := Total + X;
      else
         Total := Threshold;
      end if;
   end Add;

   function Get_Total return Amount
   is (Total) with Refined_Global => Total;

   procedure Clear
      with Refined_Global => (Output => Total)
   is begin
      Total := 0;
   end Clear;
end Adder;

Static analysis:
- Check the Refined_Global aspects: DFA of procedures bodies
- Check that each Global aspect in the spec is correctly refined (same mode and consistent with refinement def)
Example: Stack ADT

package Stack ... is
    function Is_Full return Boolean;
    function Is_Empty return Boolean;
    procedure Push(X: in integer);
    function Get_Top return integer;
    procedure Pop(X: out integer);
end Stack;

package body Stack ... is
    Stack_Size: constant := 100;
    type Top_Range is range 0 .. Stack_Size;
    subtype Index_Range is Top_Range range 1 .. Stack_Size;
    S: array (Index_Range) of Integer := (Index_Range => 0);
    Top: Top_Range := 0;

    function Is_Full return Boolean is (Top = Stack_Size);
    ...

Example: Depends aspects for Stack

```plaintext
package Stack with SPARK_Mode,
  Abstract_State => The_Stack,
  Initializes => The_Stack
is
  function Is_Full return Boolean
    with Depends => (Is_Full'Result => The_Stack);
  ...
  function Get_Top return integer
    with Depends => (Get_Top'Result => The_Stack);
  procedure Push(X: in integer)
    with Depends => (The_Stack => (The_Stack, X));
  ...
  procedure Clear
    with Depends => (The_Stack => null,
                    null => The_Stack);
end Stack;
```

Permits compositional information flow analysis of clients
package body Stack with SPARK_Mode, 
  Refined_State => (The_Stack => (S, Top))
is

... 
function Is_Full return Boolean 
is (Top = Stack_Size) 
  with Refined_Depends => (Is_Full'Result => Top);
...

procedure Push(X: in Integer)
  with Refined_Depends => (Top => Top, S => (Top, X, S))
is begin 
  Top := Top + 1; S(Top) := X;
end Push;
...

procedure Clear 
  with Refined_Depends => (Top => null, S => null, 
  null => (Top, S))
is begin 
  Top := 0; S := (Index_Range => 0);
end Clear;
end Stack;

Refinement definition: value of The_Stack is defined by values of S and Top

Refine the Depends clauses: written in terms of the concrete variables.
package body Stack with SPARK_Mode,  
    Refined_State => (The_Stack => (S, Top))  
is  

    ...  
    function Is_Full return Boolean  
is (Top = Stack_Size)  
    with Refined_Depends => (Is_Full'Result => Top);  
    ...  
    procedure Push(X: in Integer)  
    with Refined_Depends => (Top => Top, S => (Top, X, S))  
is begin  
        Top := Top + 1; S(Top) := X;  
    end Push;  
    ...  
    procedure Clear  
    with Refined_Depends => (Top => null, S => null,  
                              null => (Top, S))  
is begin  
        Top := 0; S := (Index_Range => 0);  
    end Clear;  
end Stack;

Static analysis:  
- Perform IFA of procedure bodies to check the Refined_Depends aspects  
- Check that each Depends aspect in the spec is correctly refined (consistent with the refinement definition)
Example: Use of Depends aspect for Stack

```pascal
with Stack;

procedure Use_Stack ( ... Y: out integer)
    with Depends => (Y => Stack.The_Stack,
                    ... )
is begin
    ... 
    X := Stack.Get_Top( ... );
    ... 
    Y := 3*X;
    ... 
end Use_Stack;
```

Client imports and exports the abstract machine state.

Permits flow analysis without knowing how the machine state is refined.
Example: Stack ADT with Pre/Post Contracts

package Stack with SPARK_Mode,
  Abstract_State => The_Stack,
  Initializes => The_Stack
is
  ...
  function Get_Top return integer
  with Depends => (Get_Top'Result => The_Stack),
    Pre => not Is_Empty;

procedure Push(X: in integer)
  with Depends => (The_Stack => (The_Stack, X)),
    Pre => not Is_Full, Post => not Is_Empty;
  ...

procedure Clear
  with Depends => (The_Stack => null, null => The_Stack),
    Post => Is_Empty;
end Stack;

Permits compositional proof of client code
package body Stack with SPARK_Mode,
  Refined_State => (The_Stack => (S, Top)) is

  ...
  function Get_Top return integer is ( S(Top) )
    with Refined_Depends => (Get_Top'Result => (S, Top));

  procedure Push(X: in Integer)
    with Refined_Depends => (Top => Top, S => (Top, X, S)),
      Refined_Post => (Top = Top'Old + 1 and 
      S = S'Old'Update(Top => X))
  is begin
    Top := Top + 1; S(Top) := X;
  end Push;

  ...

  procedure Clear
    with Depends => (The_Stack => null, null => The_Stack),
      Refined_Post => (Top = 0)
  is begin
    Top := 0; S := (Index_Range => 0);
  end Clear;
end Stack;
Refinement proof obligations

• Verify each refined post condition and absence of RTEs, as usual, by hoisting checks through the procedure body to an assume/assert/pre condition:
  – $H \rightarrow WP(RTE\_CHECK, P_{RTE})$
  – $H \rightarrow WP(REFINED\_POST, P_{body})$

• Verify the abstract post condition:
  – $PRE'\_Old \land RENIED\_POST \rightarrow POST$
Example: Push checks and assumptions

```plaintext
procedure Push(X: in Integer)
with ... Pre => not Is_Full,
    Post => not Is_Empty;

begin
    Top := Top + 1;
    S(Top) := X;
end Push;

function Is_Full return Boolean is (Top = Stack_Size);

function Is_Empty return Boolean is (Top = 0);

procedure Push(X: in Integer)
with ... Refined_Post => (Top = Top'Old + 1 and
                         S = S'Old'Update(Top => X));

Assume:
Top in 0 .. Stack_Size
PRE:
not (Top = Stack_Size)

Implicit post: Is_Full = (Top = Stack_Size)

POST check: Top = Top'Old + 1 and
            S = S'Old'Update(Top => X)

RTE check:
Top + 1 in 0 .. Stack_Size
```
procedure Push(X: in integer)
    with ... Pre => not Is_Full,
    Post => not Is_Empty;

function Is_Full return Boolean is (Top = Stack_Size);

**Implicit post:** Is_Full = (Top = Stack_Size)

function Is_Empty return Boolean is (Top = 0);

**Implicit post:** Is_Empty = (Top = 0)

procedure Push(X: in Integer)
    with ... Refined_Post => (Top = Top'Old + 1 and
    S = S'Old'Update(Top => X))

is...

Top'Old in 0 .. Stack_Size and
not (Top'Old = Stack_Size) and
Top = Top'Old+1 and S = S'Old'Update(Top => X)
→
not (Top = 0)
Example: Use of Post aspect for Stack

with Stack;

procedure Use_Stack ( ... Y: out integer)
  with Pre => (not Stack.Is_Full, ... )
is begin
  ...
  Stack.Push( ... );
  ...
  Stack.Pop( ... );
  ...
end Use_Stack;

Compositional proof of absence of RTEs.
Acknowledgements & references