6: Advanced SPARK

Previously, we did

- provide a quick into into the SPARK language and toolset,
- give a couple of simple examples,
- introduce the Hoare logic, and
- explain the weakest precondition calculus.

Now we get deeper into using the SPPARK language and toolset:

- We revisit the Math example (`Log_2` has not yet been done)
- we recall the linear search example,
- and then, we take the binary search challenge ("Only 10% of all programmers can implement binary search properly"),

Finally, we will explain how SPARK handles **generics**!

6.1: Math Revisit (Log_2 & SQRT)

```
pragma Spark_Mode(On);

package Math is

   function SQRT(X: Natural) return Natural with Contract_Cases
     => ( X <= 1 => SQRT'Result = X,
          X > 1  => SQRT'Result > 1 and then
                    X/SQRT'Result <= SQRT'Result and then
                    X/(SQRT'Result-1) > (SQRT'Result-1)     );

   function Log_2(X: Positive) return Natural with
     Post => 2**Log_2'Result <= X and then
             2**Log_2'Result > X/2;

end Math;
```

Math

Implementation of SQRT (Recalled)

```
function SQRT(X: Natural) return Natural is
   R: Positive;
begin
    if X <= 2 then
        return X;
    else
        R := 2;
        while X/R > R loop
            pragma Loop_Variant(Increases => R);
            R := R + 1;
        end loop;
        return R;
    end if;
end SQRT;
```

Math

Implementation of Log_2 (here begins the new stuff)

```
function Log_2(X: Positive) return Natural is
    Log: Natural := 0;
    Pow: Positive := 1;
 begin
    while Pow <= X/2 loop
       pragma Loop_Invariant(Pow = 2**Log);
       pragma Loop_Variant(Increases => Log);
       Log := Log + 1;
       Pow := Pow * 2;
    end loop;
    return Log;
 end Log_2;
```

- when entering: Pow = 2**Log holds.
- from iteration to iteration: if Pow'old = 2**Log'old and then Pow'new $<=$ X/2 then Pow'new = 2**Log'new.
- when leaving: if Pow = 2**Log and POW $>$ X/2 then postcondition.

An Unexpected Problem

overflow check might fail (e.g. when Log = 0)

```
function Log_2(X: Positive) return Natural is
    Log: Natural := 0;
    Pow: Positive := 1;
 begin
    while Pow <= X/2 loop
       pragma Loop_Invariant(Pow = 2**Log);
       pragma Loop_Variant(Increases => Log);
       Log := Log + 1;
       Pow := Pow * 2;
    end loop;
    return Log;
 end Log_2;
```

SPARK gives examples, **why** something fails. The examples can be very helpful. Sometimes! Here, it isn't.

In any case, SPARK fails to disprove that `Log + 1` might overflow.

Is this a real bug in our code? (I.e., can this ever happen?)

An Unexpected Problem (2)

overflow check might fail (e.g. when Log = 0)

```
   while Pow <= X/2 loop
      pragma Loop_Invariant(Pow = 2**Log);
      pragma Loop_Variant(Increases => Log);

      Log := Log + 1;

      Pow := Pow * 2;
   end loop;
```

Our analysis:

- `Log` < `Pow` (Proof by Induction.)

Apparently, SPARK can prove that `Pow` does not overflow.

But SPARK fails to draw the conclusion:

- If `Pow` does not overflow, then neither does `Log + 1`.

What can we do now?

Leave the Loop just Before Log would Overflow

we know that this does NEVER happen, but SPARK doesn't . . .

```
function Log_2(X: Positive) return Natural is
   Log: Natural := 0;
   Pow: Positive := 1;
begin
   while Pow <= X/2 and Log < Integer'Last
   loop
      pragma Loop_Invariant(Pow = 2**Log);
      pragma Loop_Variant(Increases => Log);
      Log := Log + 1;
      Pow := Pow * 2;
   end loop;
   return Log;
end Log_2;
```

The Problem is not yet gone

postcondition might fail, cannot prove 2 Log_2'Result > X/2**

```
function Log_2(X: Positive) return Natural with
  Post => 2**Log_2'Result <= X and then

          (2**Log_2'Result > X/2);
```

Tweak the Spec

```
function Log_2(X: Positive) return Natural with
  Post => 2**Log_2'Result <= X and then
          (Log_2'Result = Integer'Last or else
           2**Log_2'Result > X/2);
```

This keeps SPARK happy!

Tweaking the Spec is Cheating!

The condition "`Log_2'Result = Integer'Last` is

- is artificial,
- is pointless for the body,
- and harms the readability of the spec.

The natural Spec is this:

```
function Log_2(X: Positive) return Natural with
  Post => 2**Log_2'Result <= X and then
          2**Log_2'Result > X/2;
```

How can we tweak the body, while maintaining this spec?

The Solution is . . . Almost too Obvious

It had been obvious to us that "`Log := Log + 1`" cannot overflow, because `Log` is always smaller than `Pow`.

Previously, we put the conclusion "`Log <= Integer'Last`" into the exit condition for the loop.

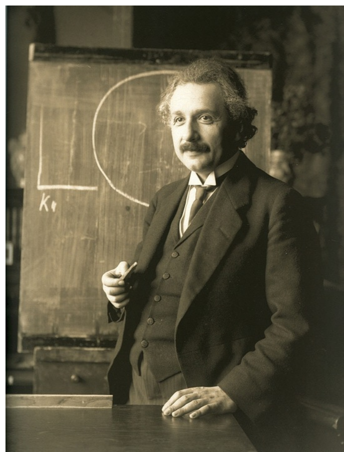
However, following our own analyis `Log < Pow` is invariant in the loop. Why don't we make this a part of the loop invariant?

```
while Pow <= X/2 loop
   pragma Loop_Invariant(Pow = 2**Log and Log < Pow);
   pragma Loop_Variant(Increases => Log);
   Log := Log + 1;
   Pow := Pow * 2;
end loop;
```

Both SPARK and we are happy. Finally!

Clever Comments (1)

- If SPARK fails to prove your code:
  - Your code may be wrong.
  - Try to find the flaw.
- But if it may appear obvious to you that your code is correct – and continues to miss something “obvious”:
  1. Ask yourself: *why is my code correct* and *what is this “obvious” property?*
  2. If you can actually make out such a property, tell SPARK about it.
- Often, this will suffice to allow SPARK to prove your code.



6.2: Linear Search Recall

```
pragma Spark_Mode(On);

package Search is

   No_Such_Index: constant Natural := 0;

   type Item is new Natural;
   type Arr is array (Positive range <>) of Item;

   function Lin_Search (A: Arr; Value: Item) return Natural
     with Post =>
     ((Lin_Search'Result not in A'Range and then
         (for all J in A'Range => not(A(J) = Value))
          ) or else
        A(Lin_Search'Result)=Value
     );

end Search;
```

The Implementation

```
pragma Spark_Mode(On);

package body Search is

   function Lin_Search (A: Arr; Value: Item) return Natural is
   begin
      for I in A'Range loop
         if A(I) = Value then
            return I;
         end if;
      end loop;
      return No_Such_Index;
   end Lin_Search;

end Search;
```

Why doesn't SPARK like this?

. . . do you remember?

```
function Lin_Search (A: Arr; Value: Item) return Natural
  with Post =>
  ((Lin_Search'Result not in A'Range and then
```

postcondition might fail . . . (e.g. when

A = (1 => 0, **others** => 1) **and** A'First = 1 **and**

A'Last = 2 **and** Lin_Search'Result = 0 **and** Value = 0)

```
        (for all J in A'Range => not(A(J) = Value))
         ) or else
       A(Lin_Search'Result)=Value
```

array index check might fail (e.g. when

A = (1 => 0, **others** => 1) **and** A'First = 1 **and**

A'Last = 2 **and** Lin_Search'Result = 0)

```
       );
```

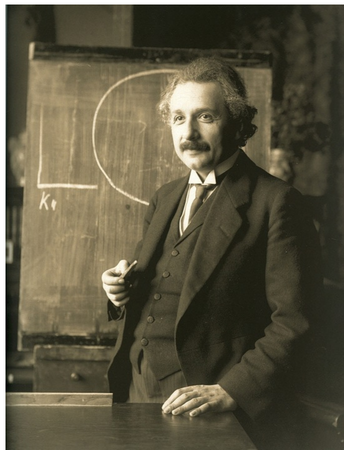
Ah, we Forgot the loop Invariant!

```
pragma Spark_Mode(On);

package body Search is

   function Lin_Search (A: Arr; Value: Item) return Natural is
   begin
      for I in A'Range loop
         pragma Loop_Invariant
           (for all J in A'First .. I-1 => A(J) /= Value);
        if A(I) = Value then
          return I;
        end if;
      end loop;
      return No_Such_Index;
   end Lin_Search;

end Search;
```

Now SPARK is happy – and so are we.

Clever Comments (2)

- If SPARK fails at a postcondition:
  - Consider your loop invariant.
  - Write one (if you have none).
  - Extend it (if you already have one).
- A loop invariant
  1. must be true before entering the loop
  2. must hold in the current iteration, if it did hold in the previous one,
  3. and, together with the exit condition, the invariant must allow to prove the postcondition
- Often, this will suffice to allow SPARK to prove your code.



6.3: The Challenge: Binary Search

"Only 10 % of all programmers can implement binary search properly." Challenge taken!

Informal spec:

> *Given a sorted array A and a value V as the input, return an index I, such that $A(I) = V$.*
> *Make only $O(log n)$ comparisons.*

This is incomplete, however:

- What shall we return, if no such I exists? (Especially, when A is empty?)
- If there are several I_1, I_2, with $A(I_1) = A(I_2) = \cdots = V$, which of these indices shall we return? (Does it matter, at all?)

Extended Specification

0. Precondition: A is sorted.
1. If A is empty, return special value for "no such index".
2. If A is not empty:
   - If I with $A(I) = V$ exists, return first such I.
   - Elsif I with $A(I) > V$ exists, return first such I.(!)
   - Else (A is not empty, but all items in A are smaller than V) also return "no such index".

The Abstract Specification

We don't even need a precondition

1. If A is empty: Result = none.
2. Else if $A(\text{First}) > V$: Result = none.
3. Else if $A(\text{First}) = V$: Result = First;

   (* Observe $A(\text{First}) < V$. *)
4. Else if $V \leq A(\text{Last})$:

   (* Observe First $<$ Last. *)
   - Both Result and Result-1 are in the range of A and then
   - $A(\text{Result}) \geq V$ and then
   - $A(\text{Result}-1) < V$.
5. Else Result = Last

simple specification $\Rightarrow$ easy verification?

Formal Analysis

Theorem:

1. If $A(\text{First}) > V$ and $V \leq A(\text{Last})$,
   then an index Result exists, such that
   - Both Result and Result-1 are in the range of A and then
   - $A(\text{Result}) \geq V$ and then
   - $A(\text{Result}-1) < V$.
2. If, in addition to $A(\text{First}) > V$ and $V \leq A(\text{Last})$ A is also sorted,
   then this Result is uniquely defined.

The Spec

```
type Item is new Natural;
type Arr is array (Positive range <>) of Item;
No_Such_Index: constant Natural := 0;
```

(same as for Lin_Search)

```
function Bin_Search (A: Arr; Value: Item) return Natural
  with Post =>
  (if A'First > A'Last or else Value < A(A'First) then
     Bin_Search'Result = No_Such_Index
  elsif A'First = A'Last or else A(A'First) = Value then
     Bin_Search'Result = A'First -- else A(A'First) < Value
  elsif Value < A(A'Last) then
    Bin_Search'Result > A'First and
    Bin_Search'Result <= A'Last and
    A(Bin_Search'Result-1) < Value and
    A(Bin_Search'Result) >= Value
  else Bin_Search'Result = A'Last
 );
```

The Abstract Algorithm

1. **if** A is empty, **return** special value for “no such index”.
2. **elsif** $A(\text{First}) > V$, **return** special value for “no such index”.
3. **elsif** $A(\text{First}) = V$, **return** First.

   (∗ At this point, we know $A(\text{First}) < V$. ∗)
4. **elsif** $V \leq A(\text{Last})$

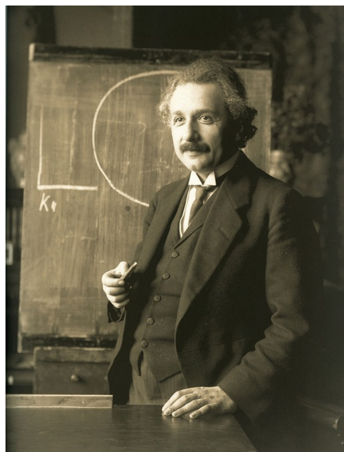
   Low := A’First; High := A’Last; **while** Low+1 $<$ High:

   4.1 Invariant: $A(\text{Low}) < V \leq A(\text{High})$
   4.2 Middle := . . . (∗ Low $<$ Middle $<$ High ∗)
   4.3 **if** $A(\text{Middle}) < V$ then Low := Middle;
   4.4 **else** (∗$A(\text{Middle}) \geq V$∗) High := Middle;

   return High;
5. **else return** Last;

Clever Comments (3)

- Always *think* first,
- then code and try to prove your code!
- If you can simplify your specification, (e.g., by eliminating a precondition which is not really required), you simplify the task for SPARK ...
- ... and you ease your own work.



The First Implementation (1)

```
function Bin_Search (A: Arr; Value: Item) return Natural is

   function Middle(X: Positive; Y: Natural) return Positive
      is ((X+Y)/2);

   Low:   Positive := A'First;
   High:  Natural  := A'Last;
   Idx:   Positive;

begin
  ... -- see next slide
end Bin_Search;
```


Bauhaus-Universität Weimar

The First Implementation (2)

```
begin
   if A'First > A'Last or else Value < A(A'First) then
      return No_Such_Index;
   elsif A'First = A'Last or else A(A'First) = Value then
      return A'First;         -- else A(A'First) < Value
   elsif Value < A(A'Last) then
     while Low < High loop
        ... -- main loop, see next slide
     end loop;
     return High;
   else
      return A'Last;
   end if;
end Bin_Search;
```

The First Implementation (Main loop)

```
-- A is not empty and A(A'First) < Value
-- and Value >= A(A'Last)
while Low < High loop
   pragma Loop_Invariant
   (Low in A'Range  and then
    High in A'Range and then
    (A(Low) < Value and A(High) >= Value)
   );
   pragma Loop_Variant(Decreases => High-Low);
   Idx := Middle(Low, High);
   if A(Idx) < Value then
      Low := Idx; -- A(Low) < Value
   else -- A(Idx) >= Value
      High := Idx; -- A(High) >= Value
   end if;
end loop;
```

Spark has many complaints

Any ideas, why?

First Two Complaints

Why is X = 2147483647 and Y = 2 a counter-example?

```
function Bin_Search (A: Arr; Value: Item) return Natural is

   function Middle(X: Positive; Y: Natural) return Positive
      is ((X+Y)/2);
```

overflow check might fail (e.g. **when** X = 2147483647 **and** Y = 1)
range check might fail

Explanation

Integer'Last+1 overflows.
On the machine I am using,

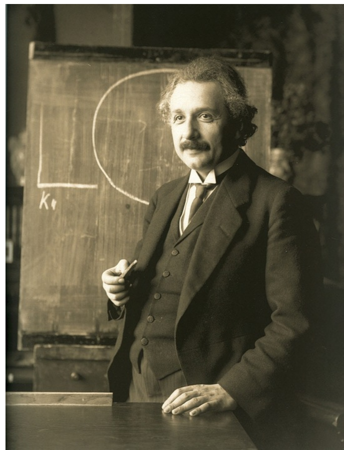
Integer'Last=2147483647,

thus

Integer'Last+1=2147483648

overflows.

SPARK needs to be configured to work with whatever Ada compiler you are working with.
By default, SPARK assumes the constants from `gnat`.



First Two Complaints (2)

Why is X = 2147483647 and Y = 2 a counter-example?

```
function Bin_Search (A: Arr; Value: Item) return Natural is

   function Middle(X: Positive; Y: Natural) return Positive
      is (X+Y/2);
```

overflow check might fail **Of Course:** X + Y may be out of range
range check might fail **follow-on-failure**

The Middle Function

```
function Middle(X: Positive; Y: Natural) return Positive
   is ((X+Y)/2);
```

overflow check might fail

X + Y may be out of range **Idea:**

1. compute (X/2) + (Y/2) (round towards zero)
2. add a commentsation ((X **mod** 2) + ((Y **mod** 2) + 1) for rounding down twice

```
function Middle(X: Positive; Y: Natural) return Positive
  is ( (X/2) + (Y/2) + ((X mod 2) + ((Y mod 2) + 1) / 2) );
```

Will This do the job?

No:

overflow check might fail (e.g. when X = 2147483647 **and** Y = 2147483647)

The Middle Function (2)

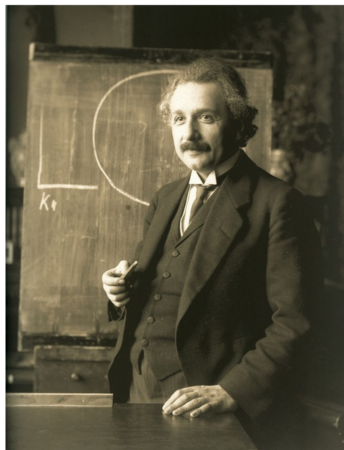
```
function Middle(X: Positive; Y: Natural) return Positive
 is ( (X/2) + (Y/2) + (((X mod 2) + (Y mod 2) + 1) / 2) );
```

This, finally, keeps SPARK happy!

Clever Comments (4)

- The isssue in ((X+Y)/2) was obvious, easy to understand and even easy to avoid.
- However, the "bracket-soup" in our second and third definition of the function Middle is
  - easy to do wrong and
  - hard to analyse

  for humans. Even when told that an "overflow check might fail", without an example, humans may find it hard to understand the issue.
- In such cases, the examples generated by SPARK can be extremely helfpul!



The Next Two complaints

```
Low:   Positive  := A'First;
```

range check might fail
(e.g. when A = (**others** => 0) **and** A'First = 0 **and** A'Last = -1 **and** Low = 0)

```
High:  Natural  := A'Last;
```

range check might fail
(e.g. when A = (**others** => 0) **and** A'First = 1 **and** A'Last = -1 **and** High = 0)

I am not sure about these issues.

Obviusly, this is about the representation of empty arrays – but then, I would expect A'First=1 **and** A'Last=0, which is allowed for empty arrays of Positive range.

Never mind – let's fix the problem!

The Repair, which keeps SPARK happy

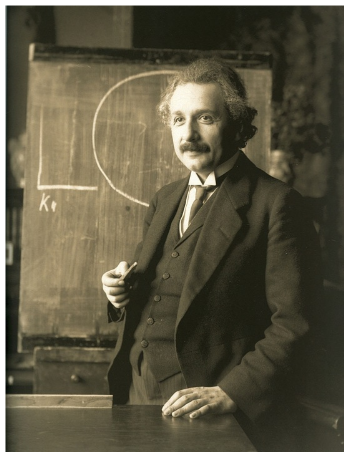
```
    Low:   Positive; -- no initialization here
    High:  Natural;  -- no initialization here
    Idx:   Positive;

  begin
```

```
    elsif Value < A(A'Last) then
       -- A is not empty and A(A'First) < Value
       -- and Value >= A(A'Last)
       Low := A'First; -- initialization now
       High := A'Last; -- initialization now
       while Low < High loop -- first use of High and Low
```

```
     end loop;
       return High;
     else
        return A'Last;
     end if;
  end Bin_Search;
```

Clever Comments (5)

- Sometimes, SPARK may complain, but even after some careful consideration, you may not be sure if there is a real issue in your code or if you suffer from a SPARK constraint.
- Never mind, as the current example shows, the examples generated by SPARK are helpful to understand and fix the problem.



The Final Complaint

```
Low  := A'First;
High := A'Last;
while Low < High loop
   pragma Loop_Invariant
   (Low in A'Range  and then
    High in A'Range and then
    (A(Low) < Value and not (A(High) < value))
   );
    pragma Loop_Variant(Decreases => High-Low);
```

loop variant might fail (e.g. when High = 3 **and** Low = 2)

```
   Idx := Middle(Low, High);
   if A(Idx) < Value then
      Low := Idx; -- A(Low) < Value
   else -- A(Idx) >= Value
      High := Idx; -- A(High) >= Value
   end if;
end loop;
return High;
```

The Final Complaint (2)

```
while Low < High loop
```

```
pragma Loop_Variant(Decreases => High-Low);
```

loop variant might fail (e.g. when High = 3 **and** Low = 2)

```
Idx := Middle(Low, High);
if A(Idx) < Value then
   Low := Idx; -- A(Low) < Value
else -- A(Idx) >= Value
   High := Idx; -- A(High) >= Value
end if;
end loop;
return High;
```

What is wrong?

Shall we change the definition of the Middle again?

The Implementation (Main Loop, Final)

```
Low := A'First;
High := A'Last;
while Low+1 < High loop
   pragma Loop_Invariant
   (Low in A'Range  and then
    High in A'Range and then
    (A(Low) < Value and A(High) >= Value)
   );
   pragma Loop_Variant(Decreases => High-Low);
   Idx := Middle(Low, High);
   if A(Idx) < Value then
      Low := Idx; -- A(Low) < Value
   else -- A(Idx) >= Value
      High := Idx; -- A(High) >= Value
   end if;
end loop;
```

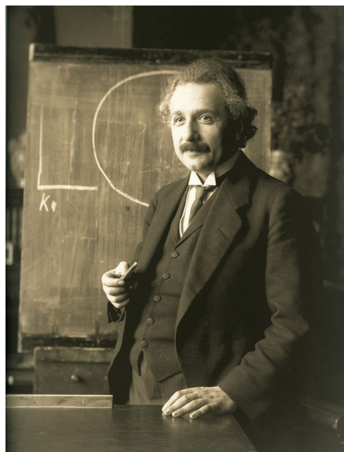
Now, SPARK is happy!

A little statistic:

- Bin_Search flow analyzed (0 errors, 0 checks and 0 warnings) and proved (21 checks)
- Bin_Search.Middle flow analyzed (0 errors, 0 checks and 0 warnings) and proved (7 checks)
- S.Lin_Search flow analyzed (0 errors, 0 checks and 0 warnings) and proved (7 checks)

Clever Comments (6)

- Once again, the example had been helpful to discover the flaw in the code.
- Just telling the user “loop variant might fail” would eventually do the job, but the specific example greatly simplifies the search for the reason *why* the loop variant might fail – and that this is not just a SPARK issue.



6.4: Claiming Performance

```
while Low+1 < High loop
   pragma Loop_Invariant
   (Low in A'Range  and then
    High in A'Range and then
    (A(Low) < Value and A(High) >= Value)
   );
```

```
   pragma Loop_Variant(Decreases => High-Low);
```

```
   Idx := Middle(Low, High);
   if A(Idx) < Value then
      Low := Idx; -- A(Low) < Value
   else -- A(Idx) >= Value
      High := Idx; -- A(High) >= Value
   end if;
end loop;
```

The loop variant High−Low is linear. Can we prove logarithmic run time (as to be expected from binary search)? (→ black- or whiteboard)

6.5: Generics in SPARK

Generics pose a special problem for static verification.

A generic implementation of binary search needs the following generic parameters:

1. Item **is limited private**;
2. **type** Arr **is array** (Positive **range** <>) **of** Item;
3. and operators to compare two items: “$<$”, “$>$” “$=$”, “$\geq$”, ...

But then, SPARK would need generic contracts also for the comparison operators (or, in general, for the generic parameters).

The Trouble with Generic Parameters

E.g., consider an instantiation of binary search with

Item => Natural, "=" => Equal, "<" => Smaller, ">" => Bigger

using the following functions as comparison operators:

```
function "Equal"(Left, Right: Natural) return Boolean is
  (Left > 0 and Right > 0);
function "Smaller"(Left, Right: Item) return Boolean is
  (Right = 0);
function "Bigger"(Left, Right: Item) return Boolean is
  (Left = 0);
```

An implementation of binary search can hardly to anything useful, when it is using these functions as comparison operators.

Functional Contract for Comparison Operators

focus on "$<$" and "$=$";
the other ones ("$>$", "$\neq$", "$\leq$", and "$\geq$") can be derved from these two

Symmetry of "$=$": for $A, B \subseteq$ Item: if $A = B$ then $B = A$,

Antisymmetry of "$<$": for $A, B \subseteq$ Item: if $A < B$ then not $(B < A)$,

Transitivity of "$=$": for $A, B, C \subseteq$ Item: if $A = B$ and $B = C$ then $A = C$.

Transitivity of "$<$": for $A, B, C \subseteq$ Item, if $A < B$ and $B < C$ then $A < C$.

Trichotomy: for $A, B \subseteq$ Item, either of $A = B$, $A < B$ or $B < A$ holds.

In an ideal world

1. The prover would be given these generic parameters
   1.1 Item **is limited private**;
   1.2 **type** Arr **is array** (Positive **range** <>) **of** Item;
   1.3 and operators to compare two items: “$<$”, “$>$” “$=$”, “$\geq$”, ...

   plus claims about trichotomy, (anti-)symmetry, and transitivity
2. The prover would then verify the generic implementation.
3. For every instantiation, the prover would verify that the instantiation matches the claims.

How does SPARK handle generics?

Until very recently, SPARK did not support generics at all.

Today:

- SPARK supports generics.
- But in SPARK, one cannot yet specify functional contracts for generic parameters.
- Thus SPARK cannot verify the correctness of a generic implementation.
- But note that, when the generic is instantiated, SPARK has the information it needs.
- Thus, SPARK performs a full verification for each instantiation.
- The advantage is that SPARK now supports generic programming.
- The disadvantage is that if one has several very similar instantiations, SPARK performs essentially the same work again and again, for each instantiation.

6.6: Generic Math

Implement mathematical functions over generic integer types, rather than over **type** Integer:

- parameter: generic type Nat, derive Pos
- both in in spec and body: replace Natural by Nat and Positive by Pos:

```
pragma Spark_Mode(On);

generic
   type Nat is range <>;
package Math is

subtype Pos is Nat range Nat'First+1 .. Nat'Last;

   function SQRT(X: Nat) return Nat with Contract_Cases =>
      ...;

   function Log_2(X: Pos) return Nat with Post =>
      ...;

end Math;
```

The full generic Spec

```
pragma Spark_Mode(On);

generic
   type Nat is range <>;
package Math is

subtype Pos is Nat range Nat'First+1 .. Nat'Last;

   function SQRT(X: Nat) return Nat with Contract_Cases =>
     ( X <= 1 => SQRT'Result = X,
       X > 1  => SQRT'Result > 1 and then
                 X/SQRT'Result <= SQRT'Result and then
                 X/(SQRT'Result-1) > (SQRT'Result-1)     );

   function Log_2(X: Pos) return Nat with Post =>
     2**(Log_2'Result) <= X  and then
     2**(Log_2'Result) > X/2;

end Math;
```

This does not even compile

The power in the predefined ""-operator must be of the predefined Natural.**

```
generic
   type Nat is range <>;                    --> line 3
package Math is
```

- math.ads:18:15: exponent must be of type Natural, found type "Nat" defined at line 3

```
   function Log_2(X: Pos) return Nat with Post =>
     2**(Log_2'Result) <= X  and then --> line 18
```

- math.ads:19:15: exponent must be of type Natural, found type "Nat" defined at line 3

```
     2**(Log_2'Result) > X/2;           --> line 19
```

- one more such error in body:

```
         pragma Loop_Invariant(Pow = 2**Log and Log < Pow);
```

This Compiles Well

```
pragma Spark_Mode(On);
generic
   type Nat is range <>;
   with function To_Natural (N: Nat) return Natural is <>;
package Math is
   ...

   function Log_2(X: Pos) return Nat with Post =>
     2**To_Natural(Log_2'Result) <= X  and then
     2**To_Natural(Log_2'Result) > X/2;
end Math;
```

spec

```
         while Pow <= X/2 loop
          pragma Loop_Invariant(Pow = 2**To_Natural(Log)
                                          and Log < Pow);
```

body

Some Instantiations to Prove

. . . and SPARK is happy with all of them!

```
pragma Spark_Mode(On); with Math;

procedure Test_Math is

   type I08 is range 0 .. 255;
   type I10 is range 0 .. 1024;
   type I31 is range 0 .. 2**31-1;

   function To_Natural(N: I08) return Natural is (Natural(N));
   function To_Natural(N: I10) return Natural is (Natural(N));
   function To_Natural(N: I31) return Natural is (Natural(N));

   package M08 is new Math(I08, To_Natural);
   package M10 is new Math(I10, To_Natural);
   package M31 is new Math(I31, To_Natural);

begin
  null; -- we only care about the instantiations
end Test_Math;
```


Bauhaus-Universität Weimar

Bigger Numbers FAIL

```
type I63 is range 0 .. 2**63-1;

function To_Natural(N: I63) return Natural is (Natural(N));
-- ERROR!

package M63 is new Math(I63, To_Natural);
```

range check might fail (e.g. when N = 2147483648)

Negative Numbers FAIL

```
Type Fool is range -1 .. 99;

function To_Natural(N: Fool) return Natural is (Natural(N));
-- ERROR!

package Fools is new Math(Fool, To_Natural);
```

range check might fail (e.g. when N = 2147483648)

Numbers from range not starting with zero FAIL

```
type Dummy is range 1 .. 199;

function To_Integer(N: Dummy) return Natural is (Integer(N));

package Dummies is new Math(Dummy, To_Natural);
```

instantiation error

value not in range of type “Nat”

“Constraint_Error” would have been raised at run time

instantiation error

value not in range of type “Pos”

“Constraint_Error” would have been raised at run time

How would an ideal spec look like?

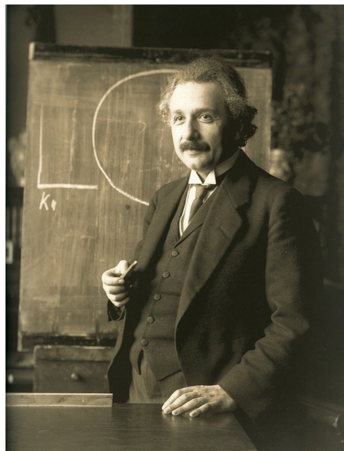
```
generic
   type Nat is range <>;
   with function To_Natural (N: Nat) return Natural is <>;
   with generic Pre =>
      Nat'First = 0 and
      Nat'Last <= Natural'Last;
   -- I just invented a new syntax for Ada and SPARK!
package Math is
   ...
end Math;
```

Clever Comments (7)

- Currently, SPARK doesn't offer this:

```
with generic Pre =>
  Nat'First = 0 and
  Nat'Last <= Natural'Last;
```

  and it isn't even proper Ada syntax.
- If implemented in a future revision (using whatever syntax), it might simplify working with generics in Ada and SPARK:
  - It pushes the design-by-contract idea to the instantiation of Ada generics.
  - SPARK would verify the generic implementation, assuming the properties.
  - For the instantiation, SPARK would just check the claimed properties.



6.7: Generic Linear Search

- Generic Parameter type Item generic.
- Derived Parameter array (Positive range <>) of Item
- "="-operator for equality

```
generic
   type Item is limited private;
   type Arr is array (Positive range <>) of Item;
   with function "="(Left, Right: Item) return Boolean is <>;
function Lin_Search (A: Arr; Value: Item) return Natural
  with Post =>
  ((Lin_Search'Result not in A'Range and then
      (for all J in A'Range => not(A(J) = Value))
       ) or else
     A(Lin_Search'Result)=Value
  );
```

- No change in body!

Some Instantiations for Linear Search

... which we will revist for binary search

1. Item = Natural
2. Item = fixed-length string
3. Item = Record
   (ID: Natural, Name: String, Birthyear: Some_Range)
   (a) "=": same ID
   (b) "=": same Name

```
generic
   type Item is limited private;
   type Arr is array (Positive range <>) of Item;
   with function "="(Left, Right: Item) return Boolean is <>;
function Lin_Search (A: Arr; Value: Item) return Natural
  with Post =>
  ((Lin_Search'Result not in A'Range and then
      (for all J in A'Range => not(A(J) = Value))
       ) or else
     A(Lin_Search'Result)=Value
  );
```

First Instantiation: Item = Natural

```
pragma Spark_Mode(On);

with Search;

procedure Lin_Test_1 is

   -- 1st type of items: predefined Natural
   type Natural_Arr is array(Positive range <>) of Natural;

   function Lin_Search_Nat is new Search.Lin_Search
     (Natural, Natural_Arr);

begin
   null; -- We care about the instantiations above,
         -- but we do not care about using them to do anything
end Lin_Test_1;
```

SPARK is happy!

Second Instantiation: Item = String

```
-- 2nd type of items: fixed-length strings
type String_Type    is new String(1 .. 20);
type String_Arr     is array(Positive range <>) of String_Type;

-- comparing fixed-length strings
 function Same_String(Left, Right: String_Type)
                         return Boolean is
   (for all I in String_Type'Range =>
      Left(I) = Right(I));

function Lin_Search_Nam is new Search.Lin_Search
  (String_Type, String_Arr, "=" => Same_String);
```

SPARK is happy!

Defining a Record Type for Third Instantiation

```
-- 3rd type of items: some records
type Year_Type   is range 1930 .. 2030;
type Rec_Type is
   record
      Name:            String_Type;
      Year_Of_Birth:   Year_Type;
      Person_ID:       Natural;
   end record;
type Rec_Arr    is array(Positive range <>) of Rec_Type;
```

Third Instantiation: Item = Record

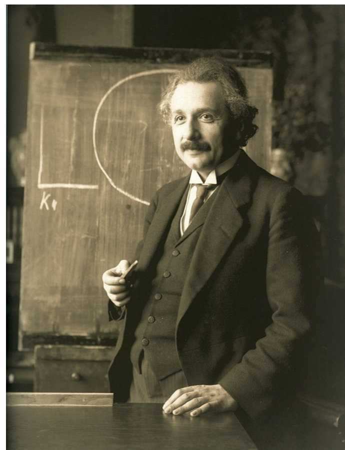
3(a): search for record with given ID
3(b): search for record with given Name

```
-- 3(a) comparison for Person_ID
-- (which should be a unique identifier)
function Equal_ID(Left, Right: Rec_Type) return Boolean is
   (Left.Person_ID = Right.Person_ID);

--3(b) comparison for Name
function Equal_Name(Left, Right: Rec_Type) return Boolean is
   (Same_String(Left.Name, Right.Name));

function Lin_Search_Rec_ID is new Search.Lin_Search
  (Rec_Type, Rec_Arr, "=" => Equal_ID);
function Lin_Search_Rec_Name is new Search.Lin_Search
  (Rec_Type, Rec_Arr, "=" => Equal_ID);
```

SPARK is happy!

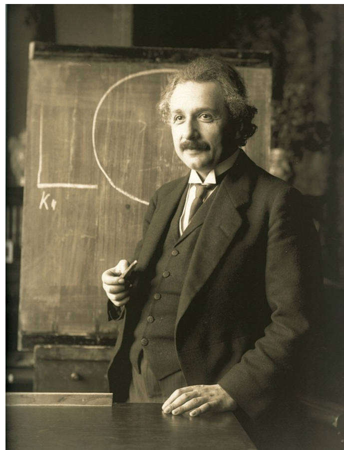
Clever Comments (8)

- All instantiations proved without problems.
- That was almost too easy!
- Perhaps, binary search is more of a challenge?



Clever Comments (8)

- All instantiations proved without problems.
- That was almost too easy!
- Perhaps, binary search is more of a challenge?



6.8: Generic Binary Search

- Generic Parameter type Item generic.
- Derived Parameter array (Positive range <>) of Item
- "="-operator for equality
- "<"-operator for less than
- for the sake of simplicity, avoid other operators

Generic Specification of Binary Search

```
generic
   type Item is limited private;
   type Arr is array (Positive range <>) of Item;
   with function "=" (Left, Right: Item) return Boolean is <>;
   with function "<" (Left, Right: Item) return Boolean is <>;
function Bin_Search (A: Arr; Value: Item) return Natural
  with Post =>
  (if A'First > A'Last or else Value < A(A'First) then
     Bin_Search'Result = No_Such_Index
  elsif A'First = A'Last or else A(A'First) = Value then
     Bin_Search'Result = A'First
  elsif Value < A(A'Last) then
     Bin_Search'Result > A'First and
     Bin_Search'Result <= A'Last and
     A(Bin_Search'Result-1) < Value and
     not (A(Bin_Search'Result) < Value)
  else Bin_Search'Result = A'Last
 );
```

For the comparison of objects of **type** Item, some trivial changes

Functions to use for "<"

```
function Smaller_String(Left, Right: String_Type)
                        return Boolean is
  (for some K in String_Type'Range =>
     Left(K) < Right(K) and then
     (for all I in String_Type'First .. K - 1 =>
        Left(I) = Right(I)));

function Smaller_ID(Left, Right: Rec_Type) return Boolean is
  (Left.Person_ID < Right.Person_ID);
```

First Instantiation: Item = Natural

This is fine – as expected!

```
function Bin_Search_Nat is new Search.Bin_Search -- line 14
  (Natural, Natural_Arr);
```

SPARK is happy!

Second Instantiation: Item = String

This fails – unexpectedly!

```
function Bin_Search_Nam is new Search.Bin_Search -- 16
  (String_Type, String_Arr,
   "=" => Same_String, "<" => Smaller_String);
```

SPARK is not happy:

postcondition might fail, cannot prove A(Bin_Search'Result−1) < Value, in instantiation at test_search.adb:16 (e.g. when A = (**others** => (**others** => 'NUL')) **and** A'First = 0 **and** A'Last = 0 **and** Bin_Search_NamR'Result = 0 **and** Value = (**others** => 'NUL'))

loop invariant might fail in first iteration, cannot prove A(Low) < Value, in instantiation at test_search.adb:16

Does SPARK also fail to prove the third instantiation?

Third Instantiation: Item is a record

I had considered this the most difficult one

```
function Bin_Search_Rec is new Search.Bin_Search -- 19
  (Rec_Type, Rec_Arr,
"=" => Equal_ID, "<" => Smaller_ID);
```

SPARK is happy!

How strange!

Why is SPARK unable to conclude A(Low) < Value at the beginning of the loop? And why is it able to draw the conclusion for Natural and Rec_Type?

```
while ... loop
  pragma Loop_Invariant
    (... and then (A(Low) < Value and not ...));
                   _______________
  ...
end loop;
```

loop invariant might fail in first iteration, cannot prove A(Low) < Value,

Why can't SPARK draw this conclusion?

```
if A'First > A'Last or else Value < A(A'First) then ...
elsif A'First = A'Last or else A(A'First) = Value then ...
elsif ... then
  Low := A'First;
  High := A'Last;
  while ... loop
    pragma Loop_Invariant
      (... and then (A(Low) < Value and not ...));
```

Before first iteration:

- **not** (Value < A(Low)), and
- **not** (A(Low) = Value).

Conclusion from trichotomy:

- A(Low) < Value.

Perhaps we should look at the comparison operators for strings?

The Comparison Operators for Strings

trichotomy: for $A, B \subseteq$ Item, either of $A = B$, $A < B$ or $B < A$ hold

```
function Same_String(Left, Right: String_Type)
                        return Boolean is
  (for all I in String_Type'Range =>
     Left(I) = Right(I));

function Smaller_String(Left, Right: String_Type)
                        return Boolean is
  (for some K in String_Type'Range =>
      Left(K) < Right(K) and then
      (for all I in String_Type'First .. K - 1 =>
         Left(I) = Right(I)));
```

Does trichotomy hold for "=" => Same_String and "<" => Smaller_String?

If "yes", can we prove that?

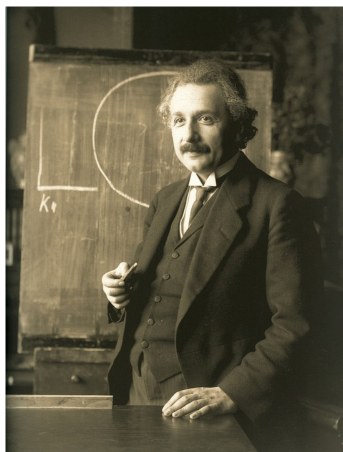
More Computational Power for the Prover

SPARK can prove everything. Finally!

```
project P is
  for Source_Dirs use (".");

  package Compiler is
    for Default_Switches ("Ada") use
         ("-gnat12", "-gnato13");
  end Compiler;

  package Prove is
   for Switches use
   ("--report=fail", "--proof=progressive",
    "--timeout=45", -- modified (was: timeout = 5)
    "--level=3"     -- new (default: level=1
   );
  end Prove;
end P;
```

Clever Comments (9)

- Failing to prove $\neq$ proof of a flaw!
- Proof strategy for generic algorithms:
  1. Start with structures over *simple types* (e.g., Natural),
  2. Later prove instantiations of *the same generic* based on increasingly complex types (records, strings, . . .).
- If the instantiations based on simple types succeed, and the more complex instantiatins fail, try again with more power (parameters `timeout` and `level`.
- You may even use an alternative theroem prover.
- Proving **array**-based instantiations can be demanding (cf. Hoare logic).



An Ideal Specification of Binary Search

```
generic
   type Item is limited private;
   type Arr is array (Positive range <>) of Item;
   with function "=" (Left, Right: Item) return Boolean is <>;
   with function "<" (Left, Right: Item) return Boolean is <>;
   with generic Pre =>
   ((for all A, B in Item => -- trichotomy
     (if A = B then not A < B) and not B < A
      elsif A < B then not B < A
      else B < A) and
     (for all A, B, C in Item => -- transitivity of = and <
      (if A = B and B = C then A = C) and
      (if A < B and B < C then A < C)));
function Bin_Search (A: Arr; Value: Item) return Natural
  with Post => ...;
```

Concluding Comments

how to write and prove sofware in SPARK

- Always *think* first, before coding!
- Try to simplify your specifications as much as possible.
- If SPARK fails to prove your code,
  1. try to find a flaw (if any)
  2. try to tell SPARK the "obvious" point, why your code is correct,
  3. try to figure out some SPARK constraint, or
  4. try to give SPARK more computational power.

  The counter-examples given by SPARK often help to identify cases 1.–3.
- If SPARK fails at a postcondition
  5. write a / rewrite the loop invariant.
- If you care about termination, write **for** loops or specify loop variants.
- Proving generics in SPARK (currently) goes instantiation-wise.