SPARK 2014 Introduction

Our requirement

- Given a global array A containing integers
- Given two integer X and Y, indices of A
- Return the length of the longest common prefix of the two sub-arrays starting at X and Y.

Example

3	4	5	0	9	4	5	0	9	2

```
LCP (X=2, Y=6) => 4
```

Assignment

The algorithm is implemented (it may have bugs). We are going to define the contracts the subprogram requires, and the contracts the subprogram must abide to. We are gonna write contracts for it in the Ada 2012 contracts syntax, and check those contracts with GNATprove. We're going to debug those contracts until they prove correct.

Here is the original code of the algorithm:

```
function LCP (X, Y : Integer) return Natural is
   L : Natural;
begin
   L := 0;
   while X + L <= A'Last
     and then Y + L <= A'Last
     and then A (X + L) = A (Y + L)
   loop
```

```
      L := L + 1;
   end loop;

   return L;
end LCP;
```

Pre-conditions

Reminder

A precondition is a Boolean expression to check before the call to a subprogram. It is a constraint to the caller, defining expectations under which a subprogram can be called in addition to parameter type constraints. It relates to the subprogram inputs, and can be composed of :

- Any visible name in the visible scope of the subprogram (even if defined afterwards)
- Any parameter of the subprogram

It is part of the specification of the subprogram. Here are some examples of pre-conditions :

```
function F (X : Integer) return Integer
  with Pre => X * X < 100;

procedure P (X : Integer; Y : Integer)
  with Pre => X + Y = 0 and then F (Y) /= 0;

procedure Some_Call
  with Pre => Initialized;

Initialized : Boolean := False;

function Some_Other_Call
  with Pre => Initialized;
```

Questions

The specification reads:

> Given two integers X and Y, indices of A

1. Write the pre-condition on Longest_Common_Prefix that corresponds to this statement
2. Write tests in main.adb that show that this pre-condition seems to fail and succeed when excepted
3. Use GNATprove to demonstrate that the precondition is indeed verified and not verified depending on the test

Post condition equality

Reminder

A post-condition is a Boolean expression to check after the call to a subprogram. It is a constraint to the implementer, defining expectations on the subprogram behavior and effect. It relates to the subprogram outputs. It can be composed of:

- Any visible name in the visible scope of the subprogram (even if defined afterwards)
- Any parameter of the subprogram
- The values of any of the above before the call to the subprogram
- The result of a subprogram

It is part of the specification. Here are some examples of post conditions:

```ada
function F (X : Integer) return Integer
with Post => F'Result < 100;

procedure P (X : out Integer; Y : out Integer)
with Post => X + Y = 0 and then F (Y) /= 0;

procedure Initialize
with Post => Initialized;

Initialized : Boolean := False;

function Reset
with Post => not Initialized;
```

Questions

From the requirements, we can set the following post-condition:

The next `LCP’Result` values of `A` following `X` are equal to the next `LCP’Result` values of `A` following `Y`

1. Write the post-condition for the above
2. Run tests in main.adb to verify that this post-condition seems correct
3. Use GNATprove to prove the post-condition

Loop invariant

Reminder

A quantifier is a construction used to define a property true for an entire set, or at least one element of the set

```
type A is array (Integer range <>) of Integer;

V  : A := (10, 20, 30);
B1 : Boolean := (for all J in V’Range => V (J) >= 10);  -- True
B2 : Boolean := (for some J in V’Range => V (J) >= 20); -- True
```

It may be used in contracts, either for static or dynamic checks

```
type A is array (Integer range <>) of Integer
   with Dynamic_Predicate =>
      (A'Length <= 1 or else
         (for all J in A'First + 1 .. A'Last => A (J - 1) < A (J));
```

A loop invariant is a condition that is always true in the loop. It is used to prove the postcondition. It can’t be created by the prover, but once specified it is proved by the prover. It can also be checked at run-time. Here is an example :

```
while A (X) /= 0 loop
   pragma Loop_Invariant
      (for all I in A'First .. X => A (I) /= 0);

   X := X + 1;
end loop;
```

Questions

From the requirements, we can deduce the following invariant

> At each iteration, the prefix going from `X` to `X + L` is equal to the prefix from `Y` to `Y + L`

1. Write the invariant in the Longest_Common_Prefix loop
2. Run tests in main.adb to verify that this invariant seems correct
3. Use GNATprove to prove the invariant

Loop variant

Reminder

A loop variant is used to determine the termination of a loop. It's based on the verification that the loop contains a (bounded) discrete type that always increases and decreases, and does not get out of bounds. It can't be created by the prover, but once specified it is proved by the prover. It can be checked at run-time

```
while A (X) /= 0 loop
   pragma Loop_Variant (Increases => X);

   X := X + 1;
end loop;
```

Questions

From the requirements, we can deduce the following variant:

> At each iteration, the value of `L` should increase

1. Write the variant in the Longest_Common_Prefix loop
2. Run tests in main.adb to verify that this variant seems correct
3. Use GNATprove to prove the variant

Contract cases

Reminder

A contract case can be used to link an input condition to an output condition. The input of the subprogram as defined by the pre-condition must be completely covered (others => will allow to consider remaining cases). The contract cases must not overlap. The above is checked either by the prover, or on the input provided at test-time.

Also, for conditions that are checked after the program execution, one can use the `'Result` and `'Old` attributes to access respectively the result of a subprogram if it is a function, and the value of a parameter or global before the execution of its body.

Here is an example of contract cases:

```
function F (X, Y : Natural) return Integer
   with Contract_Cases =>
    (X = 0 and then Y = 0 => F'Result => 0,
     X > 0 and then Y = 0 => F'Result => X,
     X = 0 and then Y > 0 => F'Result => Y,
     others => F'Result => X + Y);
```

Questions

From the requirements, we can set the following conditions:

> If `A(X) /= A(Y)`, result is `0`
>
> If `X = Y`, result is the length from `X` to last
>
> Otherwise, result is above `0`

1. Write the contract cases for the above
2. Run tests in main.adb to verify that these contracts seem correct
3. Use GNATprove to prove the contract cases

Post-condition completeness

Questions

From the requirements, one part of the post-condition is missing, the fact that the prefix is indeed the biggest one:

> Either `X + LCP'Result` or `Y + LCP'Result` is beyond `A'Last`, or the element at `X + LCP'Result` is different from the one at `Y + LCP'Result`

1. Complete the post-condition for the above
2. Run tests in main.adb to verify that this post-condition seems correct
3. Use GNATprove to prove the post-condition