SPARK 2014: Ghost Code

University.adacore.com

What is ghost code?

ghost code is part of the program that
is added for the purpose of specification

Why3 team, "The Spirit of Ghost Code"

… or verification

addition by SPARK team

Examples of ghost code:

- contracts (Pre, Post, Contract_Cases, etc.)
- assertions (pragma Assert, loop (in)variants, etc.)
- special values Func'Result, Var'Old, Var'Loop_Entry

Is it enough?

Ghost code – A trivial example

```
Data : Data_Array;
Free : Natural;

procedure Alloc is
begin
   -- some computations here
   assert that Free "increases"
end Alloc;
```

✖ how to express it?

Ghost variables – aka auxiliary variables

- **Variables declared with aspect Ghost**
  - declaration is discarded by compiler when ghost code ignored

- **Ghost assignments to ghost variables**
  - assignment is discarded by compiler when ghost code ignored

```
Data : Data_Array;
Free : Natural;

procedure Alloc is
   Free_Init : Natural with Ghost;
begin
   Free_Init := Free;
   -- some computations here
   pragma Assert (Free > Free_Init);
end Alloc;
```

Ghost variables – non-interference rules

- **Ghost variable cannot be assigned to non-ghost one**

```
Free := Free_Init;
```

- **Ghost variable cannot indirectly influence assignment to non-ghost one**

```
if Free_Init < Max then
   Free := Free + 1;
end if;
```

```
procedure Assign (From : Natural; To : out Natural) is
begin
   To := From;
end Assign;

Assign (From => Free_Init, To => Free);
```

Ghost statements

- **Ghost variables can only appear in ghost statements**
  - assignments to ghost variables
  - assertions and contracts
  - calls to ghost procedures

```
procedure Assign (From : Natural; To : out Natural)
   with Ghost
is
begin
   To := From;
end Assign;

Assign (From => Free, To => Free_Init);

Assign (From => Free_Init, To => Free);
```

Ghost procedures

- **Ghost procedures cannot write into non-ghost variables**

```
procedure Assign (Value : Natural) with Ghost is
begin
   Free := Value;
end Assign;
```

- **Used to group statements on ghost variables**
  - in particular statements not allowed in non-ghost procedures

```
procedure Assign_Cond (Value : Natural) with Ghost is
begin
   if Condition then
      Free_Init := Value;
   end if;
end Assign_Cond;
```

- **Can have Global (including Proof_In) & Depends contracts**

Ghost functions

- **Functions for queries used only in contracts**

```
procedure Alloc with
   Pre  => Free_Memory > 0,
   Post => Free_Memory < Free_Memory'Old;

function Free_Memory return Natural with Ghost;
```

- **Typically implemented as expression functions**
  - in private part – proof of client code can use expression
  - or in body – only proof of unit can use expression

```
function Free_Memory return Natural is (…);
-- if completion of ghost function declaration

function Free_Memory return Natural is (…) with Ghost;
-- if function body as declaration
```

Imported ghost functions

- **Ghost functions without a body**
  - cannot be executed

```
function Free_Memory return Natural with Ghost, Import;
```

- **Typically used with abstract ghost private types**
  - definition in SPARK_Mode(Off) → type is abstract for GNATprove

```
   type Memory_Chunks is private;
   function Free_Memory return Memory_Chunks
     with Ghost, Import;
private
   pragma SPARK_Mode (Off);
   type Memory_Chunks is null record;
```

- **Definition of ghost types/functions given in proof**
  - either in Why3 using External_Axiomatization
  - or in an interactive prover (Coq, Isabelle, etc.)

Ghost packages and ghost abstract state

- **Every entity in a ghost package is ghost**
  - local ghost package can group all ghost entities
  - library-level ghost package can be withed/used in regular units

- **Ghost abstract state can only represent ghost variables**

```
package Mem with
   Abstract_State => (State with Ghost)
is
```

```
package body Mem with
   Refined_State => (State => (Data, Free, Free_Init))
is
```

- **Non-ghost abstract state can contain both ghost and non-ghost variables**

Executing ghost code

- **Ghost code can be enabled globally**
  - using compilation switch --gnata (for all assertions)
- **Ghost code can be enabled selectively**
  - using pragma Assertion_Policy (Ghost => Check)
  - SPARK rules enforce consistency – in particular no write disabled

```
pragma Assertion_Policy (Ghost => Ignore, Pre => Check);

procedure Alloc with
   Pre => Free_Memory > 0;

function Free_Memory return Natural with Ghost;
```

- **GNATprove analyzes all ghost code and assertions**

Example of use – encoding a state automaton

- **Tetris in SPARK**
  - at http://blog.adacore.com/tetris-in-spark-on-arm-cortex-m4

- **Global state encoded in global ghost variable**
  - updated at the end of procedures of the API

```
type State is (Piece_Falling, …) with Ghost;
Cur_State : State with Ghost;
```

- **Properties encoded in ghost functions**

```
function Valid_Configuration return Boolean is
    (case Cur_State is
        when Piece_Falling => …,
        when …)
with Ghost;
```

Example of use – expressing useful lemmas

- **GCD in SPARK**
  - at http://www.spark-2014.org/entries/detail/gnatprove-tips-and-tricks-proving-the-ghost-common-denominator-gcd

- **Lemmas expressed as ghost procedures**

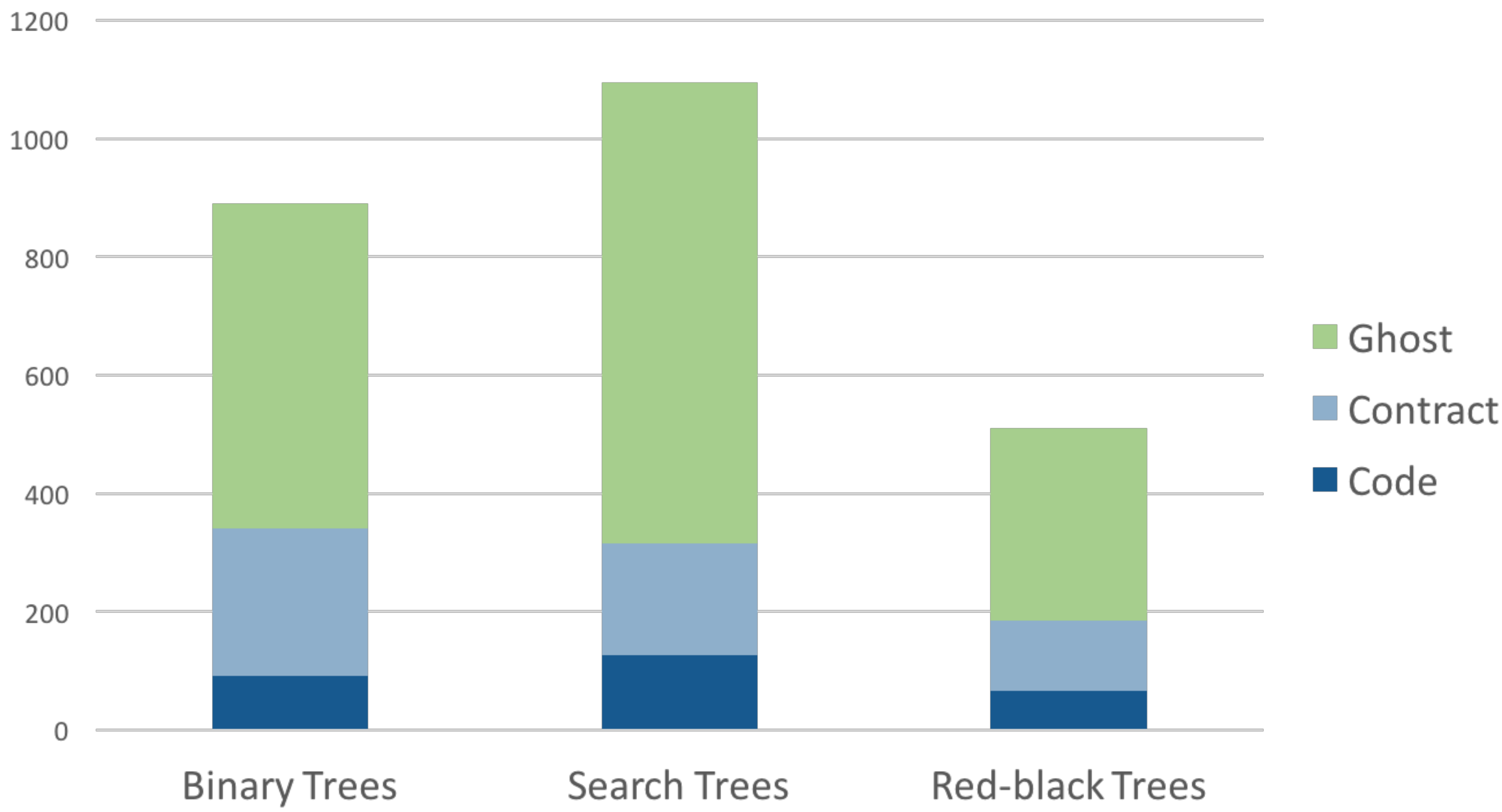
```
procedure Lemma_Not_Divisor (Arg1, Arg2 : Positive) with
   Ghost,
   Global => null,
   Pre  => Arg1 in Arg2 / 2 + 1 .. Arg2 - 1,
   Post => not Divides (Arg1, Arg2);
```

- **Most complex lemmas further refined into other lemmas**
  - code in procedure body used to guide proof (e.g. for induction)

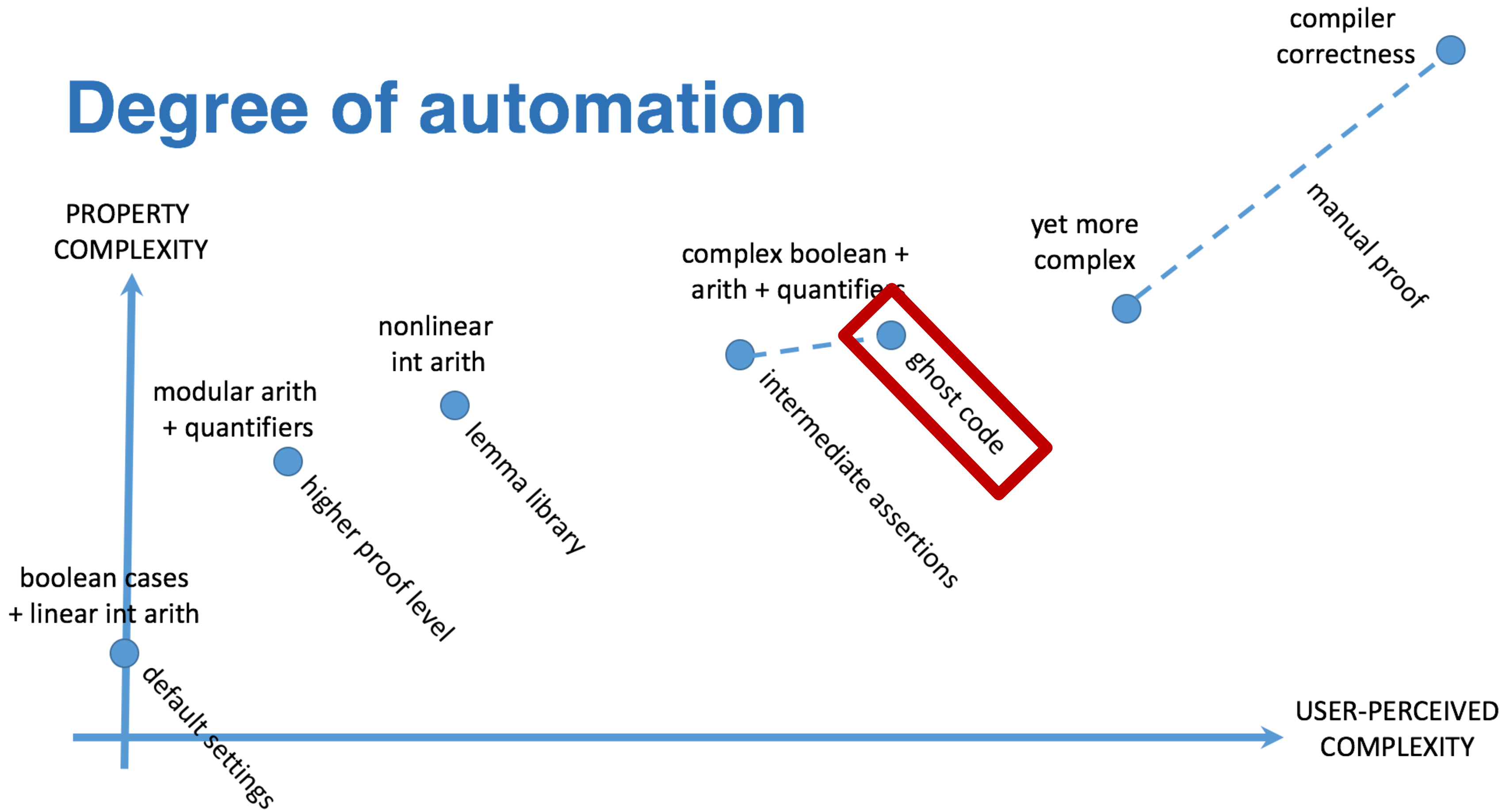
Example of use – specifying an API through a model

- **Red black trees in SPARK**
  - at http://www.spark-2014.org/entries/detail/research-corner-auto-active-verification-in-spark
- **Invariants of data structures expressed as ghost functions**
  - using Type_Invariant on private types
- **Model of data structures expressed as ghost functions**
  - called from Pre/Post of subprograms from the API
- **Lemmas expressed as ghost procedures**
  - sometimes without contracts to benefit from inlining in proof

Extreme proving with ghost code – red black trees in SPARK



Positioning ghost code in proof techniques



Quiz

Is this correct? 1/10

YES (click on the check icon)

NO (click on the error location(s))

```
Data : Data_Array;
Free : Natural;

procedure Alloc is
   Free_Init : Natural with Ghost;
begin
   Free_Init := Free;
   -- some computations here
   if Free <= Free_Init then
      raise Program_Error;
   end if;
end Alloc;
```

Is this correct? 1/10

NO

```
Data : Data_Array;
Free : Natural;

procedure Alloc is
   Free_Init : Natural with Ghost;
begin
   Free_Init := Free;
   -- some computations here
   if Free <= Free_Init then
      raise Program_Error;
   end if;
end Alloc;
```

ghost entity cannot appear in this context

Is this correct? 2/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
Data : Data_Array;
Free : Natural;

procedure Alloc is
   Free_Init : Natural with Ghost;

   procedure Check with Ghost is
   begin
      if Free <= Free_Init then
         raise Program_Error;
      end if;
   end Check;
begin
   Free_Init := Free;
   -- some computations here
   Check;
end Alloc;
```

Is this correct? 2/10

YES

```
Data : Data_Array;
Free : Natural;

procedure Alloc is
   Free_Init : Natural with Ghost;

   procedure Check with Ghost is
   begin
      if Free <= Free_Init then
         raise Program_Error;
      end if;
   end Check;
begin
   Free_Init := Free;
   -- some computations here
   Check;
end Alloc;
```

Note that procedure Check is inlined for proof (no contract).

Is this correct? 3/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
pragma Assertion_Policy (Pre => Check);

procedure Alloc with
   Pre => Free_Memory > 0;

function Free_Memory return Natural with Ghost;
```

Is this correct? 3/10

NO

```
pragma Assertion_Policy (Pre => Check);

procedure Alloc with
   Pre => Free_Memory > 0;

function Free_Memory return Natural with Ghost;
```

Incompatible ghost policies in effect during compilation, as ghost code is ignored by default.

Note that GNATprove accepts this code as it enables all ghost code and assertions.

Is this correct? 4/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
procedure Alloc with
   Post => Free_Memory < Free_Memory'Old;

function Free_Memory return Natural with Ghost;
```

```
Max : constant := 1000;

function Free_Memory return Natural is
begin
   return Max - Free + 1;
end Free_Memory;

procedure Alloc is
begin
   Free := Free + 10;
end Alloc;
```

Is this correct? 4/10

NO

```
procedure Alloc with
   Post => Free_Memory < Free_Memory'Old;

function Free_Memory return Natural with Ghost;
```

```
Max : constant := 1000;

function Free_Memory return Natural is
begin
   return Max - Free + 1;
end Free_Memory;

procedure Alloc is
begin
   Free := Free + 10;
end Alloc;
```

No postcondition on Free_Memory that would allow proving the postcondition on Alloc.

Is this correct? 5/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
procedure Alloc with
   Post => Free_Memory < Free_Memory’Old;

function Free_Memory return Natural with Ghost;
```

```
Max : constant := 1000;

function Free_Memory return Natural is (Max - Free + 1);

procedure Alloc is
begin
   Free := Free + 10;
end Alloc;
```

Is this correct? 5/10

YES

```
procedure Alloc with
   Post => Free_Memory < Free_Memory'Old;

function Free_Memory return Natural with Ghost;
```

```
Max : constant := 1000;

function Free_Memory return Natural is (Max - Free + 1);

procedure Alloc is
begin
   Free := Free + 10;
end Alloc;
```

Free_Memory has an implicit postcondition as an expression function.

Is this correct? 6/10

YES
(click on the check icon)

NO
(click on the error location(s))

```ada
subtype Resource is Natural range 0 .. 1000;
subtype Num is Natural range 0 .. 6;
subtype Index is Num range 1 .. 6;
type Data is array (Index) of Resource;

function Sum (D : Data; To : Num) return Natural is
  (if To = 0 then 0 else D(To) + Sum(D,To-1))
with Ghost;

procedure Create (D : out Data) with
   Post => Sum (D, D'Last) < 42
is
begin
   for J in D'Range loop
      D(J) := J;
      pragma Loop_Invariant (2 * Sum(D,J) <= J * (J+1));
   end loop;
end Create;
```

Is this correct? 6/10

NO

```
subtype Resource is Natural range 0 .. 1000;
subtype Num is Natural range 0 .. 6;
subtype Index is Num range 1 .. 6;
type Data is array (Index) of Resource;

function Sum (D : Data; To : Num) return Natural is
  (if To = 0 then 0 else D(To) + Sum(D,To-1))
with Ghost;

procedure Create (D : out Data) with
   Post => Sum (D, D'Last) < 42
is
begin
  for J in D'Range loop
     D(J) := J;
     pragma Loop_Invariant (2 * Sum(D,J) <= J * (J+1));
  end loop;
end Create;
```

info: expression function body not available for proof ("Sum" may not return)

Is this correct? 7/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
subtype Resource is Natural range 0 .. 1000;
subtype Num is Natural range 0 .. 6;
subtype Index is Num range 1 .. 6;
type Data is array (Index) of Resource;

function Sum (D : Data; To : Num) return Natural is
   (if To = 0 then 0 else D(To) + Sum(D,To-1))
with Ghost, Annotate => (GNATprove, Terminating);

procedure Create (D : out Data) with
    Post => Sum (D, D'Last) < 42
is
begin
   for J in D'Range loop
       D(J) := J;
      pragma Loop_Invariant (2 * Sum(D,J) <= J * (J+1));
    end loop;
end Create;
```

Is this correct? 7/10

YES

```
subtype Resource is Natural range 0 .. 1000;
subtype Num is Natural range 0 .. 6;
subtype Index is Num range 1 .. 6;
type Data is array (Index) of Resource;

function Sum (D : Data; To : Num) return Natural is
  (if To = 0 then 0 else D(To) + Sum(D,To-1))
with Ghost, Annotate => (GNATprove, Terminating);

procedure Create (D : out Data) with
   Post => Sum (D, D'Last) < 42
is
begin
   for J in D'Range loop
      D(J) := J;
      pragma Loop_Invariant (2 * Sum(D,J) <= J * (J+1));
   end loop;
end Create;
```

Note that GNATprove does not prove the termination of Sum here.

Is this correct? 8/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
subtype Resource is Natural range 0 .. 1000;
subtype Num is Natural range 0 .. 6;
subtype Index is Num range 1 .. 6;
type Data is array (Index) of Resource;

function Sum (D : Data; To : Num) return Natural is
  (if To = 0 then 0 else D(To) + Sum(D,To-1))
with Ghost, Annotate => (GNATprove, Terminating);

procedure Create (D : out Data) with
   Post => Sum (D, D'Last) < 42
is
begin
  for J in D'Range loop
     D(J) := J;
   end loop;
end Create;
```

Is this correct? 8/10

YES

```
subtype Resource is Natural range 0 .. 1000;
subtype Num is Natural range 0 .. 6;
subtype Index is Num range 1 .. 6;
type Data is array (Index) of Resource;

function Sum (D : Data; To : Num) return Natural is
  (if To = 0 then 0 else D(To) + Sum(D,To-1))
with Ghost, Annotate => (GNATprove, Terminating);

procedure Create (D : out Data) with
   Post => Sum (D, D'Last) < 42
is
begin
  for J in D'Range loop
     D(J) := J;
   end loop;
end Create;
```

The loop is unrolled by GNATprove here, as D'Range is 0..6.
The automatic prover unrolls the recursive definition of Sum.

Is this correct? 9/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
subtype Resource is Natural range 0 .. 1000;
subtype Index is Natural range 1 .. 42;

package Seqs is new
  Ada.Containers.Functional_Vectors (Index, Resource);
use Seqs;

function Create return Sequence with
   Post => (for all K in 1 .. Last (Create'Result) =>
              Get (Create'Result, K) = K)
is
   S : Sequence;
begin
   for K in 1 .. 42 loop
      S := Add (S, K);
   end loop;
   return S;
end Create;
```

Is this correct? 9/10

NO

```
subtype Resource is Natural range 0 .. 1000;
subtype Index is Natural range 1 .. 42;

package Seqs is new
  Ada.Containers.Functional_Vectors (Index, Resource);
use Seqs;

function Create return Sequence with
   Post => (for all K in 1 .. Last (Create'Result) =>
               Get (Create'Result, K) = K)
is
   S : Sequence;
begin
   for K in 1 .. 42 loop
      S := Add (S, K);
   end loop;
   return S;
end Create;
```

Loop requires a loop invariant to prove the postcondition.

Is this correct? 10/10

YES
(click on the check icon)

NO
(click on the error location(s))

```
subtype Resource is Natural range 0 .. 1000;
subtype Index is Natural range 1 .. 42;

package Seqs is new
  Ada.Containers.Functional_Vectors (Index, Resource);
use Seqs;

function Create return Sequence with
   Post => (for all K in 1 .. Last (Create'Result) =>
              Get (Create'Result, K) = K)
is
   S : Sequence;
begin
   for K in 1 .. 42 loop
      S := Add (S, K);
      pragma Loop_Invariant (Integer (Length (S)) = K);
      pragma Loop_Invariant
        (for all J in 1 .. K => Get (S, J) = J);
   end loop;
   return S;
end Create;
```

Is this correct? 10/10

YES

```
subtype Resource is Natural range 0 .. 1000;
subtype Index is Natural range 1 .. 42;

package Seqs is new
  Ada.Containers.Functional_Vectors (Index, Resource);
use Seqs;

function Create return Sequence with
   Post => (for all K in 1 .. Last (Create'Result) =>
               Get (Create'Result, K) = K)
is
   S : Sequence;
begin
   for K in 1 .. 42 loop
      S := Add (S, K);
      pragma Loop_Invariant (Integer (Length (S)) = K);
      pragma Loop_Invariant
        (for all J in 1 .. K => Get (S, J) = J);
   end loop;
   return S;
end Create;
```

university.adacore.com