Enhance Verification using Ghost Code

Claire Dross

SSAS Workshop 2018
Ghost Code, What Is It?
• Ghost code does not affect normal execution of a program.
Ghost Code – General Definition

- Ghost code does not affect normal execution of a program.
- It is used to monitor execution (can terminate the program).
Ghost Code – General Definition

• Ghost code does not affect normal execution of a program.
• It is used to monitor execution (can terminate the program).
• Example: assertions in code / subprogram contracts

```
pragma Assert (X /= 0);
--   Runtime exception: raised Assert_Failure - failed assertion

procedure Increment (X : in out Integer) with
  Pre  => X < Integer’Last,
  Post => X = X’Old + 1;

Increment (X);
--   Runtime exception: raised Assert_Failure - failed precondition
```
Ghost Code in SPARK

• In SPARK, all entities (variables, subprograms, types...) can be ghost.

```plaintext
procedure Do_Something (X : in out T) is
  X_Init : constant T := X with Ghost;
begin
  Do_Some_Complex_Stuff (X);
  pragma Assert (Transformation_Is_Correct (X_Init, X));
  -- It is OK to use X_Init inside an assertion.
```

---

In SPARK, all entities (variables, subprograms, types...) can be ghost.
Ghost Code in SPARK

• In SPARK, all entities (variables, subprograms, types...) can be ghost.
• The compiler detects most incorrect usage.

```plaintext
procedure Do_Something (X : in out T) is
  X_Init : constant T := X with Ghost;
begin
  Do_Some_Complex_Stuff (X);
  pragma Assert (Transformation_Is_Correct (X_Init, X));
  -- It is OK to use X_Init inside an assertion.

  X := X_Init;
  -- Compilation error:
  -- Ghost entity cannot appear in this context.
```
Ghost Code in SPARK – Execution

• Ghost code can be executed like normal code ...

Regular Code

Ghost Code

Compilation with assertions enabled
Ghost code can be executed like normal code ...

... or can be removed at compilation.
Ghost Code in SPARK – Verification

- Static verification applies to regular code + ghost code.
Ghost Code in SPARK – Verification

- Static verification applies to regular code + ghost code.
- SPARK also verifies that ghost does not affect regular code.
Ghost Code in SPARK – Verification

- Static verification applies to regular code + ghost code.
- SPARK also verifies that ghost does not affect regular code.

→ Regular code is verified.
Enhance Expressiveness in Specifications
Speciation-Only Functions

- Ghost functions are used to factor out expressions in contracts.

```haskell
function Sort (A : in out Nat_Array) with
  Post => Is_Sorted (A) and then Is_Permutation (A, A'Old);

function Is_Sorted (A : Nat_Array) return Boolean is
  (for all I in A'Range =>
    if I > A'First then A (I) >= A (I - 1)))
  with Ghost;

function Search (A : Nat_Array; E : Natural) return Index with
  Pre => Is_Sorted (A);
```
• Ghost functions are used to factor out expressions in contracts.
• They can disclose state abstractions for specification purposes.

```package Private_Counter is
  function Disclose_Content return Natural with Ghost;
  function Is_Max return Boolean with
    Post => Is_Max’Result = (Disclose_Content = Max);
  procedure Incr with
    Pre => not Is_Max;
    Post => Disclose_Content = Disclose_Content’Old + 1;
  private
    Counter_Value : Natural := 0;
end Private_Counter;
```
 Specification-Only Functions

- Ghost functions are used to factor out expressions in contracts.
- They can disclose state abstractions for specification purposes.
- Inefficient is OK if assertions are disabled in the final executable.

```plaintext
function Occurrences (A : Nat_Array; E : Natural) return Natural;

function Is_Permutation (A, B : Nat_Array) return Boolean is
    (for all E in Natural => Occurrences (A, E) = Occurrences (B, E))
with Ghost;
```
• Ghost variables can be used to store intermediate values of variables.

```plaintext
X_Interm : T with Ghost;

procedure Do_Two_Thing (X : in out T) with
  Post => First_Thing_Done (X’Old, X_Interm) and then
  Second_Thing_Done (X_Interm, X)
is
  X_Init : constant T := X with Ghost;
begin
  Do_Something (X);
  pragma Assert (First_Thing_Done (X_Init, X));
  X_Interm := X;
  Do_Something_Else (X);
  pragma Assert (Second_Thing_Done (X_Interm, X));
end Do_Two_Things;
```
Specifying-Only Data

- Ghost variables can be used to store intermediate values of variables.
- Some properties are best expressed by constructing a witness.

```ada
Perm : Permutation with Ghost;
procedure Perm_Sort (A : Nat_Array) with
  Post => A = Apply_Perm (Perm, A'Old)
is
begin
  Perm := Identity_Perm;
  for Current in A'First .. A'Last - 1 loop
    Smallest := Index_Of_Minimum (A, Current, A'Last);
    if Smallest /= Current then
      Swap (A, Current, Smallest);
      Permute (Perm, Current, Smallest);
    end if;
  end loop;
end Perm_Sort;
```
• Ghost variables can be used to store intermediate values of variables.
• Some properties are best expressed by constructing a witness.
• Ghost variables can also store interprocedural information.

```plaintext
History : Buffer_Of_Bool (1 .. 2) with Ghost;

procedure Count_To_Three (Is_Third : out Boolean) with Post =>
  Is_Third = (not Last_Value (History'Old)
           and then not Before_Last_Value (History'Old))
           and then History = Enqueue (History'Old, Is_Third);```

specification-only data
• Ghost variable can also model interprocedural control flow.

```plaintext
Last_Accessed_Is_A : Boolean := False with Ghost;

procedure Access_A with
    Post => Last_Accessed_Is_A;

procedure Access_B with
    Pre  => Last_Accessed_Is_A,
    Post => not Last_Accessed_Is_A;
```
• Ghost variable can also model interprocedural control flow.
• More generally, expected control flow can be expressed as an automaton.

```plaintext
type State_Kind is (S1, S2, S3) with Ghost;
State : State_Kind := S1 with Ghost;

procedure Access_A with
  Pre       => State in S1 | S3,
Contract_Cases =>
  (State = S1 => State = S2,
   State = S3 => State = S3);

procedure Access_B with
  Pre => State in S2 | S3,
  Post => State = S3;
```

Models of Control Flow
Models of Control Flow

- Ghost variable can also model interprocedural control flow.
- More generally, expected control flow can be expressed as an automaton.
- An invariant can link the ghost and regular states.

```pascal
type Mailbox_Status_Kind is (Empty, Full) with Ghost;
Mailbox_Status : Mailbox_Status_Kind := Empty with Ghost;

function Invariant return Boolean is
  (if Mailbox_Status = Full then Valid (Message_Content)) with Ghost;

procedure Receive with
  Pre => Invariant and then Mailbox_Status = Full,
  Post => Invariant and then Mailbox_Status = Empty;
```
Models of Data Structures

• A model is an alternative view of a data structure.

A ring buffer

Its model: an array
Models of Data Structures

• A model is an alternative view of a data structure.
• They are typically simpler and less efficient.

A ring buffer

Its model: an array
A model is an alternative view of a data structure.
They are typically simpler and less efficient.
They can be stored in global variables or computed through a function.

Buffer_Content : Nat_Array;
Buffer_Top : Natural;
Buffer_Model : Nat_Array with Ghost;

procedure Enqueue (E : Natural) with
   Post => Buffer_Model = E & Buffer_Model’Old (1 .. Max - 1);
Models of Data Structures

- A model is an alternative view of a data structure.
- They are typically simpler and less efficient.
- They can be stored in global variables or computed through a function.

```ada
type Buffer_Type is record …;
subtype Model_Type is Nat_Array with Ghost;
function Get_Model (B : Buffer_Type) return Model_Type with Ghost;
procedure Enqueue (B : Buffer_Type, E : Natural) with
    Post => Get_Model (B) = E & Get_Model (B)’Old (1 .. Max – 1);
```
Guide the Proof Tool
Intermediate assertions can help the tool.

```plaintext
pragma Assert (Complex_ASSERTion);
```
Guide the Proof Tool

• Intermediate assertions can help the tool.

```plaintext
pragma Assert (Intermediate_Assertion_1);
pragma Assert (Intermediate_Assertion_2);
pragma Assert (Complex_Assertion);
```
Intermediate assertions can help the tool.

\texttt{pragma Assert (Intermediate\_Assertion\_1);}
\texttt{pragma Assert (Intermediate\_Assertion\_2);}
\texttt{pragma Assert (Complex\_Assertion);}
• Proving an existential quantifier is difficult for provers.

`pragma Assert (A (A’First) = 0 and then A (A’Last) > 0);`

`pragma Assert`

`     (for some I in A’Range =>`

`           I < A’Last and then A (I) = 0 and then A (I + 1) > 0);`
• Proving an existential quantifier is difficult for provers.
• A witness can be constructed and provided.

```
function Find_Pos (A : Nat_Array) return Positive with Ghost,
  Pre  => A (A’First) = 0 and then A (A’Last) > 0,
  Post => Find_Pos’Result in A’First .. A’Last - 1 and then
          A (Find_Pos’Result) = 0 and then A (Find_Pos’Result + 1) > 0;
pragma Assert (A (A’First) = 0 and then A (A’Last) > 0);
pragma Assert (Find_Pos (A) in A’Range);
pragma Assert
  (for some I in A’Range =>
   I < A’Last and then A (I) = 0 and then A (I + 1) > 0);
```
Guide the Proof Tool – Proof by Induction

• Provers mostly can’t perform induction.

```vhdl
pragma Assert
(for all I in A'Range =>
  (if I > A'First then A (I) > A (I - 1)));
```

```vhdl
pragma Assert
(for all I in A'Range =>
  (for all J in A'Range =>
    (if I > J then A (I) > A (J))));
```

Guide the Proof Tool – Proof by Induction

• Provers mostly can’t perform induction.
• Loop invariants allow to perform induction.

```plaintext
procedure Prove_Sorted (A : Nat_Array) with Ghost is
begin
    for K in 0 .. A’Length loop
        pragma Loop_Invariant
            (for all I in A’Range => (for all J in A’Range =>
                (if I > J and then I - J <= K then A(I) > A(J)))));
    end loop;
    pragma Assert (for all I in A’Range =>
        (for all J in A’Range => (if I > J then A(I) > A(J))));
end Prove_Sorted;
```
Guide the Proof Tool – Lemmas

• Procedures for lemmas have a contract but no effects.

```plaintext
procedure Prove_Sorted (A : Nat_Array) with Ghost,
  Pre  => (for all I in A’Range =>
    (if I > A’First then A (I) > A (I - 1))),
Post => (for all I in A’Range =>
  (for all J in A’Range =>
    (if I > J then A (I) > A (J))));
```
Guide the Proof Tool – Lemmas

• Procedures for lemmas have a contract but no effects.
• They must be called manually to assume the lemma.

```haskell
pragma Assert
  (for all I in A’Range =>
   (if I > A’First then A (I) > A (I - 1)));

Prove_Sorted (A);
-- Precondition of Prove_Sorted is proved

pragma Assert
  (for all I in A’Range =>
   (for all J in A’Range => (if I > J then A (I) > A (J))));
```
Guide the Proof Tool – Lemmas

• Procedures for lemmas have a contract but no effects.
• They must be called manually to assume the lemma.
• A lemma library is provided with SPARK for classical lemmas.

```haskell
procedure Lemma_Div_Is_Monotonic
    (Val1 : Int;
     Val2 : Int;
     Denom : Pos)
with Ghost,
    Pre  => Val1 <= Val2,
    Post => Val1 / Denom <= Val2 / Denom;
--  Proven manually using Coq
```
Conclusion
An Everyday Tool for Formal Verification

• miTLS\(^1\) and HACL\(^2\): TLS layer protocol and cryptographic functions
  • Pure ghost specification in F\(^*\)

• Ironclad and IronFleet\(^3\): Verifying distributed systems
  • Ghost safety specification using Dafny

• Imperative red-black trees in SPARK\(^4\)
  • Multi-layer ghost specification and ghost proofs

What Ghost Code Can Do for You

• Ghost code provides provably non-interfering instrumentation.

• Ghost code can enhance expressiveness of the specification.

• Ghost code can be used for static or dynamic verification.

• Ghost code can guide the proof tool.

• Ghost code is the bridge between automatic and interactive verification.