Beyond Functional Properties

Johannes Kanig, AdaCore

Hi-Lite annual meeting - May 10th, 2011

AdaCore
The GNAT Pro Company

Overview

Specifications can contain errors, too

- Assertions may contain run-time errors
- valid contracts may be meaningless or unhelpful
- valid contracts may not adequately summarize a subprogram

additional features that help write correct contracts

- Absence of run-time errors in assertions (in progress)
- "Semantic Dead Code" (not implemented)
- Detection of inconsistent preconditions (not implemented)

Assertions can contain run-time errors themselves

A principle of Hi-Lite

Proofs adopt the executable semantics of assertions

A question ...

What is the meaning of an assertion that raises a run-time error?

Our answer

It's the wrong question: assertions should never do that.

One goal of GNATprove

Prove the absence of run-time errors in programs *and* assertions

Assertions generate additional checks

Given the type definitions:

```
type Array_Range is range 1 .. 10;
type IntArray is array (Array_Range) of Integer;
```

The following assertion will require an additional check:

```
for Index in Table'Range loop
   --  This will generate a (provable) check:
   --     J in Table'Range
   pragma Assert
      (for all J in Table'First .. Index - 1 =>
         Table (J) /= Value);
   ...
end loop;
```

AdaCore
The GNAT Pro Company

Preconditions must be self-guarded

Preconditions

- are treated as any other assertion;
- but cannot use any context

Wrong:

```
procedure P (X : IntArray; I : Integer)
   with Pre => (X (I) > 0);
```

Correct:

```
procedure P (X : IntArray; I : Integer)
   with Pre => (I in X'Range and then X (I) > 0);
```

A precondition must always contain all guards that guarantee run-time error free execution

An Alternative - Adding implicit checks

Accept:

```
procedure P (X : IntArray; I : Integer)
   with Pre => (X (I) > 0);
```

In the body of P, we assume I in X'Range.

But insert the check at every *call*:

```
--  Generates two checks:
--     I in X'Range and then X(I) > 0
P (X, I);
```

At the call site, more context is available to prove the checks

In Hi-Lite we choose the first variant

- Requires the programmer to write the check down;
- Does not add any implicit assumptions;
- Makes a subprogram declaration self-contained.

Semantic dead code

Goal: improve postconditions

Detect situations where the postcondition is correct, but:

- The postcondition is trivial
- Some code does not contribute to the postcondition;
- Not all modified variables are mentioned in the postcondition(?)

A trivial postcondition

```
function Max (X, Y : Integer) return Integer
  with Post => ((if X < Y then Max'Result = Y)
                or (if X >= Y then Max'Result = X));

function Max (X, Y : Integer) return Integer is
begin
   if X < Y then
      return Y;
   else
      return X;
   end if;
end Max;
```

- The postcondition is trivial (always true)
- The programmer wanted to join the conditions with "and"

An incomplete contract

```
procedure Set_Zero (X, Y : out Integer)
   with Post => (X = 0);

procedure Set_Zero (X, Y : out Integer) is
begin
   X := 0;
   Y := 0;
end Set_Zero;
```

- The postcondition does not mention all effects;
- The assignment to `Y` is not used to establish the postcondition.

Detecting redundant and inconsistent preconditions

```
procedure P (X, Y : in out Integer)
   with Pre  => (X <= 0 and X > 0),
   with Post => (...);

procedure Q (X, Y : in out Integer)
   with Pre  => (X > 0 and X > 0),
   with Post => (...);
```

- In both examples, the programmer made a mistake and wrote `X` instead `Y` in the precondition;
- The precondition of `P` is *inconsistent*, it can never be true; without any special mechanism, this subprogram will be proved correct, regardless of the postcondition;
- The precondition of `Q` contains a *redundant* part;
- We propose to detect such situations in GNATprove.