Bill J. Ellis
August 18, 2003

Visiting Praxis: Harder Examples

1 Introduction

Although large scale software does not present any fundamental new problems it does produce increasing amounts of complexity. It remains to be seen if this complexity is artificial, simply presenting the same fundamental operations with a lot of syntax, or genuine, raising the difficulty of proof problem by dealing with more varied and more cryptic properties.

2 Some points

2.1 Functions and procedures

A reminder that functions and procedures are handled slightly different in SPARK.

Functions do not introduce a cut. If the function has a precondition it is evaluated using a check (like the run time checks). The functions post condition is primarily aimed for analysis within the function. The details of the postcondition are introduced by the user creating a separate rule that maps to an automatically generated proof function. Note that it may be possible to synthesise a proof function for a supplied function post condition by changing the function to a procedure, applying the EXAMINER and appropriately reconfiguring the result.

Procedures do introduce a cut. The precondition is proved and the postcondition is cut into the proof[1]. These allow more direct propagation of SPARK annotated properties throughout the code.

2.2 Rules are used

To introduce higher level properties proof rules are used. For example, it is difficult to express that particular instances of records have different constraints on their values using regular SPARK. As these properties occur beyond the code, it is not possible to determine these from the source. If correctness relies on external information, this information needs to be available to complete the proof.

2.3 SPARK is used

Unsurprising, a richer kind of SPARK is used in large examples:

- **Record creation -** Records are created and assigned using the tick notation.
- **Tick succ and pred -** The tick successor and tick predecessor is used where addition or subtraction might be used instead.

2.4 Type and subtype casting

New types are treated differently than subtypes. Operations do not become immediately available for new types, and casting is different as illustrated in 3.2. The use of tightly constrained types tends to result in a high amount of casting. However, as this adds additional information in the code, the casting should be beneficial for proof.

2.5 Is provable or is not

Either the VCs are provable or they are not. Where the VCs are not provable it is due to a lack of information. This tends to be a result of:

- **Composite -** The information exists in external rule files, separated to support different configurations and interface with hardware.

*Notes in this series are for ϵ-baked ideas, for $1 \geq \epsilon \geq 0$. Only exceptionally should they be cited or distributed outwith the NuSPADE project (EPSRC GR/R24081).

[1]This is done under the assumption that the called subprogram has been proved to meet its specification. Note that this specification need not be full partial correctness. It may be sufficient to describe some light properties about the associations between variables or even just their types.

- **Functions -** The information is not propagated through functions. External rule files are created and used instead.

These are frequent enough to warrant consideration. Note that the SIMPLIFIER is fairly weak at exploiting external rules, making these a significant bottleneck in practise.

- **Missing program properties -** These will not occur in provable code. However, in regular unannotated code this represents a key source of information.

Where the VCs are provable, but not discharged by the SIMPLIFIER, limited reasoning is to blame. The SIMPLIFIER targets common patterns, successfully discharging 90% of VCs in most applications. However, this 90% naturally captures the most common patterns as each development of the SIMPLIFIER has targeted these. Thus the remaining 10% do not have any obvious characteristics. If they did, the SIMPLIFIER would be extended to target them. It seems that the SIMPLIFIER focuses on fairly specific patterns. By generalising to a wider pattern, such as reasoning about integer inequalities, it may be possible to cover the remaining 10%.

2.6 Exception freedom properties

Using code comments it is possible to identify a large class of assertions added solely for the purpose of proving exception freedom. Going through all of these, classes of properties emerge and their frequency. Note that this is not an especially calculated analysis but should be fairly accurate.

2.6.1 Cut for decomposition

- **Just cut (common) -** It is common to force the cutting of the VC generation by inserting strictly unnecessary assertions. These may be the trivially true property `true`, or a collection of properties as would be seen for a regular assertion.

2.6.2 Describing type

- **In type (very common) -** Most variables not automatically described by the EXAMINER are asserted to be within their types. This takes the form `Variable in Type`.
- **Inside range (very common) -** It is also common to see the type information presented in a less concise form, as `Variable >= Type'First and Variable <= Type'Last`, where `Type'First` and `Type'Last` are sometimes constants rather than references to the type.
- **Split intervals (very rare) -** It is possible to express a range as being within a type or equal to another value `Variable in Type or Variable=Type'Last+1`. This odd notation is potentially a little more constrained than asserting a single range. However, the need for this kind of assertion is not common.

2.6.3 Unchanged since last

- **Induce from local code (common) -** Where a variable is unchanged since the last assignment or start of the subprogram, it is asserted to have its assigned value or its initial value respectively.

  This is pertinent for asserting that values are unchanged inside loops or for asserting the current status of variables for the strictly unnecessary additional assertions.
- **Induce from subprogram call (rare) -** Following a subprogram call assignment to a variable it is difficult to establish its value directly. The tendency is to resort to asserting that the variable is within the declared return type. However, it is possible to induce stronger properties from the nature of the subprogram call. It is rare for the subprogram to reveal tighter constraints (as if the result was logically more constrained this would suggest using a new type). However, this is exploited where relevant (When it is inconvenient to introduce another type).
- **Induce from precondition (rare) -** Where a precondition is available it can be be used to establish the initial values of variables to assert stronger properties.

2.6.4 Invariants

- **Inside for loop range (all for loops with ranges) -** Technically for loop ranges are not a type. However, within a for loop any declared range is essentially the type of the loop counter. It is common to assert that the loop counter is inside this pseudo type.
- **Dynamic loop ranges (rare) -** These are the the Phil category of variables, whose asserted upper or lower bound depends on the current iteration.
- **Invariant or exit (common in loops) -** Invariant assertions are typically cut in SPARK such that they must also be true when the loop terminates. This is occasionally addressed by asserting the regular invariant or-ed with the exit case.
- **Cut short property (rare) -** It is difficult to express exponentially changing properties within a loop. However, it may not be necessary to express the full details of the property. The effect of a single iteration may be sufficient. For example if a loop repeatedly divides by half, asserting less than half of the largest value may be sufficient for proofs at any iteration.

2.6.5 Other

- **Conditionally true (very rare) -** A property is true, typically within a type, on the condition of an expression. These occur where the type constraint is conditional on performing a check that the type constraint is valid.

3 Examples

3.1 Double Function Call

Short - Due to the treatment of functions rules need to be declared to introduce the required information. Here a double function call is used to highlight the problem. The result of this call may be used to index an array, introducing harder array indices for exception freedom proof.

Proof - Make available the type information of the function outside the function using proof rules. (Could go beyond this and also describe the operation of the function. This may be necessary for partial correctness).

3.1.1 Specification

```
package Example1
is
    subtype TheIndex_Type is Integer range 0 .. 1000;

    function Indexer(Value: in Integer) return TheIndex_Type;

    procedure EX1(A: in TheIndex_Type ;
                  B: out TheIndex_Type);
    --#derives B from A;

end Example1;
```

3.1.2 Body

```
package body Example1 is

    function Indexer(Value: in Integer) return TheIndex_Type
    is
        Result: TheIndex_Type;
    begin
        --Some weird indexing function.
        --(Detials not significant.)
        if Value=TheIndex_Type'Last then
            Result:=TheIndex_Type'First;
        else
            Result:=Value+1;
        end if;
        --Might also experiment with going the other
        --direction in one shot:
        --return (Value+TheIndex_Type'Last) mod
```

```
        --       (TheIndex_Type'Last + 1)
        return Result;
    end Indexer;

    procedure EX1(A: in TheIndex_Type ;
                  B: out TheIndex_Type)
    is
        Intermediate: TheIndex_Type;
    begin
        Intermediate:=Indexer(A);
        B:=Indexer(Intermediate);
    end EX1;
end Example1;
```

3.2 Casting

Short - Shows the different behaviour of the EXAMINER when presented with casting of subtypes and types.

Proof - Four types of legal casting are possible:

- **Unchanged -** The two types are coincident. This does not present any problems or introduce any advantages.
- **Smaller to larger -** Going from a smaller type to a larger type. It is straight forward to prove that the casting is always successful. The result of the cast will be inside the bounds of the smaller type. This smaller bound can be used for analysis.
- **Larger to smaller -** Going from a larger type to a smaller type. It need not be straight forward to prove that the casting is successful. The benefit of completing the proof is that the tighter bound can be used during analysis.
- **Disjoint -** Moving from one type to another type at an overlapping portion. For example:

  ```
  From: 0---------------*----2000
  To:          1000-----*--------------3000
  ```

  This will present a proof that the value is inside the disjoint portion. The tighter bound of the disjoint portion can be used for analysis.

3.2.1 Specification

```
package Example2
is
    type DeclareType is range 0 .. 1000;
    subtype DeclareSubType is Integer range 0 .. 1000;

    function EX3_1(A: Integer) return DeclareType;
    function EX3_2(A: Integer) return DeclareSubType;

end Example2;
```

3.2.2 Body

```
package body Example2 is

    function EX3_1(A: Integer) return DeclareType
    is
        R: DeclareType;
    begin
        R:=DeclareType(A);
        return R;
    end EX3_1;

    function EX3_2(A: Integer) return DeclareSubType
    is
        R: DeclareSubType;
    begin
        R:=DeclareSubType(A);
        return R;
    end EX3_2;

end Example2;
```

3.3 And then

Short - SPARK programmers like to minimise the number of variables and number of paths through the code. It is this aim that sees constructs like `and then` being used within return statements. This example also shows the somewhat nasty syntax used to introduce infix numerical comparison operators for declared types.

Proof - The syntax for introducing numerical comparison is necessary for SPARK but may be ignored for analysis. If an expression uses a type with an operator, we take this is legal assuming the code has been already been successfully passed through the EXAMINER. The `and then` constructs introduce some more classes of VC conclusions. Moving the left hand side of the implication into the hypotheses will probably be a good initial planning step.

3.3.1 Specification

```
package MyTypes
is
    type WideType is range 0 .. 1000;
    type NarrowType is range 400 .. 500;
end MyTypes;

with Mytypes;
--#inherit Mytypes;

package Example3
is
    function ">="(Left, Right: in MyTypes.WideType) return Boolean
        renames MyTypes.">=";
    function "<="(Left, Right: in MyTypes.WideType) return Boolean
        renames MyTypes."<=";
    function ">="(Left, Right: in MyTypes.NarrowType) return Boolean
        renames MyTypes.">=";
    function "<="(Left, Right: in MyTypes.NarrowType) return Boolean
        renames MyTypes."<=";

    function IsInsideType(A: MyTypes.WideType) return Boolean;
    function InTypeLessThan(A: MyTypes.WideType;
                            Upper: MyTypes.NarrowType) return Boolean;

end Example3;
```

3.3.2 Body

```
with MyTypes;

package body Example3 is

    function IsInsideType(A: MyTypes.WideType) return Boolean
    is
    begin
         return (A >= MyTypes.WideType(MyTypes.NarrowType'First)) and
                (A <= MyTypes.WideType(MyTypes.NarrowType'Last));
    end IsInsideType;

    function InTypeLessThan(A: MyTypes.WideType;
                            Upper: MyTypes.NarrowType) return Boolean
    is
    begin
         --Use 'and then' to ensure that A is inside
         --the narrow type before doing its cast for
         --the comparision to upper.
         return IsInsideType(A) and then
                MyTypes.NarrowType(A) <= Upper;
    end InTypeLessThan;

end Example3;
```

3.4 Conditional property

Short - Properties are generally flat, they specify that something is unconditionally true. This presents a cases where a conditional property is suitable.

Proof - It is difficult to envisage a general and useful discovery of conditional properties. It may be possible to focus on boolean types to constrain the problem.

3.4.1 Specification

```
package Example4
is
    subtype WideType is Integer range 0 .. 1000;
    subtype NarrowType is Integer range 0 .. 500;

    procedure CondProp(A: in WideType;
                       B: out NarrowType);
    --#derives B from A;

end Example4;
```

3.4.2 Body

```
package body Example4 is

    procedure CondProp(A: in WideType;
                       B: out NarrowType)
    is
        R: NarrowType;
        RValid: Boolean;
    begin
        R:=A+10;
        RValid:=(R<=NarrowType'Last);

        --#assert RValid -> (R in NarrowType);
        if RValid then
            B:=R;
        else
            B:=0;
        end if;
    end CondProp;

end Example4;
```

3.5 Big return

Short - It is fairly common to see large expressions. These might present a practical problem in the automation of proofs.

Proof - This example trivially can not raise exceptions. However, replacing the boolean variables with expressions will result in the creation of deeply nested exception freedom VC conclusions. These should not present any additional logical problems, however the scale may present practical problems.

3.5.1 Specification

```
package Example5
is
    function BigReturn(A,B,C,D,E,F,G,H,I,J,K: Boolean) return Boolean;
end Example5;
```

3.5.2 Body

```
package body Example5 is
    function BigReturn(A,B,C,D,E,F,G,H,I,J,K: Boolean) return Boolean
    is
    begin
        return (A and then
                ((B or else
                 (C and then
                  D) or else
                 E)) and then
                (F or else
                 (G and then
                  H) or else
                 I) and then
                J and then
```

```
                  K);
    end BigReturn;
end Example5;
```