Bill J. Ellis
August 18, 2003

Visiting Praxis: Less critical SPARK

1 Introduction

The week is closed by diving into a huge less critical SPARK project, developed by many people over many years, to get a comparison with the critical software and explore the scope of proofs with this level of code.

2 Examples

2.1 Reverse

Short - Reverses an array. This is interesting for its dual counters with a definite relationship between the two.

Proof - For exception freedom, type should be sufficient. For partial correctness some more sophisticate properties will be required.

2.1.1 Specification

```
package Example1
is
    subtype Index_Type is Integer range 0 .. 1000;
    type Contents_Type is range -1000 .. 1000;
    type Array_Type is array (Index_Type) of Contents_Type;

    procedure DoReverse(AForword: in Array_Type ;
                        AReverse: out Array_Type);
    --#derives AReverse from AForword;

end Example1;
```

2.1.2 Body

```
package body Example1 is
    procedure DoReverse(AForword: in Array_Type ;
                        AReverse: out Array_Type)
    is
        Up: Index_Type;
        Down: Index_Type;
    begin
        Up:=Index_Type'First;
        Down:=Index_Type'Last;

        --This is seen a a while loop, with some
        --odd treatment at zero.
        --while Down > Index_Type'First loop
            --AReverse(Down):=AForword(Up);
            --Up:=Up+1;
            --Down:=Down-1;
        --end loop;

        loop
            --Assign zero's to hide flow error?
            AReverse(Down):=AForword(Up);
            if Down = Index_Type'First then
                exit;
            end if;
            Up:=Up+1;
            Down:=Down-1;
        end loop;
    end DoReverse;
end Example1;
```

*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).

2.2 Implicit specification or bug?

Short - Searches an array for the index of a value. However, to prove this is free from exceptions requires proving that the element is found. This might be provided by a precondition.

Proof - Precondition discovery is needed to complete the proof. This seems quite difficult.

2.2.1 Specification

```
package Example2
is
    subtype Index_Type is Integer range 0 .. 1000;
    type Contents_Type is range -1000 .. 1000;
    type Array_Type is array (Index_Type) of Contents_Type;

    function DoSearch(LookIn: in Array_Type ;
                      Find: in Contents_Type) return Index_Type;
end Example2;
```

2.2.2 Body

```
package body Example2 is
    function DoSearch(LookIn: in Array_Type ;
                      Find: in Contents_Type) return Index_Type
    is
        I: Index_Type;
    begin
        I:=Index_Type'First;
        loop
            exit when LookIn(I)=Find;
            I:=I+1;
        end loop;
        return I;
    end DoSearch;
end Example2;
```

2.3 Guard and ignore

Short - A common problem in proving exception freedom is incrementing off the end of a type. In theory, the increment should have a precondition stating that the value is less than the maximum. However, this can introduce various problems in propagating the required information for proof. For example, a function `Inc(I: in out Integer)` may be called very frequently deep in the subprogram hierarchy. To prove exception freedom every caller must provide a precondition that `I` is not its maximum value. This presents a significant task if `Inc` is called, directly or indirectly, by most subprograms. An alternative is to insert guards and just ignore the not incremented case.

Proof - The guards can be used to extract tighter bound assertions. These will be introduced naturally as VC hypotheses at the critical point of the increment, but outside this point the information will not be available.

2.3.1 Specification

```
package Example3 is
    subtype Wide_Type is Integer range 0 .. 1000;
    subtype Narrow_Type is Integer range 400 .. 500;
    function CountUp(A: Wide_Type) return Narrow_Type;
end Example3;
```

2.3.2 Body

```
package body Example3 is
    function CountUp(A: Wide_Type) return Narrow_Type
    is
        R: Narrow_Type;
    begin
        R:= Narrow_Type'First;
        for I in Wide_Type range A .. (Wide_Type'Last) loop
            if R<Narrow_Type'Last then
                R:=R+1;
            end if;
        end loop;
```

```
        return R;
    end CountUp;
end Example3;
```

2.4 Just a little too hard

Short - Although not very common there still exists code that beats the recurrence relation analysis.

Proof - It has been thought for a while that something mixing monotonicity, direction and maybe recurrence relations could get a few more relevant properties. This may be sufficient for this case. An important point is that the recurrence relation analysis should be built to fail gracefully (This should be fairly straight forward). Where the recurrence relation analysis fails, it should always be possible to provide type information. This may be sufficient to prove this example.

2.4.1 Specification

```
package Example4
is
    subtype Index is Integer range 0 .. 1000;
    type C_Type is array (Index) of Boolean;

    procedure EX4(A: in C_Type;
                  B: in Index;
                  C: out Index);
    --#derives C from A,B;

end Example4;
```

2.4.2 Body

```
package body Example4 is

    procedure EX4(A: in C_Type;
                  B: in Index;
                  C: out Index)
    is
        Middle: Index;
        L: Index;
        R: Index;
        I: Index;
    begin
        I:=0;
        Middle:=Index'First;

        L:=Index'First;
        R:=Index'Last;
        while I<B loop
            Middle:=(L+R)/2;
            if A(Middle) then
                L:=Middle;
            else
                R:=Middle;
            end if;
            I:=I+1;
        end loop;
        C:=Middle;
    end EX4;
end Example4;
```