Bill J. Ellis
April 14, 2004

Praxis Retreat: Random Notes

1 SPARK to C

The facility now exists to translate SPARK to C. Would this support additional experiments at NASA? The current plan was to modify their existing tools from generating C to SPARK. Now the SPARK could generate C also.

2 Changing variables within loops

For loop exit tests are evaluated as the loop is arrived at. Changes to any variables of the exit test within the loop is legal and will not affect the initial entry condition. This affects what can be accurately said about the loop exit test where the code modifies the variables of the exit test within the loop. Note that this behaviour is considered bad practise, but still perfectly legal SPARK.

3 Extra cut between if statements

It would be possible to propose cuts between if statements. This can be undertaken even if the if statements are within nested code. This would help reduce the number of VCs and assist proof. In particular, this would represent an algorithmic approach for adding strictly unnecessary (but quite useful) cuts into the code. For example:

```
if A then
    if B then ...
    end if;

    --#assert A and ...;

    if C then ...
    end if;
end if;

--#assert ...;

if D then ...
end if;
```

Note that there may also be scope for inserting assertions between case statements (as case statements are ultimately if statements).

Note that it would also be possible to insert cuts at the request of the user. That is, enrich invariants where the user has requested them.

4 Requested information

It would be possible to generate long annotations for tightly constrained high level specification requests. This is more a possible feature for a kind of interactive program analysis, rather than a real suggestion for proof automation. The high level specification request might ask for a constraint disjoining all of the possible values a variable can have at a given point.

[*] 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).

5 Type of subexpressions

In evaluating:

```
R:=(A+B)/2;
```

It must be the case that:

```
(A+B)/2
```

Is within the type of R. However, it must also be the case that:

```
(A+B)
```

Is within the base integer type for this project:

```
--#assert (A+B) >= Integer32'Base'First and
--#assert (A+B) <= Integer32'Base'Last;
```

This is quite significant for proving lack of overflow for nested expressions (not of the form (variable/constant op variable/constant).

Note that this is not required. The constraints on A and B should be sufficient to show that (A + B) is legal. If A and B are too wide to show this then (A + B) is an error. Thus these assertions are added to assist the Simplifier. It is likely that the modern Simplifier can complete proofs without these extra assertions. In particular, the assertions must still be shown correct, this will impose the same kind of proof effort required to prove the original (A + B) goal with the assertions absent. I.e. it moves the problem to an earlier point, but does not make it any easier. However, there may be some efficiency gains in introducing these extra assertions if (A + B) occurs often.

6 Press on with debugging methods

The subexpression never in type because its parts are too big is a good debugging example (cpu.fulldatabasepromtest). For example, want to show that (A+B) is within its bounds, given bounds on A and B that are too wide. It should be possible to locate these bugs using a constraint solver.

7 Working with boolean values

Where working with Boolean values it may be possible to discover additional properties such as are usually found from tests. For example:

```
if A then
    B
else
    C
end if;
```

Inside B it is taken that A holds (until a variable in A might have changed) and inside C it is taken that not(A) holds (until a variable in A might have changed). Outside B and C nothing is said about A as a result of the test.

```
A:Boolean;
...
A:=E;
```

After this assignment, until a point is reached where A may have changed, it is true that:

```
A <-> E
```

holds. I.e. If A is true then E must have been true and if A is false E must have been false. This little trick may be used to find additional properties, and in one example at least (coc4p.calcadvice) could lead to valuable properties.

8 Getting information outside tests

After a test the true and false branches merge, cancelling out any introduced properties. However, it may still be possible to determine which branch was entered and thus allow properties to be carried outside the test branch. For example:

```
if A then
    B:=True;
else
    B:=False;
end if;
```

Following this it is valid to assert:

```
--#assert B<->A;
```

so long as A does not contain B. This kind of reasoning might be applicable to an example (coc4p.calccone.findwidestcone) however, in this case additional reasoning is still required to derive useful properties. This may also be used to find stronger postconditions, probably more frequently for functions. Note that this would require variables being globally visible (and non global variables being suitably eliminated). From:

```
if A then
    B:=True;
else
    B:=False;
end if;
return B;
```

To constraint:

```
--# (B=true)->A and (B=false) -> not(A);
```

```
--# ((B=true) and (A)) or ((B=false) and not(A));
```

To postcondition:

```
--# return RESULT =>
--# ((RESULT=true) and (A)) or ((RESULT=false) and not(A));
```

Note that the values assigned to the variable in the separate branches must be different. Otherwise A and not A will be introduced for the same property, and one of these will be false.

9 Refine properties

The properties discovered through program analysis can be refined, possibly through using constraint solvers or external reasoning tools. Certainly this refined information is valuable, but should it be presented in isolation from the original less refined information? Consider:

```
A: Integer;
for I in T range 1 .. A-1 loop
...
```

It can be found that, assuming A is kept unchanged in the loop:

```
I>=1 and I<=A-1
```

But, given the type of A, it can also be found that:

```
I>=1 and I<=Integer'Last-1
```

Or even:

```
I>=1 and I<Integer'Last
```

So long as constraints on A are given, this can be deduced. In other words, where finding constraints on a variable involving variables, consider replacing these variables with their type.

Note that their are issues with this trick where applied inside loops. Proving an invariant to invariant case with fixed bounds must appeal to the negation of the loop guard. Such a situation is unlikely.

Note that this is a bit like adding the type of subexpressions - it should not be needed to be done so long as individual variables are asserted to be within their types.

10 Postconditions

Have rather dismissed the benefits of generated postconditions as the best that can often be said at the end of a subprogram is that variables are within their types. This is not entirely correct. Functions can often directly reveal full postconditions. Procedures (Such as the Phil example) may employ overly large types, such that the internal constraints of a subprogram are valuable outside.

Note that this raises additional issues of when these postconditions should generated and how they should be added. A proof failure might indicate the need for stronger information on a variable, but it may be difficult to tie this to a requirement for a stronger postcondition on a called subprogram.

11 Feedback into the SPARK

If doing large grain experiments on SPARK, it would be handy to have some mechanism to automatically enrich the SPARK code with annotations. This need not be perfect. Consider adding an invariant to every loop. This would immediately eliminate the hassle of deciding where Examiner inserted invariants occurred. Consider annotating the annotations? It is difficult to relate the analysed flowgraph to the syntactic SPARK. Doing:

```
--# assert dummyInvariantTag(1) ...
```

Will insert junk into the VCs. Doing:

```
-- dummyInvariantTag(1)
--# assert ...
```

Will require processing comments. Doing:

```
--# tag dummyInvariantTag(1);
--# assert ...
```

Will upset the Examiner. It would be neat to explicitly distinguish between the user added and automatically discovered annotations:

```
--+ tag dummyInvariantTag(1);
--# assert (user-added-assertions) and
--# --+ tag insert generated below
--# (generated-assertions);
--# --+ tag end of generated annotations
```

However this is rather complicated. An enriched annotation language should be avoided if possible.

12 Constrained by proof planning

Only need to add annotations where proofs fail. However, what annotations should be added? Is the process of working this out worthwhile? It would be quite inexpensive to throw in all information if things are not working out.

13 Function rules

Called functions typically require proof rules to complete the reasoning. These proof rules introduce an expression for the function. Recall that the function postcondition is not propagated into function calls.

It might be possible for the program analysis to generate function rules, but the program analysis is unsound, and hence so are its rules and so this is not an especially good route to take.

Another route may be to manually propagate function postconditions into assertions. To prove the assertions the function definition could be added as a precondition property. These preconditions could be propagated to the top calling main subprogram. These would then be preconditions to the system, to be ticked off accordingly. Although sound this approach would lead to lots of additional proofs and annotations. It would be cleaner to introduce rules directly, immediately reaching the point where items need to be ticked off.

Ideally it should be possible to prove that a functions postcondition corresponds to its proof rule. Then the program analysis could generate rules without worrying about soundness. Currently this proof is conducted by a person. Thus the program analysis could generate candidate proof rules for functions, issuing suitable warning messages that these have not been proved.

14 Program analysis with pre and post

Program analysis may also exploit pre and post conditions. These are additional properties known to be true for a called subprogram. This may be a means to do n-subprogram analysis. It is likely that only details of the immediately called subprograms need be considered. Note that this approach would be most effective where tackling proof problems from the bottom up. This way the called subprograms of a subprogram should have already been considered.

15 Why only properties at variables

Consider:

```
if A then
    --#assert A and ...;
end if;
```

The property A is not attached to a single variable, however it can still be asserted. These properties are no less important than the properties (abstract values) attached to variables.

16 Add gained postcondition

Postconditions for a subprogram will naturally become available in the VCs for subprograms that call this subprogram. However, in passing through an invariant (or cutting assertion) these properties are not carried through. It becomes reasonable to enrich assertions with all information, including pulling up the full details of postconditions.

17 Talk to theorem prover

We want to be able to negate logical formulae and simplify to something useful. Integrate with a first order (or higher order) theorem prover. Look for theorem provers for this purpose.

18 Having the labels helps

Using:

```
X'First
```

is more meaningful than using its constant value 0. However, maintaining these labels internally can be problematic. Should:

```
X'First+1
```

be presented as is or simplified to:

```
1
```

In particular where employing external reasoners (such as a constraint solver) the labels can not be sensibly maintained. Perhaps the labels should be kept for as long as sensibly possible, being discarded when better (but no longer named) constraints are found.

19 Examiner and type

The latest version of the Examiner will add type for all variables once the variables have been assigned too (variables are not within their type prior to an initialising assignment). Some points on this:

- **Where is the information added -** The information is assumed. If SPARK code passes the Examiner with zero errors then every referenced variable must have a legal value. Thus the information that referenced variables are within their type can be safely added, assuming the Examiner has zero flow errors. Note that this will only ever insert hypotheses, and never conclusions.

```
(--#post Y=Y~ and X=X~ and X in type and Y in type and T in type;)
T:=X;
(--#post Y=Y~ and T=X~ and T in type and Y in type and T in type;)
X:=Y;
(--#post X=Y~ and T=X~ and T in type and X in type and T in type;)
Y:=T;
--#post X=Y~ and Y=X~ and T in type and X in type and Y in type;
```

These type hypotheses occur for all variables at all code points under the been assigned to condition. The nature of the type hypotheses will still be controlled by the hoisting process.

```
package body Switch
is
   procedure Int_Switch(X, Y: in out Integer)
   is
      T: Integer;
   begin
--#assert X in Integer and Y in Integer;
      T:=X;
--#assert X in Integer and Y in Integer and T in Integer;
      X:=Y;
--#assert X in Integer and Y in Integer and T in Integer;
      Y:=T;
--#assert X in Integer and Y in Integer and T in Integer;
   end Int_Switch;
end Switch;

package Switch
is
   procedure Int_Switch(X, Y: in out Integer);
   --# derives X from Y &
   --#         Y from X;
   --# pre false;
   --# post ((X=Y~) and (Y=X~));
end Switch;
```

- **What if the initial assignment is within a loop but after the invariant -** The approach only adds properties where a variable is being referenced and thus has been assigned to.
- **Detection of assigned to will be limited for arrays -** This has always been the case. However, now if the flow analysis is incomplete their could be logical holes in generated VCs. It might be interesting to consider proving the correctness of loop assignments. However, this would be difficult if the Examiner generates the assumed conclusions. Could use a proof function: touch(L,U,Array) asserting something like: touch(Array'First,I,Array) within the loop and: touch(Array'First,Array'Last,Array) just after. Note that SPARK encourages aggregate assignments, however it may still be interesting (and quite straight forward?) to prove these array initialisations.

20 Function returns

SPARK forces the function return to be the last statement in a function. This typically means that the function returns a variable which is processed earlier in the function. Can also assert that the statement returned is within the declared type of the function. Failure to prove this indicates that a stronger precondition is needed.

21 Simplifier feature

The Examiner generates rules about constants in rls files in a form that can be directly exploited by the Simplifier. Rather than use explicit universally quantified properties, the rules are generated with Prolog variables in place. These are also decomposed into individual conjuncts. For example, what might be expressed as:

```
for_all(x, (x>=1 and x<=150) ->
    (element(array, [x]) >= lower and
     element(array, [x]) <= upper))
```

Is expressed as:

```
rule(1): element(array, [I]) <= upper may_be_deduced_from [I>=1, I<=150].
rule(2): element(array, [I]) >= lower may_be_deduced_from [I>=1, I<=150].
```

22 More going backwards

Consider the following:

```
loop
    --#assert I in ...;
    I:=I+1;
    ...
end loop;
```

The increment of I is not conditional having seen the invariant. Thus, by working backwards from the increment it can be found that I need not within its type. Note that this is similar to precondition discovery by considering the code in reverse.

23 Procedures cut a hole

Information on one side of a procedure may not be available on the other side if the procedure imports a variable with out mode. The modified variable is given a new name and the only information on this variable comes from the postcondition of the called procedure.

The called subprogram may have a stronger precondition and postcondition taking into account the calling context. However other calls may not also have this context available. In these cases it might be valid to create copies of subprograms, each with more specialised preconditions and postconditions. This seems more a problem of less suitable subprogram choices.

24 No class invariants

Is the absence of class invariants the reason subprograms have matching preconditions and postconditions?

25 Why assertion discovery

Assertions should be created in tandem with the code. So why bother with assertion discovery when the assertions should have already been found? Often the assertions are not immediate and can become large. As the code changes as do the assertions. The assertions are distant to the coding process for most programmers. Think of assertion discovery as a way of automatically telling the programmer things about their code. A more clever kind of lint. That the properties are geared for proof is another part of the story.

26 Consider low integrity

High integrity code is developed with a high integrity mindset. Annotations are added, loops are avoided, proofs are tackled, code is refined accordingly. Regular code will not be subject to such luxuries. Is it automated proof that is significant, or automated proof of high integrity systems. Why not consider both? There are low integrity SPARK systems that could be considered.

27 Precondition discovery to make provable

It is common amongst the VCs not expected to be proved by the Simplifier that the VC is not provable, and that the VC is not provable due to a too weak specification.

In the cases where this occurs the reasons for the weak specification can vary. For example, because the constraints come from an assumption a database is in order. It is fairly straight forward to identify a weak specification as an unprovable VC. However tracing this to the correct stronger precondition is difficult.

Often the problem is a simple off by one problem. One option is to assume that when a program terminates all of its variables are in their type and trace the program backwards to establish what constraints must exist on these variables at the start. This is not always possible, and still quite difficult. Strengthening preconditions is a possible future extension, but not sufficiently understood or important enough to consider before the regular system is going.