Bill J. Ellis
August 18, 2003

Visiting Praxis: Random SPARK Observations

1 Irregular SPARK

1.1 Fixed point

Fixed point numbers are a means for encoding a discrete range of floating point numbers. Proof of these is conducted as for floating point numbers, but it is no longer inappropriate to assume that the arithmetic is perfect. Note that fixed point numbers are declared using tick delta.

1.2 Named parameters

The named parameters mechanism is used frequently, especially where indexing records. Need a common way to express both positional and named indexing in the program representation for analysis.

1.3 Separate (package)

The details of larger procedures are sometimes held in separate files using the separate syntax.

2 EXAMINER 7.3

The latest version of the EXAMINER now seems to discharge a greater number of trivial VCs. It now looks for a direct match with the conclusions in the hypotheses. This removes all overflow checks where performing simple assignments between variables of the same types.

3 Z specification

A Z specification for a high integrity software system was made available. Although not considered in any great depth the following observations are made:

- **Is large -** The Z specification is a large structured document presenting both natural language and formal Z descriptions of the systems behaviour.
- **Is part of a series -** Following the specification is a design document and following the design document the code is produced.

  Although their is no formal proof of the refinements, as for example following the B method, there exists trace keywords which allow a concept in the specification to be linked to the corresponding point in the design document and then from the design document to the code.
- **Is tied to the code -** Although the design document was not available, making it difficult to trace from the code to the specification following the trace keywords, it is clear that the code corresponds closely to the specification.

  In particular, Z types, various low level logic and mathematical functions and top level call sequences generally have a very similar representation in SPARK. This is probably primarily because these features of the Z can be coded in SPARK in a natural efficient manner. The remaining Z must be subjected to larger transformations to express the same behaviour in SPARK.

*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).

4 Assertions in the large

One of the problems of proving properties about large programs is the production of several large VCs. These are difficult for human consumption and can even become difficult for machine consumption.

It may be argued that such problems arise as a result of poorly partitioned software. However, it is sometimes the case that following the good guidelines of high cohesion and low coupling the result should be a single subprogram[1]. The partition into separate subprograms to assist proof would mean allowing proof to dictate design. (In many cases subprograms could be introduced without any conceptual problems. However, even the best software still contains constructs that produce many large VCs.).

A solution, other than partitioning into subprograms, is to decorate the code with assertions. Each assertion brings together all of the paths, minimising the number of VCs. Further, the assertions can throw away information no longer needed, keeping the VCs more relevant.

5 Hashing formulae

It has been our intention for some time to carry around formula in the whole, following the HOL approach to theorem proving. Facing similar problems with explicitly named formula, the Praxis developers had been considering carrying around hashed formula. Essentially munging a formula into a hopefully unique number and carrying this around instead.

This hashing approach is necessarily not perfect (clashes are possible) but may be more efficient. However, until efficiency becomes a major issue, it seems reasonable to continue to purse the much more straight forward approach of using the full formulae.

6 Growing the SIMPLIFIER

The SIMPLIFIER is developed in reaction to the observation of common patterns. To avoid repeating the discoveries made within the SIMPLIFIER the CVS (Concurrent Versions System) repository is given a quick search.

6.1 Some forms targeted

- **Exploit -** $A > I \rightarrow A <> I$ and $A < I \rightarrow A <> I$
- **Handle -** $A + B \geq Z$ and $A + B \leq Z$
- **Common for RTC -** $A \leq B$ and $A \geq B$
- **Introduce constants -** For all variables initially.
- **Pred and succ -** Replace enumerated functions with there referenced value (for example replace $succ(Saturday)$ with $Sunday$).

7 And E is bigger than S

The data structures in the S system were refined for proof by artificially decomposing structures into smaller structures more amiable to proof. This is something of a special case, with systems such as E carrying around several times larger structures as artificial decomposition for proof has not been conducted.

8 Its true! I think. Now what?

Diverse proof checking increases confidence that a proof is correct, especially if running the proofs through on a dedicated heavily tested proof checker. However, diverse proof is not always available, so the proofs are instead manually inspected.

Once confidence in the proofs has been shown, the code needs to be compiled. Having not been proved correct, it is necessary to check that the byte code outputted from the compiler is correct. Again, this requires manual inspection.

[1] Subprograms may be nested within a subprogram, correctly recording associations. However, having subprograms that are only called once seems odd.

9 Where are the recurrence relations?

Code that would reveal useful properties through our brand of recurrence relation analysis is quite rare. In CS-note 59 the kinds of assertions seen are classified. The type based or unchanged assertions are most common. Note that the recurrence relation analysis can be applied to most loops, only it need not directly reveal anything new.

Thus there is definitely a need for program analysis. Recurrence relations provide a way for discussing loops which may allow for more principled property discovery, adding type where that is applicable. However, it is increasingly clear that breadth is more important that depth. Many little techniques would bring greater results than one large technique.

Note that recurrence relations or not, the implementation of program analysis remains essentially the same. The key is to build tools allowing the development of program analysis heuristics.

10 Analysing SPARK

The current program analysis is about to undergo the next phase of splitting into symbol table generation and subprogram extraction. The following rough plan had been considered:

```
          SPARK
            |
            V
  --------------------
  |                  |
  V                  V
Symbol             Subprogram
Table  -------->   Extraction
                     |
                     V
                   Spark in
                   Prolog form
```

Some observations are made:

- **This seems kind of reasonable -** The EXAMINER employs symbol table generation, well formedness checks and then performs the analysis requested. There is a need for a symbol table generation phase (To establish what a name refers to).
- **Ada compilers ignore comments -** To exploit an Ada compiler would require doing something to deal with SPARK annotations as Ada compilers will dismiss these as comments.
- **Aspects of SPARK are hard -** Separate and child packages have complicated semantics. There is (currently) no good way to translate a full SPARK program into a normalised SPARK program, respecting comments appropriately (And probably not straight forward ignoring comments). It may be worthwhile implementing this as a genuine program transformation, rewriting a SPARK program with separate and child packages into a semantically equivalent program without these constructs.
- **The EXAMINER is different -** The EXAMINER parses SPARK into a large internal tree performing similar analysis to above. However, without the high level semantics of STRATEGO, this is necessarily implemented using stacks and other suitable data structures. The EXAMINER does more than is needed in a different way thus it is reasonable to tackle the required program transformation within a new system.