CS-Note* 67

Bill J. Ellis
April 14, 2004

Praxis Retreat: A Roadmap (Why am I here? ; Where am I going?)

1 Introduction

In a previous visit to Praxis the nature of high integrity software was explored and compared against our ideas for proof automation. Here we return to Praxis to evaluate the strength of our existing tools and gather data enabling a comprehensive experiment at a later date.

This short note describes the initial observations made and the rough plan of action constructed for my brief time at Praxis.

2 In-house development

The game of increasing proof automation is also played at Praxis. Both the Examiner and the Simplifier are being modified:

- **Examiner -**
  - **Type -** Adds type information once variable have been assigned to. This was previously available for import parameters. Now the same idea is applied for for local variables and soon for export variables.
  - **Constants -** Constants are inserted into the code in SPARK, thus should be presented as rules by the Examiner. Currently the Examiner is a little weak with these. The problem is that condensed constant expressions involving quantification are difficult for the Simplifier to exploit. However, generating all of the flat constant replacement rules will be prohibitively expensive in some cases. Some suitable compromise between these extremes is being sought.
- **Simplifier -**
  - **Package 1: Arithmetic and inequality reasoning -** Improve reasoning with more tricky and non linear arithmetic operators such as abs, mod, div and pow.
  - **Package 2: Structured objects -** Clever simplification of structured objects (records and arrays). Note that simplification means simplification, eliminating logically unnecessary update terms by exploiting hypotheses. Also improve working with universally quantified hypotheses and conclusions.
  - **Package 3: Logic handling -** Improved handling of logic term manipulation, for conjunctions, implications, disjunctions and boolean equalities.
  - **Package 4: Working with rules -** Improve appealing to structured constant bound rules and 'pos and 'val rules. This will involve configuring the Examiner and Simplifier to adopt a consistent practical policy with regard to structured constants.

3 Observations

- **Simplifier changes from deep analysis -** The changes to the Simplifier are made to target recurring classes of problems. Finding the classes requires deep experience of SPARK VCs. This is a game Praxis will win!
- **Biggest gains / minimal changes next -** Unsurprisingly the changes with the biggest proof gains and minimal changes to implement are targeted first. The difficult examples are less likely to be pursued at Praxis. Arguably our tools are more suited to these difficult problems. This is a good place to focus our efforts.
- **True, testing, true, true, true -** The tools of the SPARK approach do not follow the proof planning paradigm. False is not an option. Are there occasions where our more flexible approach to soundness can be exploited?

*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).

- **Limits of property discovery -** It is difficult to discover the specification. This applies both to simple constraining preconditions and more complex loop invariants and post conditions.
- **Rekindle n-subprogram analysis -** Is n-subprogram program analysis feasible? This has been given less emphasis on the basis that single subprogram analysis is a more suitable starting point. However it may be reasonable to return to the n-subprogram ideas at a later point.
- **Show me the constants -** Constants are inserted into the code in SPARK, thus should be presented as rules by the Examiner. It might be sensible if the Examiner generated the most concise possible presentation of the full constant information. This could then be exploited by the Simplifier (or NuSPADE) as required. Until the constant information is generated by the Examiner, NuSPADE can not really progress for these.
- **Constant selection constrained by proof failure -** Need for specific constants might be suggested through a critic. The point of proof failure could indicate the particular constant required rather than initially adding all constant information. The rules themselves might be created through our program analysis, however this would raise dangerous soundness issues.
- **Why just assertions -** Is there scope for creating rules from the code? In some cases it would be possible to discover a proof function for SPARK functions. Further, it would be possible to discover rules introducing constant information. However, doing this will raise soundness issues (how do you prove that a constant rule or proof function rule is correct?).

4 Roadmap

Some of our features are beginning to overlap slightly with the the new features emerging from Praxis. The aim is to focus on the problems that have been marked as unlikely to be proved through upcoming changes to the Examiner and Simplifier. I.e. focus on the problems Praxis are unlikely to automate while exploiting the differences (benefits) of our strong proof planning framework.

4.1 Production

The intention is to produce a collection of examples that reflect the problems our tools will prove. Implementing coverage for these should naturally lead to strong empirical evidence showing the applicability of our technique.

4.2 Presentation

An example will comprise off:

- **VCs -** VCs include a vcg, rls and fdl file. For security reasons a small number of VCs may reasonably be exported to Heriot-Watt for analysis. Classifying VCs is tricky (especially those that Praxis do not expect to prove soon - if they were straight forward they would be classified and tackled with a suitable amendment to the Simplifier). Thus rather than spending time on VC classification, examples will be returned to Heriot-Watt for more in depth analysis.
- **Code -** For security reasons code can not be exported to Heriot-Watt. Thus code examples will be generalised to a few cases and these cases will be referenced.

There are three different kinds of examples:

- **Sole property generation (SPG) -** A subprogram which can not be proved without first adding some annotations. Once the annotations are added the Simplifier can prove the subprogram.
- **Sole proof planning (SPP) -** A subprogram generates provable VCs that can not be provable by the Simplifier. Suitable proof planning methods may be discovered to prove these VCs.
- **Dual property generation and proof planning (DPGPP) -** A subprogram which can not be proved without first adding some annotations. Once the annotations are added the Simplifier leaves provable VCs undischarged that might be tackled within the proof planner.