Bill J. Ellis

June 9, 2005

Let there be PropGen-Light

Contents

[*]Hums are short notes intended for distribution between those involved with EPSRC grant GR/T12289/01. Hums describe ϵ-baked ideas, where $1 \geq \epsilon \geq 0$. (Hum refers to both the Praxis Humming bird and Winnie-the-Pooh's indirect approach to writing: "Poetry and Hums aren't things which you get, they're things which get you.").

1 Introduction

In Hum 1 a project plan is presented, placing proof planning as the critical path and program analysis as a speculative activity. Here a few (speculative) points are made about how some form of program analysis might be achieved.

1.1 A thought: PropGen-Light

Progress on the SPADEase-Parser will involve hacking together code[1] from the alternative NuSPADE code base, NuSPADE itself and various parts of PropGen. The emerging system is a lighter, more straight forward, component of NuSPADE. No longer a research system, it is clear what the system should do and consequently fairly clear how it should be implemented. PropGen similarly evolved as a research system. It contains various features that reflect its research heritage:

- **Semi automated input -** PropGen has some input provided by the user (via the infamous fdlcode file) and some input provided through parsing and transforming SPARK (via the disciplined but cryptic Stratego).
- **Lose connection to mini planner -** PropGen has a lose connection to a mini proof planner, originally developed to be part of NuSPADE (which, surprisingly, works rather well, but, in all fairness to software engineering, it probably should not).
- **Unstructured integration to external tools -** PropGen has a conceptual integration to a recurrence relation solver (which is reasonable). It also has genuine integration to constraint solving (which is rather unstructured and becomes flummoxed with large numbers).
- **Lack of coherent output -** PropGen lacks any form of coherent output. The results are calculated and stored in an internal form in a few data files. These would not be expected to be read by any user of the system.
- **Wild formulae translations -** Sometimes formulae is described as seen in the code, sometimes as desired by the constraint solver, and sometimes in a form of FDL (and sometimes it is difficult to tell which is which).
- **Support for multiple subprogram analysis -** PropGen, in theory, supports the analysis of multiple subprograms. This is an interesting feature to explore, but is not part of the core behaviour of PropGen.
- **Support for distributed planning -** PropGen has features to support distributed planning on multiple machines. This is a legacy from originally anticipated days to complete analysis. This feature is not required.

It is speculated that, somewhere inside PropGen, is a lean system that simply does single subprogram invariant discovery, as was finally seen for NuSPADE.

2 Sweeping away the clutter

2.1 One subprogram at a time

What if... subprograms were analysed in isolation?
Currently, this is exactly what is done. The PropGen system is used to analyse individual subprograms in isolation from other subprograms. However, this is achieved using a more complex framework capable of multiple subprogram analysis. Processing multiple subprograms together is an obvious and interesting line of research, but not part of the core PropGen behaviour.

[1] Read: Carefully exploiting software reuse.

2.2 Abstract away subprogram calls

What if... subprogram calls were abstracted?

When PropGen encounters a call to a subprogram it does not analyse preconditions or postconditions. Such annotations are rare for exception freedom. For procedure calls the constraints on modified variables are lost, being replaced with type constraints. Functions are replaced with their return types. This behaviour is consistent, regardless of which subprogram or function is called. Thus every function can be described as: `returnType(TYPE)` and every procedure as: `resetToType(LISTOFVARIABLES)`. This would not affect the core behaviour of PropGen but would radically simplify its behaviour[2]. Note that this seems a very obvious simplification for a core version of PropGen, but would be a rather narrow, and difficult to undo, constraint for a research system.

2.3 Disband relationship to the code

What if... subprograms were not related to the original code?

A fair amount of data is maintained to relate subprograms and variables to their SPARK counterparts. This information would support the automated insertion of invariants back into the SPARK code. This would be a useful feature, but is not essential. By pruning all subprogram context information a more simple (and weaker, but sufficient) data structure will remain. Note that a consequence of this will be the discovery of invariants in a syntax with a very obvious translation to the correct SPARK syntax[3].

2.4 Theoretical parsing

What if... there was zero SPARK parsing?

PropGen supports some SPARK parsing. If the SPARK is in a prescribed form it can automatically be translated into a graph structure for analysis. Unfortunately, the parsing process for variables is complex[4] and was replaced with a user created file. As the user is already providing some information, it would be reasonable to require them to provide all information. Bypassing the SPARK parsing would allow for immediate access to an ideal data structure for input. The onus on the user for creating this input is unfortunate, and something of a hassle, but quite feasible for small examples. Note that another tool might subsequently take up the task of generating this input file from the original SPARK. A quick look at the SPARK Examiner code suggests this would be quite difficult. However, their may be potential for a limited functionality version of such a system.

2.5 Reduced data structures

What if... there was integers, boolean and arrays, and nothing else?

Complex situations can be created via these three types. In particular, arrays of arrays (of arrays) of integers or booleans can be created. Although records add complications, these are not at an analysis level - a variable within a record is still just a variable. Enumerated values can be mapped directly to integers, highlighted by the `type.pos(X)` conversion. Boolean might also be eliminated, being thought off as an integer taking the values 0 or 1. However, boolean occurs often in code, and is fundamentally different to integers[5].

2.6 Integrated planner

What if... the planner was directly integrated for program analysis?

Currently the planner is an entirely external system, invoked automatically via the command line from Prolog. Communication is via a few carefully configured files. Directly integrating the planner with the PropGen system would eliminate this communication and reduce complexity.

---

[2] If subprograms are not abstracted, then their must be a means to link the subprograms to their corresponding code. The point of call does not contain type information, thus the corresponding code must be inspected to achieve the desired behaviour. Further, in the FDL all procedures calls are translated to functions. PropGen respects this translation (via fdlcode), leading to a collection of slightly cryptic functions. Some functions are built in and others a consequence of subprograms. The subprogram functions are identified as those not built in, requiring the built in functions to be made explicit...

[3] In most cases it is expected that the translation will involve adding `--#assert`. However, in general, addressed variables (`A.B.C`) can no longer be translated into their true SPARK.

[4] A variable's type is declared away from the variable use. Type itself is rather complex (types, subtypes, subtypes of subtypes). Factor in global variables, scope problems, for loop variables, information encoded in SPARK annotations, and the process becomes quite an ordeal.

[5] And, interesting things can be done with Boolean during program analysis. It is thought that keeping boolean around will not result in much additional effort yet should support the formulation of more interesting and natural examples.

2.7 Ditch pre and post condition processing

What if... pre and post condition processing was fully abandoned?
PropGen continues to support the potential for processing pre and post conditions. Various parts of the code contain hooks identifying the relevant changes that would be required. Fully abandoning this would not affect the functionality, and only slightly affect the code. However, doing so would mean that this is no longer a missing piece of functionality. Thus, it eliminates having to implement pre and post condition processing to finish PropGen.

2.8 Overhaul the syntax

What if... the syntax made sense?
Possibly the most questionable aspect of the PropGen implementation is its syntax. It is necessary to have a means to refer to program variables, and various other variables, during the execution of the system. In particular the additional variables tend to have a connection to the program variables, for example program variable `a` might lead to additional revised values for `a` as `a'`, `a''`, `a'''`. Currently this association is achieved by embedding the parent variable inside a Prolog structure `newvar(a, 1)`, `newvar(a, 2)`, `newvar(a, 3)`. This, of course, means that syntactically `a` is both a variable and a component of a variable, leading to various problems in term processing. It is apparent that an overhaul of the existing syntax is required, however the exact form of this will require more detailed consideration.

2.8.1 More consideration...

All of the syntax processing problems stem from variables not being Prolog atoms. This is a low level implementation detail, but does affect the clarity of the system at many levels. having all variables as atoms is a good improvement and key target. It is quite possible to have every variable as an atom, and attach properties to this atom in a separate data structure. The following is envisaged:

```
variable(NAME, PROPERTY).
```

Having such a lookup table, for every variable in the system, would be straight forward to implement, would allow variables to be atomic, and would be generally useful. For example, as every variable will be explicitly accounted for, it will be straight forward to generate a definitely new variable name[6].

2.8.2 Even more consideration...

Unfortunately, this does not take functions into account. An array can be thought of as a function, returning different variables for different indexes. It is not practical to assign a discrete atomic variable for every element in an array. Arrays may be accessed though the usual `element(A, I)` and `update(A, I, B)` functions. This led to the following being recorded in PropGen:

```
getPageTargetVarsDetails(subprogramBlock(1),
                         arrayElementVariable(d,d,element(d,[A]),[A],integer)).
```

This supports the testing for elements of array `d`. However, this will not match if `d` has been updated, thus such testing methods are not generally applicable.

Every variable, including arrays, remain atomic. It is the accessing of the elements of an array which require a function `element(...)`. These functions will need to be considered accordingly. This will impose complexities on the analysis process, however this complexities can not be avoided where accepting that functions can return variables. Fortunately, this case would be well constrained in PropGen-Light as most other functions are eliminated or simplified.

3 Executive summary

A consequence of the research nature of the NuSPADE project is that PropGen has a large and confusing ad-hoc implementation. By sacrificing some areas of functionality and rationally reconstructing the PropGen architecture and core data structures a new, lighter, version of PropGen could be realised. This PropGen-Light may be integrated within SPADEase. However, a key weakness of PropGen-Light is the need for examples to be manually configured for analysis. This would restrict PropGen-Light to academic experimentation rather than real world analysis. Nevertheless, it would be possible to write an additional component that automatically presented examples in the form desired for PropGen-Light. Thus PropGen-Light may be adapted for real world analysis if desired.

[6]Separate systems PropGen and the mini planner generate new names, thus they need to communicate what names have been used.