An Introduction to ProofPower

Roger Bishop Jones

Date: 2006/10/21 16:53:33

Abstract

An introductory illustrated description of ProofPower (not progressed far enough to be useful).

http://www.rbjones.com/rbjpub/pp/doc/t015.pdf

Id: t015.doc,v 1.2 2006/10/21 16:53:33 rbj01 Exp

Contents

1 Introduction

ProofPower is a suite of software supporting the application of formal methods in "high assurance" applications, particularly intended for applications in which some kind of certification is required. Originally *secure* systems were the motivating applications, but later *safety critical* applications became more important. These perceived target applications determined the functionality which ProofPower now delivers, not so much as because of the intrinsic characteristics of these application domains as of more or less accidental features of the regulatory environment or of other developments occurring in these domains.

Thus, ProofPower began primarily as a re-engineering of the Cambridge HOL system (then "HOL88") to meet an expected demands for:

- the use of formal methods in the development of secure systems
- the requirement for certification of tools to be used in the development of secure systems
- the inclusion of development tools in the evaluation of secure products

Over and above mere re-engineering, the major new functionality provided by ProofPower was support for formal specification and proof in the Z language [1]. Support for Z was thought desirable because of the clearly expressed preference of the security certification authorities for formal specifications to use that language. Z was supported by a semantic embedding into HOL, but prior experience, in using HOL for proofs relating to Z specifications and from the prototyping of Z proof support on a prototype of ProofPower, resulted in the design of ProofPower-HOL being substantially influenced by the desire to support Z (and possibly other languages) by embedding. ProofPower is the result of reconciling and merging two quite distinct subcultures of the UK formal methods research activity (centred around Cambridge and Oxford), and its character shows these origins.

Extensions to ProofPower beyond support for Z were stimulated by safety critical applications. RSRE Malvern (now QinetiQ) had developed a method and a notation (the compliance notation) for refining Z specifications into programs in a safe subset of Ada. This was intended for and subsequently applied to the formal verification of safety critical systems for operational military applications. Support for the compliance notation is provided by the ProofPower "daz" option. An associated program called ClawZ makes it possible to obtain Z specifications from Simulink models for use in similar applications.

2 What ProofPower is *for*

ProofPower is primarily a proof assistant which facilitates the construction and checking of formal proofs in Higher Order Logic, and also, via a semantic embedding of Z into HOL, supports proof in Z. It is intended to support the development of *large* proofs, in the context of substantial formal specifications, and this makes it necessary for it to provide facilities which are not otherwise thought necessary.

The process as a whole which it supports is therefore that of creating and checking a formal specification, which consists of one or more documents containing both formal and informal materials, establishing a *coherent* logical context in which all the required specifications are in context, and providing an interactive environment in which proofs can be undertaken, intermediate lemmas stored for future use, results displayed or printed or incorporated into other documemts.

We will give a sense of the kind of support offered by ProofPower for this process using fragments from an example [2] which is distributed with ProofPower and is also separately available from the `lemma-one.com` web site.

3 Preparing a Specification

It is usual, though not strictly essential, to work with specifications in LaTeX documents, and to use ProofPower interactively during the production of such a specification for checking the formal content. The diagnostics from syntax and type-checking a specification are valuable aids in getting a specification, and are best obtained at the earliest possible moment.

Usually, a specification for an application of ProofPower to an IT system will spread over more than one document but the example we draw on now is simple enough to fit in a single document. A ProofPower document is a LaTeX document with some special extensions for formal material. Such a document can be processed by the system in various different ways which include its transformation for printing into a LaTeX document in which the formal material is in a form acceptable to LaTeX and also processing to extract the formal materials for batch processing by ProofPower.

The preparation of such a document is facilitated by a tool called *xpp* which is essentially a text editor using an extended font and a command window in which the interactive proof tool is running. The formal material is edited into the document in a near *wysiwyg* format making use of templates, or palettes or of special keyboard setups which give access to all the special characters through the keyboard.

The first step is to establish a logical context for the specification, which would normally be done by creating a new theory placed in the theory heirarchy managed by ProofPower so that all the mathematical theories or other Z specifications on which the current specification logically depends are in scope.

For this purpose the metalanguage *standard ML* (SML) is used and an SML section is placed in the LaTeXsource document which looks like this:

```
=SML
open_theory"z_library";
set_pc "wrk050";
=TEX
```

and will print in the document as:

```
SML
open_theory"z_library";
new_theory "wrk050";
```

Having placed this material in the document, it would then be normal immediately to execute it by selecting the SML material and selecting execute (either on keyboard or from menu), thus placing ProofPower in a context in which it can process the Z specification to come.

Z specifications in a ProofPower LaTeX document are not presented as the usual jumble of LaTeX macros, but are inserted using the ProofPower's extended character set in a form closely resembling the printed form of the specification.

Here is an example which appears early in [2]. The schema:

z

READ

$$
\begin{array}{|l}
OPERATION; \\
\boldsymbol{class?} : \mathbb{N}; \\
\boldsymbol{data!} : DATA \\
\hline
class? \in dom\ classified\_data; \\
class? \leq clear?; \\
data! = classified\_data\ class?; \\
classified\_data' = classified\_data \\
\hline
\end{array}
$$

is inserted in the document as: [1]

$$
\begin{array}{|l}
\ulcorner [\!| READ |\!] \text{———————————————————————————} \\
\quad OPERATION; \\
\quad [\!| class? |\!] : \mathbb{N}; \\
\quad [\!| data! |\!] : DATA \\
\quad \text{———————————} \\
\quad class? \in dom\ classified\_data; \\
\quad class? \leq clear?; \\
\quad data! = classified\_data\ class?; \\
\quad classified\_data' = classified\_data \\
\quad \text{———————————————————————————} \\
\end{array}
$$

Having entered this paragraph into the document, it can immediately be checked and stored in the current theory by *executing* it in the same way as a series of ML statements. If any syntax errors or type errors are discovered they are reported and the saving of the paragraph is inhibited. The author of the specification can then correct the erorrs and resubmit the paragraph for checking.

In this way a specification can be developed interactively until the point is reached at which the specification is sufficiently complete for propositions about the specified system to be expressed and for proof development to begin.

Typically a specification will run over several documents and the interactive checking which takes place while writing each document will be supplemented by reviews of the documents to establish whether the specifications are not only valid syntactically and type correct, but also correctly reflect the prior informal understanding of the requirements.

4 Proof in ProofPower

ProofPower is an LCF-like proof system. This means that the logic is implemented as an abstract data type in SML which has a type of theorems (THM) which can only be computed by methods reflecting precisely the rules of the logic, thus ensuring that values of type THM really are theorems.

[1] There are in fact no gaps between the characters which make up the schema box. Gaps are shown here only to make clear that the box is in fact made up from these characters. Usually a new box is inserted into a document by copying another one or by using the template tool.

Computation of theorems directly or indirectly through the abstract datatype is called forward inference, and explicit forward proof plays an important rule in effective use of ProofPower. However, the most productive primary proof idiom is backward or goal oriented proof, in which the user first identifies the theorem which he wants to prove and then works backwards from their until he reaches easily demontrable premises from which it can be derived. Behind the scenes, orchestrated by the goal package, such proofs are achieved by forward proof through the primitive rules in the abstract data type of theorems, but at the level of interaction with the user the preception is different.

We present here a small example of proof in Z drawn from the same paper as the above specification fragments. A proof is usually developed by a method similar to that for preparing a specification. A document is used as a host for a proof script which consists of SML commands to the goal package guiding it to construct the required proof.

The first step is to tell it the goal which is to be proven. This is done by passing to the `set_goal` procedure a *goal* which is essentially a sequent which has not yet been proven and hence does not have type *THM*. A sequent is a pair consisting of a list of assumptions and a conclusion, all of which are HOL terms of type BOOL which represent Z formulae. Normally a goal has no assumptions.

SML

```
set_goal ([], ⌜∀ x, y : ℤ • x ≤ y ⇒ (0 .. x) ⊆ (0 .. y)⌝);
```

The effect of executing this command is to initiate the goal oriented proof. At each stage in such a proof a tree of subgoals remains outstanding, one of which is distinguished as the *current* subgoal. The main way of progressing the proof is for the user to supply a function called a TACTIC to be applied to the current subgoal. If the application of the TACTIC is successful then it will return a new set of subgoals from which the current subgoal is provable, together with a rule which is capable of performing the derivation of the current subgoal from the new set of subgoals. In the special case that the new set of subgoals is empty, the tactic is said to have discharged the current subgoal, and the proof of the main goal is complete when the last outstanding subgoal is discharged. Once the proof is complete, the goal is made available as an object of type *THM* and can be bound to an SML identifier, saved in the current ProofPower theory, or used in the computation/proof of further theorems.

So what kinds of thing can one do in a goal oriented proof? Among the most common are:

stripping This is a general method generalising the applications of natural deduction like rules chosen by inspecting the primary connective or relation in the conclusion of the current subgoal.

rewriting The conclusion of the current subgoal can be rewritten using the assumptions of the subgoal and/or any theorems which either are or can be transformed into equations in the present context and/or any collection of *conversions* (a conversion is a computation which takes a term and yields a theorem which is an equation with that term on its left hand side).

forward inference the most common use of forward inference in a goal oriented proof is to reason forward from the assumptions of the current goal, yeilding either a new assumption, or a transient theorem which may be used in rewriting the conclusion.

This result is proven by expanding the definition of .., stripping the result and then forward chaining using transitivity of $\leq$.

SML

```
a(rewrite_tac[z_get_spec ⌜(_.._)⌝] THEN REPEAT strip_tac);
a(all_fc_tac[z_≤_trans_thm]);
```

SML

```
val le_dots_lemma1 = save_pop_thm "le_dots_lemma1";
```

References

[1] J.M. Spivey. *Understanding Z.* Cambridge University Press, 1988.

[2] ds/fmu/ied/wrk050. *Methods and Tools for the Verification of Critical Properties.* R.B. Jones, Lemma 1 Ltd., `http://www.lemma-one.com`.