ProofPower

Roger Bishop Jones

Abstract. A description of the specification and proof tool ProofPower.

Table of Contents

1. Introduction

ProofPower is a tool supporting the application of formal mathematical modelling using the specification languages HOL, Z and others. It began as a commercial software product developed by International Computers Limited as part of a collaborative research and development project partly funded by the Information Engineering Directorate of the Department of Trade and Industry. The project ran for three years from 1990 through 1992, but the development has continued since then under a variety of other funding arrangements. Much of the later work has been funded by the Royal Signals Researach Establishment and various sucessors (now QinetiQ). The software is now owned by Lemma1 limited.

Originally ProofPower was aimed at supporting the application of formal methods to the development of highly secure computer systems, but latterly most developments have been oriented toward safety critical applications, though none of its features is specific to these application domains.

The design and implementation of ProofPower drew on an impressive pedigree of achievement in the fields of Mathematical Logic, Language Design and Automation of Reasoning, and the characteristics of the tool may best be understood in the first instance through an account of its pedigree.

2. Pedigree

ProofPower is a proof assistant for Higher Order Logic which supports other formal notations, some by semantic embedding, some by other methods. Other languages for which some actual support is available include Z, SPARK and the QinetiQ compliance notation, but the design of ProofPower is sympathetic to extension to support additional notations or languages via semantic embedding and supports mixed language working.

2.1. Higher Order Logic

The primitive logic supported by ProofPower is a variant of Higher Order Logic. This logic is a direct descendant of the Theory of Types(Russell, 1908) which Bertrand Russell devised for the formal derivation of mathematics which he undertook jointly with A.N.Whitehead and published as Principia Mathematica between 1910 and 1913(A.N.Whitehead and B.Russell, 1910). Russell's original type theory was designed

to provide a logical system sufficient for the derivation of mathematics by conservative extension, with a particular concern to avoid the antinomies which had recently been found problematic in a philosophically satisfactory manner.

Russell had sought to impose restrictions for which a good rationale could be offered, rather avoid the contradictions by arbitrary constraints for which no convincing rationale could be offered. Unfortunately his rationale, based on proscription of "vicious circularity" lead to an unacceptably weak (predicative) type theory and he was forced to rescue it with an appalling hack called the axiom of reducibility. Ramsey later observed that an equivalent effect could be obtained in a simpler manner by dropping the so-called ramifications in Russell's theory after which the difficult to swallow axiom of reducibility was no longer needed. This gave the simple theory of types, which is now called higher order logic, or more precisely, ω-order logic, indicating that it has all finite orders but no infinite orders.

The simple theory of types was then given an elegant reformulation using the typed lambda calculus by Alonzo Church(Church, 1940).

The final stage in reaching the particular variant of Higher Order Logic supported by ProofPower was accomplished by Mike Gordon at Cambridge University, when he adapted the language for use in a proof assistant derived from Cambridge LCF. The adaptations which tool place at this stage were partly the kind of adaptations which are appropriate to a logic when it crossed from being an object of study by mathematicians to one applied by computer scientists, and partly logically insignificant changes which made exploitation of the code base already implemented for Scott's Logic for Computable Functions. Most of the former were also inherited from LCF.

Taking these in turn, the features which are arguably essential for practical applications included:

polymorphism In Church(Church, 1940) type variables appear only in the meta-language. In practice it is highly desirable to have them in the object language. This involves a small change to the grammer of types, and a new rule for type instantiation.

type constructors Church had two primitive types (individuals and propositions) and one type constructor (function space constructor). HOL has the same primitives but allows additional type constructors to be introduced, type constants are treated as 0-ary constructors, and provision is made for introducing type constructors of any arity by conservative means.

constants Several mechanisms are introduced in HOL for introducing new constants by means which are guaranteed conservative. The set of primitive constants in HOL (equality, implication, choice) differs from that in STT (negation, disjunction, universal quantification, description). This naturally affects the axioms of the logic, but it is easy to see that the resulting theories are the same (the axioms of HOL are provable in STT and vice versa).

2.2. The Z Specification Language

The Z specification language was developed at the University of Oxford. The language is based on set theory, in particular on the axiomatisation of set theory due to Zermelo (Zermelo, 1908), which was subsequently enhanced to yield the theory ZFC.

The enhancements to Zermelo set theory en-route to the Z specification language include:

1. an explicit axiom of regularity asserting that all sets are well-founded.
2. sharpenening of the notion of definite property used in forming new sets by separation (this sharpening arises from formlisation, zermelo's original set theory is not a formal theory)
3. the addition of a type-system together with a kind of polymorphism in the form of set-generic specifications (i.e. specifications which are parameterised by arbitrary sets)
4. the introduction of labelled products and the elaborate use of these labelled products for modelling relations (which can represent the behaviour of computer systems), and which serve as abbreviations for complete signatures (collections of variables together with constraints on those variables) and predicates (properties of signatures).

The Z specification language introduces a rich syntax for a typed set theory which makes it much more suitable for use as a specification language in the development of computer systems.

2.3. Implementation

The implementation follows a novel paradigm introduced in the Edinburgh LCF system and subsequently adopted by several direct descendents of that system and also by other implementations of proof assistants round the world. This is called the *LCF paradigm*.

The implementation is in the language Standard ML (SML), which is a modern version of the language ML first devised for the Edinburgh LCF system (there are several other modern variants of ML). This language is intended not only for use in implementing the proof assistant, but also as a language for interaction between the tool and its users.

2.4. The LCF Paradigm

The main features of the LCF paradigm were first used in the Edinburgh LCF system, a second attempt, lead by Robin Milner, at implementing a proof assistant for Dana Scott's Logic for Computable Functions. The first attempt was Stanford LCF, and the main characteristics of LCF were therefore arrived at by Milner in the light of experience with the implementation in LISP, and application, of a proof assistant for LCF.

Already in Stanford LCF (1972), the idea of using an (impure) functional language for interaction with a proof assistant was exploited. The first step in developing this idea for Edinburgh LCF was to use a typed (impure) functional language and to use an abstract data type to guarantee that any computation of a theorem would yeild a result derivable in the logic. The experience of Stanford LCF, which constructed and retained actual proofs was that these became very large, but were there only to be checked. The use of an abstract data type permitted the checking to be built in, and made the retention of the proof superfluous, so the LCF paradigm effectively treats a computation as a proof.

The replacement of LISP as an implementation language by a typed functional language created obvious difficulties in relation to the implementation of general list processing facilities. The power of LISP in part arose from the power and flexibility of general list processing functions in a type-free context. Typed programming languages were either poor in support for generic facilities (languages such as COBOL, FORTRAN, even Algol), or had very complex type system such as that for Algol68. The polymorphism adopted in ML was a startlingly simple way of adding types while retaining the ability to write list processing functions which could operate over all types of list. It created an effective compromised between the complexity of then modern languages such as Algol68 and the simple but crude power of LISP. It also also permitted another starting compromise, types and type checking without type declarations. Functions could be defined without stating their type, and their most general polymorphic type could then be computed automatically.

2.5. Standard ML

Subsequent to the design of the original ML for the Edinburgh LCF system considerable development to typed functional programming languages took place, including for example the introduction of pattern matching function definitions.

There was by this time a significant community of users of ML, as well as several variants of the language, which wanted to benefit from the state of the art in functional programming without entirely leaving ML behind.

An international effort was initiated to re-design ML an create a new standard for impure typed functional programming.

3. Functionality

What does ProofPower do?

It provides:

- Document preparation facilities (using LaTeX) for LaTeXdocuments containing near wysiwyg formal materials.
- Syntax and type checking of specificatons in HOL and Z.
- Management of a theory database in which the details of formal specifications and the results proved about them are stored.
- Facilities for computing theorems in specific logical contexts (positions in the theory heirarchy), by means which reliably check formal derivability of the theorems in the relevant context.

With the supplementary DAZ facility the capability of ProofPower is extended to support of SPARK and the QinetiQ compliance notation which supports verified refinement of Z specifications into SPARK.

A tool, CLAWZ, is also available which translated models in Simulink (a graphical modelling tool associated with the Matlab mathematical software).

ProofPower is essentially a tool which assists in the construction and checking of formal proofs in Higher Order Logic, and in various other languages which can conveniently be interpreted in Higher Order Logic.

The idea, following Russell's conception of the foundations of mathematics, is to utilise a "logic" sufficiently strong that the concepts of mathematics, or of application domains which are suitable to be modelled mathematically, are introduced by definition (or, slightly more liberally, by conservative extension).

Once these concepts have been introduced the tool will support the construction and checking of mathematical proofs of propositions involving them. The mathematics required for modelling information systems using these languages is not trivial, and the specifications of the systems (which consist of large numbers of often complex definitions) can be large. All these must be logically in context before a proof can be undertaken, and the proof is invariably primarily a work of human ingenuity, in which the more trivial labour, and the careful checking for correctness, are undertaken by ProofPower.

For this kind of work special document preparation facilities are required which can cope with the exotic notations employed in such a way that the formal specification can not only be printed, but can also be processed by ProofPower to establish a context in which proofs can be conducted.

It is essential in practice that the definitions assembled together for the specification and proof can be structured suitably, and for this purpose ProofPower maintains a theory heirarchy. A theory is a bit like a module of specification, and a dependency heirachy exists between the theories in which a theory A which makes use of definitions introduced in theory B is a descendent of B, and A an ancestor of B.

Theories include or have associate with them the following kinds of information:

Definitions of new types, type constructors or constants. All definitions must be saved in a theory before they can be used.

Theorems some of theorems which have been proven in the context of the definitions in the theory (and its ancestors). The storage of theorems in theories is entirely at the discretion of the user. Theorems are in any case a special type of value which includes contextual information which determines the scope in which the theorem is derivable and can be used for further proofs.

Proof Contexts which are bundles of resources for theorem proving and are associated with a particular theory, Typically they will contain theorems from the theory and its ancestors which are likely to be useful in proving further results, and conversions which are functions in ML capable of aspects of proof automation going beyond what can be captured in a theorem, e.g. which can determine and prove the result of an arithmetic computation and return a theorem which captures that result in an equation (e.g. $\vdash 2+2 = 4$) suitable for simplifying a proof goal by rewriting.

4. ProofPower's Innovations

International Computers Limited had a need for a proof tool supporting formal reasoning about specifications written in the Z specification language. There was at that time no established logic for the language, though the semantics for the main features of the language had been defined by Spivey in his doctoral dissertation (Spivey, 1988).

ICL already had acquired experience of reasoning about specification in Z by manual transcription into HOL using the Cambridge HOL proof assistant. In principle it seemed probable that a properly engineered semantic embedding of Z into HOL would yield an effective proof tool. It was thought however, that to base a commercial product on software which was the product of ongoing academic research would not be a good idea. Changes to Cambridge HOL would not reflect the requirements for stability for a commercial software product. In addition ICL had an interest in acquiring skills in the development of proof technology, and a re-engineering of HOL was one way to achieve that.

The proposed re-engineering was therefore not intended to be an innovative undertaking. It was inteded to yield software products which were suitable for certification permitting their use in the most demanding high security developments. However, the need to support proof in Z imposed some novely in the development, and most of the novel features in the core ProofPower system were introduced, either to make it credentials for high assurance work more conspicuous or to make support for Z possible.

Among these novelties are:

Generic Multilingual Quoting The Cambridge HOL system has facilities for parsing and pretty printing the object language HOL in which proofs are conducted. ProofPower has some generic support for embedding arbitrary languages into HOL, allowing quotations to be tagged with a language identifier, so that the underlying HOL terms could be entered or printed using various different concrete syntaxes. A single term in HOL can be presented as a hybrid quotation involving multiple languages.

Context Sensitive Proof Facilities It proved essential for smooth proof in Z that even the most basic proof facilities could be made sensitive to the principle language in which the proof was being conducted. For example, the same universal quantifier is used both in pure HOL and also in Z embeeded into HOL, but the best way to handle the quantifier in a proof depends upon what language is in use. One reason for this is that though any proof step which is

legal in HOL will also be legal for embedded Z, (such as quantifier elimination) some sound proof steps will take the user from a subgoal which is in Z (i.e. a HOL term which is the image under the embedding of some Z term or formula), to a term which is good HOL, but not in the image of the embedding, taking the proof out of the Z language.

Though introduced to permit proofs in Z to stay in Z, the context sensitive facilities (which were made sensitive to things called "proof contexts") had wider applicability. The enable all aspects of the systems proof capabilities to be enhanced for support of the various theories developed.

References

A.N.Whitehead and B.Russell: 1910, *Principia Mathematica*. Cambridge University Press. 3 vols.

Church, A.: 1940, 'A Formulation of the Simple Theory of Types'. *Journal of Symbolic Logic* **5**, 56–.

Russell, B.: 1908, 'Mathematical Logic as based on the Theory of Types'. *American Journal of Mathematics* **30**, 222–262.

Spivey, J.: 1988, *Understanding Z*. Cambridge University Press.

Zermelo, E.: 1908, 'Investigations in the Foundations of Set Theory I'.