The Story of ProofPower

Rob Arthan
rda@lemma-one.com
Roger Bishop Jones
rbj01@rbjones.com

February 11, 2005

Contents

1 Introduction

The development of tools to assist with mathematical activities is a subject with a long and fascinating history. Since the earliest days of electronic computation, researchers have endeavoured to produce mechanised tools to assist with the development of formal mathematical theories. The possibility of using formal mathematics to improve the rigour of hardware and software systems design has spurred this research. The result has been the development during the past 20 years or so of practical systems to support rigorous formal mathematics and its applications.

ProofPower is one such system for mechanised formal mathematics. It is a tool supporting formal specification in Higher Order Logic. It also supports, by semantic embedding [4] and other techniques, various other languages, including the Z specification language. ProofPower is an LCF-like proof tool [6], originally conceived as a re-engineering of the Cambridge HOL system, which was built from Cambridge LCF (mainly the work of Larry Paulson [8]) by Mike Gordon *et al.* [3, 5] for use in his research into hardware verification. As well as syntax- and type-checking specifications, and managing a hierarchy of theories in which these specifications are stored, it provides the user with a high level functional programming language (Standard ML [9]) for constructing (and checking, on-the-fly) proofs in its core logic. The logic is a polymorphic variant (following Robin Milner [10]) of the elegant and simple reformulation by Alonzo Church [2] of the *Theory of Types* [11] which Russell devised for use in *Principia Mathematica* [1]. It is adequate not just for the hardware verification domain but for the development (and application) of pretty much all applicable mathematics.

The combination of this logical strength in the object language, and the power of the meta language, has resulted in the Cambridge HOL system, and the various other proof tools which it has spawned or inspired (including ProofPower), securing a global user base, applying these tools to a range of applications which could not at first have been imagined. ProofPower combined from its earliest days these important elements of the LCF-HOL line of research with the somewhat different subculture of formal methods originating in Oxford and embodied in the Z specification language [13, 7]. Many of the distinctive features of ProofPower arise from creative friction between these two subcultures. To this were later added further elements, support for additional languages, orientation to distinct application domains. The end result is a tool which, while bearing the marks of its intellectual history, is unique in its capabilities.

The purpose of this article is to relate something of the history of ProofPower, and to explain some of the rationale behind its design. *En passant*,

we will look at some of the important systems to which ProofPower is related and also some of the applications that is has found. We hope that it may also stimulate interest in the application and continued development software support for formal mathematics.

2 Beginnings

The original motivation for developing ProofPower was in information security. Mindful of the threat to its national security which was posed by insecure information processing systems, the US government established a certification regime for secure systems. The guidelines, known as *TCSEC* (Trusted Computing System Evaluation Criteria) and were published in 1983 and mandated the use of *formal methods* where the highest levels of certification were sought. In order to continue in its privileged position in relation to access to sensitive information, the UK government would either have to buy its computer equipment from manufacturers who had completed the required US certification, or they would have to put in place their own comparable system of certification and encourage or contract the UK computer industry to develop systems for certification under that scheme. The UK was the first country to follow the US in this regard with its own national certification scheme, but this eventually lead to a harmonised regime across Europe (the ITSEC standards).

While these standards for information security where being established, the UK government was placing contracts with industry relating to the development of secure systems and also to the establishment (in industry) of the capability in formal methods thought essential for developing the most secure systems. UK based International Computers Ltd. (ICL) won some of these contracts, and became aware of the need for competence in formal methods. In 1985 the ICL established a Formal Methods Unit within in its Defence Technology Centre to supply necessary capability.

Z had already been identified as the preferred formal specification language for work on government secure systems and so familiarisation with Z was an early priority, as was gaining experience in the proof technologies which seemed most promising for the kind of work expected. The two proof tools which were being evaluated by the formal methods team were NQTHM (otherwise known as the Boyer-Moore prover after its authors) and the Cambridge HOL system, both of which had also been used at RSRE Malvern for the formal treatment of digital hardware.

Of these two the Cambridge HOL system seemed to offer best prospect of success in reasoning about specifications in Z, in default of any proof tool

for Z itself.

The main factors in this assessment were:

- firstly that the language HOL (Higher Order Logic) supported by the Cambridge HOL system was much closer in logical expressiveness to Z than was the impoverished first order logic of NQTHM.

- secondly that the LCF paradigm, giving the user a powerful functional programming language for programming proofs and for other kinds of extension seemed to promise a greater flexibility of the tool for adaptation to tasks for which it had not been originally intended.

By contrast with work in HOL, for which the concrete syntax was entirely coded in ASCII, Z had a culture of pencil and paper specification using not only more customary logical symbols but also many other special mathematical symbols and also a special graphical layout in which formal specifications were presented in various kinds of boxes and other structures. It was also a novelty of the Z culture that specifications were presented embedded into into formal textual discourse. The relationship the formal and the informal material in computer documents (usually programs of course) was inverted, instead of adding annotations into the formal material, Z added formal materials into a proper English document.

Cambridge HOL needed no special facilities for document preparation, expecting HOL scripts to be treated just like a program. Specifications in Z however were intended for a readership who might admire a beautifully handwritten English document with interspersed mathematics, but would look askance at a program listing. An element of our initial tasks in developing ways of reasoning formally about Z was therefore to provide document preparation facilities.

The method adopted was to manually transcribe a specification from Z to HOL, maintaining as close a correspondence as possible between the original specification in Z and the translated specification in HOL.

Sun workstations came with a font editor and it proved straightforward to edit into a standard font a good selection of the logical and mathematical symbols used by Z. Cambridge HOL could be wrapped in a couple of UNIX filters which translated between these symbols and the corresponding ASCII sequences used by HOL.

As well as crudely effecting a partial reconciliation between the lexis of Z and HOL, some more fundamental issue were addressed at this early stage.

The Cambridge HOL user community attached value to the fact that HOL is the kind of logical system in which mathematics and its applications could be undertaken by conservative extension (it provides a *logical foundation* for

mathematics). The introduction of new axioms was supported by the tool, but frowned upon by its users except in those very rare circumstances in which it was not technically avoidable.

The Z notation had no apparent concern for this matter, and the approved specification style freely admitted the introduction of "axiomatic specifications" which combined the introduction of new (so called) global variables (constants) with an axiom stating any desired property.

At ICL we were fully sympathetic to the HOL tradition harking back to *Principia Mathematica* but could not eschew the use of axiomatic specifications in Z. We therefore translated Z axiomatic specifications into a good but not absolutely faithful approximation in Z by a (behind the document) translation into a definition using the HOL choice function. When we later came to a substantial project in which formal verification would be critical a special version of Cambridge HOL was produced for that project in which, along with a small number of other security enhancements, a feature was introduced which make such conservative translations more fully faithful to the meaning of the original specification. An equivalent facility was later introduced to the main Cambridge HOL in response to our enquiries on this point.

The ICL Formal Methods Unit applied formal methods to the development of secure systems, using transcription into HOL for formal proofs for three years before the opportunity arose to proposed a substantial program of work on the development of tools to support this kind of work. This work included the formal aspects of a complete hardware development and manufacturing project which was undertaken so as to achieve certification at the highest possible levels of assurance. The formal work involved the development of a formal machine checked proof that the system met the formal statement of the critical security requirements. The end product was certified at ITSEC level 6 and was for many years the only product to have achieved this level of certification.

Transcribing Z into HOL with care to ensure that meaning is preserved and then reasoning formally with the resulting specification, all the while wondering how the methods of transcription might be made more complete and systematic to the point that they could be implemented in an embedding is not a bad way to get a good understanding of the semantics of Z. We were therefore reasonably equipped when the opportunity to start developing formal methods tools in earnest to make some substantial strides forward.

3 HOL

In 1990, ICL began a programme of research in collaboration with Program Validation Ltd. and the Universities of Cambridge and Kent (the *FST* project). While proof support for Z was very high on ICL's list of priorities, it was not clear what level of support for Z would be achievable. The project proposal was therefore almost mute on this topic, and put forward as ICL's task the production of a proof assistant for the HOL logic engineered to standards appropriate for its use in commercial developments of software for certification at the highest levels of assurance. Best achievable support for Z was for ICL a tactic objective. Additional declared desiderata were, improved usability of the tool and productivity in its intended uses.

These desiderata were to be realised by:

- Development following industrial product quality standard methods (including inspection of detailed design).
- The application of formal methods to the design of the new tool. Formal specifications of the syntax and semantics of the HOL language were prepared, and of the "critical aspects" of the design of the proof assistant (viz. those which were pertinent to the risk of the tool accepting as a theorem some sentence which was not legitimately derivable in the relevant logical context).

Work began on a disposable prototype proof tool several months before the official start of the FST project at the beginning of 1990, so a working proof tool was available very early in the project. This prototype was used as a test bed which underpinned the development of a product quality tool. This tool which was subsequently christened ProofPower is *essentially* a re-engineering of 'classic' Cambridge HOL (HOL88). The salient features of this re-engineering are listed below.

[**Note: Would the following paragraphs be better as sub or subsubsections so that they may appear in the content listing? Should we list them first? Shall we weed some out?**]

3.1 Use of Standard ML

Standard ML failed to draw in the entire ML community, but adopting standard ML for this development still looks like the right choice.

3.2 Formal Specification of Critical Features

Key features of the tool, notably the logic it supports and the logical kernel which ensures that the tool will only accept as theorems sentences which are derivable in that logic, and formally specified (in ProofPower-HOL). The logical kernel includes critical aspects of management of the theory hierarchy.

3.3 Systematic Software Development

ProofPower has a detailed design, implementation and test documents for each software module. Since all these documents contain SML code which is either compiled into or executed to test ProofPower, (detailed design documents include signatures) there is little tendency for the detailed design to slip out of sync with the implementation.

3.4 Name Space Improvements

One feature of LCF-like provers is that most of the tools used to implement the prover are accessible to the user through the metalanguage namespace and may be re-used in extending the capabilities of the system. This make for a rich and powerful environment for programming proofs, but also for a very large namespace in which the user may struggle to find the features he needs. It was a design objective of ProofPower to ameliorate this situation in modest ways.

The general policy was as follows:

(a) For low level features needed for efficient coding of new proof facilities the coverage of each kind of facility should be complete and systematic, so that the names of the various individual interfaces can often be obtained by following some obvious rule. Elementary examples of such conventions are the systematic use of "mk_" and "dest_" prefixes for syntactic constructors and destructors.

(b) User level proof facilities (e.g. tactics) should so far as possible make it unnecessary for the user to select a specific tool. An elementary example is *strip_tac* which inspects the principle logical connective of the current goal and perform an appropriate proof step, making it less likely that the user will have to know the name of the low level tactic which deals with each individual connective. The tactic of the same name in ProofPower is considerably extended in power, addressing not only principle logical connective of a formula but also in many cases the relation of an atomic formula. It is context sensitive, applying

more rules in richer logical contexts and, for example, in the case of membership predications will be able to apply in appropriate contexts a range of different rules for the various kinds of structure of which membership can be asserted (e.g. membership of an intersection will be transformed into a conjunction of membership assertions).

(c) documentation of the namespace in the reference manual is generated automatically from the detailed design documentation, and includes a KeyWord In Context index making it easy to discover all the names which contain a particular substring. Names are usually compounded from a sequence of elements related to the function of the name.

(d) the use by ProofPower of special logical and mathematical symbols extends to the names in the standard ML namespace (of which ProofPower effects a lexical variant with an extended character set).

3.5 The Theory Hierarchy

For theorem proving in the context of large specifications some way of structuring the logical context is desirable, and in LCF-like tools this is done through a hierarchy of "theories", which in this context are a kind of hybrid between the logician's notion of a theory and the computer scientist's notion of a module. In normal use a theory is the largest unit of conservative extension of the logical system, though in extremis extensions not know to be conservative will also be permitted and recorded as such. It is important for the preservation of logical coherence that when changes, other than mere additions, are made to the hierarchy everything which logically depends upon the material changed is modified or invalidated at that point. In the Cambridge HOL system this was accomplished by deleting theories, and any change to a theory could be accomplished only by deleting that theory in its entirety and all its descendants, and then reloading the theories from modified scripts.

Flexibility in this matter depends upon how much detailed information is held about the interdependecies. Holding more information permits greater flexibility, but complicates the data structures involved in theorem proving. When LCF was first developed computers were very slow, and one put up with an inflexible system, for the sake of faster inference.

The ProofPower theory hierarchy is designed with a greater degree of flexibility, falling short of the full flexibility which might flow from the fullest recording of dependencies. A definition or specification can be deleted from a theory without deleting the entire theory. It is however necessary to delete

(and reload if required) all the definitions in the theory which took place subsequent to the one under modification (and those in theories lower in the hierarchy).

3.6 The Logical Kernel

Cambridge HOL, as well as the inferences rule of HOL provides a back-door "mk_thm" which allows any sequent to be made into a theorem and thereby rendering the logic technically inconsistent. In a tool intended for use in developments subject to formal evaluation this is difficult to defend. It is of course no feat of engineering to omit this feature, the rationale for which still escapes me even though some have felt it worth defending. It is I believe now used in Cambridge HOL to check the validity of intermediate states of the subgoal package.

3.7 The Subgoal Package

LCF-like provers are implemented using an abstract data type to implement a type of theorems, the only constructors of which are the rules of the logic (let us assume for present purposes that axioms are rules without premises). In all the cases of interest here the underlying logic is in fact an asymmetric sequent calculus (in which the sequents have a list of assumptions and a single conclusion) permitting a forward (or Hilbert style) proof system which has some similarities with a natural deduction system (and can be made very similar with the benefit of derived rules).

The full convenience of "backward proof" is then realised using a "subgoal package". The user starts a proof by passing to the subgoal package the sequent which he wishes to prove (as a sequent rather than a theorem, since it will not be available as a theorem until after it has been proven). At each step in the proof which follows the subgoal package presents to the use a single "current goal" and the user nominates to the subsoil package a "tactic" to be applied to the subgoal. When a tactic is successfully applied to a subgoal it breaks it into zero or more further subgoals accompanied by a rule which derives the subgoal (as a theorem) from the list of theorems corresponding to the new subgoals. If a tactic can solve a subgoal, then its introduces no new subgoals and offers a proof which requires no theorems to be supplied to it, and which can therefore be invoked by the subgoal package to obtain the desired theorem. If the user persists in supplying relevant tactics to the subgoal package until all subgoals are in this way discharged, the subgoal package will be able to construct a proof of the original goal by composing the proofs obtained from each of these steps. This proof will be

a rule which when applied to the empty list of theorems will compute and return the desired theorem.

Tactics are not infallible, not only may they fail to offer a step in the desired proof at all, they may make an offer which they later fail to redeem. The failure of a tactic to deliver on its promise is a bug in the tactic, and these are rare. Tactical programming is often done by users, and it is very inconvenient to discover an error in a tactic only at the point of completion of a large proof which uses the tactic, not least because the diagnostics available at this point may not be good. This kind of thing did happen in the Cambridge HOL system which we were using before the development of *Product*.

The subgoal package design in ProofPower is immune to this problem. Instead of remembering a tree of subgoals and proof functions the state of the subgoal package is coded up as a theorem in which the assumptions are (codings of) the outstanding subgoals, and the conclusion is a coding of the target goal (a constant is used which mimics the semantics of the sequent turnstile). The construction of this subgoal package state theorem involves invocation of the proof function at the same time as the tactic is invoked so that its failure is detected immediately.

In some versions of Cambridge HOL the desired checking of the proof function takes place at the point at which it is produced using "mk_thm" to obtain the necessary premises. This method however does depend upon this hole in the abstract data type, which might undermine trust in the proof tool.

3.8 document preparation

It is the Z style of literate formal specification which has determined that ProofPower is oriented around processing of formal texts extracted from LaTeX documents. The document processing facilities are based around a program called "sieve" which processes documents according to the instructions in a sieve steering file watching for tags in the document which introduce the many different kinds of formal material which may be included in such a document. The same kinds of documents may be processed in different ways for different purpose, for example to yield pure LaTeX for printing, or scripts suitable for loading the formal material into the ProofPower proof tool for processing to enter specifications into the theory hierarchy, to generate and check formal proofs and store the resulting theorems. It is hard now to see how we could have managed without this kind of machinery, but it remains the case that the initial impetus to this manner of working came from Z, and that the academic versions of HOL still work directly from ASCII text files

and do not involve themselves in these matters.

4 Z in HOL

The prototype proof tool was also used in some more speculative, but ultimately fruitful, research into support for Z. Though at the time of writing the project proposal no complete injection of Z into HOL was known, in the first few months of 1990 a mapping was devised and partially specified in Z. This was thought to be semantically correct (up to a bit of debate about the semantics of undefinedness) and practicable.

The idea of a semantic embedding of one language into some other language is to treat the first language as if it were an alternative syntax for certain expressions in the other. The idea is that some capability in respect of the second language is thereby transferred to the first. In this case the primary capability of interest is proof construction and checking.

In the case of embedding the language Z in HOL, though in some deep logical sense the languages are closely related, in superficial matters such as syntax they are worlds apart. A semantically correct embedding of the whole of Z into HOL would be relatively complex, and in many aspects of functionality a proof tool would have to be customised to Z in order to achieve reasonable levels of usability and productivity.

The advantages of providing proof support via an embedding are few but substantial. One important advantage is that by this means proof capability can be realised where the semantics are known but the proof rules are not known. Soundness of inference is guaranteed by the soundness of the proof system for the target language, provided that the embedding is semantically correct. A second advantage is that sharing between the languages of that most precious and costly item, theorems, is maximised. Thus a theory of integer or real numbers developed for HOL will be substantially re-usable in Z, together with domain specific proof automation such as a linear arithmetic prover.

Z was not the only other language for which support might be needed, though it was thought to be the most important (in terms of prospective business), so a generic tool was desirable. A syntactically generic approach might have been adopted, but we decided instead to aim for genericism via embedding.

4.1 Some Generic Multilingual Features

In this section, as well as describing some of the key features of the support for Z in ProofPower the effect on the underlying core implementation of HOL will be picked out, not only specifically of the need to do Z, but also of the perceived need for a degree of genericism.

4.1.1 Term Quotations and Pretty Printing

The dialogue with the proof tool takes place through the interactive metalanguage ML. Cambridge HOL had special mechanisms to make invocation of an object language parser straightforward in the form of quotations marks for this purpose. Metalanguage quoting in object language terms is also supported, allowing the insertion into an object language term of an expression of the metalanguage of type *TERM*.

In ProofPower this kind of facility is extended in two main ways.

Firstly the character set is extended so that the dialogue includes the most commonly used logical and mathematical symbols. This is done by coding up the special characters into strings of characters which are acceptable in standard ML.

The effect is that not only the quoted object language terms may contain these special characters, but also the ML names, so that ML names may be chosen which directly relate to the symbols in the language, e.g. $\Rightarrow\_elim$, $\forall\_intro$.

The object language quotation facilities in ProofPower include not only the primary object language HOL and embedded ML, but also the Z language, and are designed to allow other languages to be added. Full multilingual mixed language parsing and pretty printing is supported. HOL and Z can be mixed together in a single term quotation, fragments of HOL being included inside Z or vice-versa. Of course there are constraints on what is allowed and this is largely controlled by the type system and the injection used by the embedding of Z in HOL of the types of Z into those in HOL. A HOL term quoted inside a Z term must have a type which is in the image of the injection and its position in the surrounding Z term or predicate must be consistent with that type. To make pretty printing of a term possible constants are associated with languages, and this information controls the selection of pretty printers for the different parts of a term. Mixed language terms are not normally encountered in Z proofs, the proof facilities are smart enough to keep the proof in Z. However, tactical programming will frequently involve programming transformations which provide an inference within one language using transformations passing through terms which do not belong

to that language (the primitive rules of the language will not generally stay in the image of a language embedding).

ProofPower comes with a package of development software which includes an SRLP parser generator which can be used in constructing a parser for a new embedded language.

4.1.2 Proof Contexts

"Proof contexts" are a general mechanism for making all aspects of proof support sensitive to the language in which reasoning is being conducted. They are essentially packages of parameters to the various proof tools; selection of proof context can have radical effects upon the behaviour of the proof facilities. Though introduced to make embedded languages work smoothly, they are used with a finer granularity of context, making it possible to customise proof automation to particular theories.

When prototyping of support for Z on the first prototype ICL HOL reached the point of attempting goal oriented proofs using the subgoal package it was necessary to make the behaviour of *strip_tac* sensitive to the language so that it could handle correctly the Z universal quantifier. As soon as the possibility of making *strip_tac* context sensitive is considered the question what it can beneficially be used for becomes open. ICL were not involved in the early development of the LCF system of which *strip_tac* appears to be a highly used historical remnant. *strip_tac* looks like it is an incomplete attempt to provide a tactic which knows the basic natural deduction rules for the predicate calculus and automatically applies the rule relevant to the current goal, provided that can be done without extra information (such as an existential witness). This picture makes more sense in terms of the original LCF logic, in which there were fewer logical connectives than there are in HOL. If this idea is thought through it becomes apparent that a natural generalisation of *strip_tac* can be produced which is complete in respect of the propositional calculus (when repeated).

Z is based on set theory, and a natural systematic approach to proof in Z is to characterise each of the set-valued constructs in the Z language extensionally. Set theory used in this way fits well into a natural deduction framework. A simple example is the handling of intersection. A goal which is a membership assertion of which the right hand expression is an intersection can be transformed into a conjunction of membership statements. Treating intersection in this way permits its handling to be integrated into the natural deduction like method provided by *strip_tac*. In general, if the semantics of the set-valued Z terms is given extensionally as an equivalence statement in which an assertion of membership in the construct is said to be equivalent to

some formula in which that construct does not occur (though its constituents will), then these equivalences theorems provide extensions to the stripping behaviour, or to the default behaviour of rewriting facilities. The effect of systematic adoption of these methods is to automate the transformation of quite elaborate expressions in Z's set theory into predicate calculus in which the set theoretic vocabulary has been eliminated.

4.2 Embedding Z in HOL

A typed polymorphic set theory is logically similar in strength to a polymorphic simple theory of types, and so in principle one ought to be able to interpret Z in HOL. The challenge is to devise an interpretation which works well in practise, i.e. which can be implemented in a proof tool yielding convenient efficient support for proof in Z.

Interpreting one logical system in another is something which logicians do for theoretical purposes. The kind of interpretation needed to provide proof support for one language in another is not exactly the same kind of thing. A typical reason for a logician to interpret one system in another is to establish their relative proof theoretic strength (or obtain a relative consistency result). For such proof theoretic motivations semantics is not important, these result are relevant even to uninterpreted formal systems. What is essential in these proof theoretic applications is well defined deductive systems, it is the theorems which are "interpreted".

In the context in which ProofPower support for Z in HOL was implemented this was not the case. The semantics of Z was known reasonably well, by extrapolation from the partial semantics provided by Mike Spivey in his doctoral dissertation, published as the book *Understanding Z* [12]. But there was no comparably extensive documentation of a deductive system for Z, and there were some very novel features in the language which might be expected to make the establishment of such a deductive system to be fraught with problems. In this context a semantic embedding of Z into HOL had the great advantage that it promised sound reasoning in Z via derived rules of the well established HOL logic, in a fail-safe manner (checked by the ProofPower-HOL logical kernel).

The kind of interpretation which is of interest to us here is therefore a semantic interpretation, of a kind which is now known as a semantic embedding. The discussion which follows has more the flavour of computer science or software engineering than of mathematical logic and proof theory.

4.2.1 What Kind of Embedding?

It is possible to approach this in several different ways, and not very easy to decide which of these is best (to some extent it depends upon the intended applications).

There are two interconnected initial choices which must be made. Firstly, between a *deep* and a *shallow* embedding.

In a deep embedding the semantics of the embedded language is completely formalised in the supporting language, in this case, the semantics of Z would be coded up in HOL. This would involve introducing inductive datatypes in HOL corresponding to the kinds of phrase in the abstract syntax of Z and defining valuation functions over these types yielding values in suitable semantic domains, all defined in HOL. Each sentence in Z can then be translated into the sentence in HOL which asserts that the image of the representation of the abstract syntax of the sentence under the semantic mapping takes the appropriate semantic value for a true sentence.

In a shallow embedding the mapping from the interpreted to the interpreting language is defined in some suitable metalanguage rather than (as in a deep embedding) in the interpreting language. For each constructor in the abstract syntax of Z, a constant in HOL is defined which captures the semantics of that constructor. Phrases in Z which are made with that constructor are mapped by a function defined in the metalanguage ML to terms in HOL which are applications of the HOL constant which captures the semantics of the phrase constructor. The operands of the constructor in the translated expression are the translations into HOL of the constituents of the Z phrase. Thus, in a shallow embedding, the most of the detail of the semantics of Z is coded into HOL constants, but the actual semantic mapping is defined in the metalanguage.

A second important choice concerns the correspondence between types in Z and types in HOL. Thought there are doubtless compromises which might be considered, at the extremes there are the possibility of choosing a distinct type in HOL to represent each type in Z, or the possibility of using a single type in HOL to represent the entire value space of Z.

These two choices are interconnected in that a deep embedding requires there to be at most one type in HOL for each phrase type in Z (phrase types are things like *formula* or *term* and are therefore much coarser than the types in the Z type system, which are all types of Z terms).

This leaves us with a choice among the three following possibilities:

- deep embedding
- shallow embedding into small number of types

- shallow embedding with type injection

We will pass over the second alternative here and mention some particular difficulties and benefits of the other two.

- A deep embedding allows reasoning about the embedding (i.e. about the semantics of Z) in HOL, but requires a non-conservative extension to the HOL logic. A more tangible disadvantage is that questions of type correctness in Z which are essential for sound reasoning will be pushed from the metalanguage into the object language and may make reasoning in Z more complex. As against that, the difficulties which will be noted below in relation to use of a type injection are avoided.

- A shallow embedding using a type injection gives a closer relationship between the type systems of Z and HOL, permits the embedding to be undertaken without strengthening the HOL logic, and may involve less reasoning about types during proofs. Meta-theoretic reasoning about the interpretation of the language as a whole is not possible, because the semantic functions are not defined in the object language (and in this context the metalanguage ML is only supported for evaluation, not for deduction). This kind of embedding is suitable for reasoning in the interpreted object language, which is what is needed in the application of formal methods. Though the type-checking of parsed expressions demands a type checker customised to the interpreted language, computations involving the resulting values are type-checked automatically by the type rules built into the abstract data type for the interpreting language. Considerations relating to type correctness are less likely to intrude into object language proofs as side conditions, possibly reducing the cost of proof. A particular difficulty here is that the schema type construction does not map easily into HOL, and we end up having to use a family of type constructors to get the type injection.

Without practical experience of the workings of these different methods with these particular languages it is not easy to know which would be best.

For ProofPower the shallow embedding with type injection was chosen, this has worked pretty well, but we still don't know for sure how the other approaches would have worked out.

4.2.2 The Type Injection

The main problem in constructing a type injection is the fact that the schema type constructor in Z takes as its parameter a finite map from component

names to component types, whereas type constructors in HOL take a finite sequence of types, and cannot be supplied with a map. The Z type system is anomalous in relation to schema types and the operations over these objects, since schema operations do not have a single type in the Z type system, not even a polymorphic or generic type, but have to be considered either as having a family of types indexed by compatible operand signatures or as consisting of family of operators, each having a different type.

To deal with this in the injection into HOL a bijection between the types in Z and HOL is achieved even though there is no bijection between the type constructors, and families of constants are used for the schema operations.

The bijection is achieved using a family of constructors in HOL for the schema type constructor. The signature of a schema type is partly coded into the name of the type constructor, which contains a canonical encoding of the names in the signature. The types associated with the names in the signature are passed as parameters to the HOL type constructors in a canonical order determined by the names of the components.

The power set constructor is easily constructed in HOL, sets are represented by boolean valued functions.

Generic types in Z are mapped to function types. Thinking of a Z generic type as a tuple of formal type parameters together with a Z mono-type in which these type variables may occur, the image of such a generic type is a HOL function type in which the domain type is a tuple of power sets of type variables corresponding to the formal generic parameters, and the range type is the image under the type injection of the Z mono-type.

4.2.3 Mapping Formulae and Terms

The broad pattern for the mapping of formulae and terms is as follows.

Where possible, a construct in Z is mapped directly to the corresponding construct in HOL. This happens mainly for the propositional connectives.

Otherwise a new constant or family of constants is defined for the constructor in the abstract syntax of the Z language which correctly captures the semantics of that part of the Z language. The semantic mapping is then a primitive recursion over the abstract datatype.

There are a small number of important features of Z which complicate this picture. They are:

- constructs whose value is undefined
- constructs in which occurrences of variables are hidden or implicit
- variable binding constructs

- generic values and their instantiation

These are discussed in turn in the following sections.

4.2.4 Undefinedness in Z in HOL

ProofPower adopts the simplest treatment of 'undefinedness' in Z which is consistent with the semantics in the Z standard [7]. In effect definite description is taken as the primitive undefinedness handler, and is defined as it might be in a pure first order set theory, similar to the use of a choice function in that context, Function application can then be defined using definite description, and undefinedness is nipped in the bud without the need to use an 'undefined' value. This has the effect of maximising the cases in which equational theorems in Z can be used unconditionally for rewriting a goal and hence of reducing the cost of proof.

4.2.5 Hidden Variables

A novel feature of Z is the ability to use the name of a schema as an abbreviation for the defining predicate of the schema. This is in addition to the possible use of that name as the name of the set of bindings denoted by the schema. When the name is used as a predicate rather than as a set, the effect is as if a formula were substituted at the point of the schema reference in which the names in the signature of the schema appear as free variables. It is rather like using the schema name as a macro. When such a construct is mapped into HOL, to get it semantically correct it is necessary to make the occurrences of the variables explicit.

There are three ways in which these implicit variables appear in Z. The first is in theta terms, the others are in the use of schemas as predicates and as declarations.

In a "theta term" a schema name is preceded by the Greek letter θ and this expression denotes the binding which has the type corresponding to the type of the schema (i.e. the type of the bindings which are the members of the schema) and whose components have the value which the free variable of the same name takes in the context of occurrence of the theta term. This is dealt with in the mapping by using in the image the appropriate binding constructor with all the variables as arguments, i.e. in the image the variables are all made explicit. They are introduced by the injection, and discarded by the pretty printer when the theta term is formatted for printing as Z.

A schema used as a predicate is semantically equivalent to the assertion that the corresponding theta term (in which the names in the signature are free variables) is a member of the set denoted by the schema. A schema used

as a declaration is semantically similar to its use as a predicate, at least as far as the predicate implicit in the declaration is concerned. Its other role is in determining the bound variables, which will be discussed below in the treatment of variable binding constructs.

4.2.6 Variable Binding Constructs

The variable binding structures in Z all admit, instead of single binding occurrences of variables, an arbitrary signature, which will include set constraints and may include the use of schema expressions as declarations. These must all be mapped down in a semantically correct way to a language in which there is only one variable binding structure (the lambda expression) which binds a single variable subject only to a type constraint.The image of a variable binding structure must include a nested lambda expression in which all the names are bound which are bound by the Z binding (explicitly or implicitly. In the body of this expression will appear the translation of all the Z which is in the scope of the binding, in which all semantically relevant information is made explicit. If this involves several constituents then these must be combined together in the body of the lambda expression in such a way that they can be separately accessed as necessary by the semantic constant corresponding to this kind of Z construction (which will be applied to the resulting lambda expression). Because these variable binding constructs bind arbitrary numbers of names the type system makes it impossible to code up the semantics in a single semantic constant, and a family of constants indexed by the number of variables bound is usually required.

A good example is the Z lambda expression and we will therefore work through this in some detail.

The lambda expression in Z has up to three explicit top level constituents, which are:

d the declaration part or signature

p a predicate which further constrains the domain of the required function

b the body of the lambda expression giving the value of the function

The translation of the declaration part must yield three separate semantically significant values. The first is the set of names which are bound by the declaration, together with the types inferred for these variables by the Z type checker. The second is the predicate implicit in the declaration, roughly the predicate which asserts that each of the names is a member of the set of which it was declared to be a member, or, when combined with other declared

names in a binding is a member of a schema used in the declaration. The third is a tuple of variable names formed according to the prescribed rules which indicates the structure of the required arguments to the function. This latter is implicit in the syntactic form of the declaration part of the Z lambda expression, but since the syntactic form of the HOL lambda expression has no such semantic significance, this information must be rendered explicit by the mapping.

The method of combining constituents into a single value for use in the body of the lambda expression is to combine them as a binding. A tuple would have done just as well, but the use of a binding allows slightly suggestive component names to be used. In the case of the lambda expression the component names are the letters used above in listing the explicit constituents, together with the letter "t" for the tuple implicit in the Z declaration. The bound structure is therefore a higher order function which takes values for the bound variables in turn and which then yields a binding the components of which give:

d whether the values of the variables satisfy the predicate implicit in the declaration

p whether the values of the variables satisfy the explicit predicate

t the value in the domain of the required function which corresponds to the values of the variables

b the value of the body of the lambda expression, and hence of the function if it is defined at this point

The semantic constant which is applied to this function must convert it into a set of ordered pairs which is the graph of the required function. An ordered pair p will be a member of this set if there exists an assignment of values to the bound variables which, when supplied to the function, gives a binding whose b component is the second element of p, whose t component is the first element of p and whose d and p components are *true*.

All this can be observed interactively by the ProofPower user by entering a lambda expression and then taking it apart. We will do this here, illustrating several features of ProofPower in the process. The inclusion of formal material in this document is done in the normal way that ProofPower supports for document preparation, with the interactive proof tool available to process this material while the document is in preparation.

For the task in hand we must first set the context appropriately for working in Z, which we do by opening the theory which encompasses the whole

of the Z library, and setting a suitable proof context. We then create a new theory which will contain any specifications or theorems which we want to store there.

SML

```
open_theory "z_library";
set_pc "z_library";
new_theory "temp";
```

To delve into the mapping of Z lambda expressions into HOL we first enter a Z lambda expression, binding it to an ML name as follows:

SML

```
val zle = ⌜λ x:ℕ; y:ℤ | x > 1 • y * x⌝;
```

Note the use of subscripted "Quine corners" for quoting object language terms. Bare, unsubscripted Quine corners are used for terms quoted in HOL. The subscripts are not necessary on the closing quote.

The effect of executing this SML statement is to bind to the name *zle* a value of type TERM which is the image of the Z under the mapping, and the SML system will then display the result of doing so, as follows:

```
val zle = ⌜λ x : ℕ; y : ℤ | x > 1 • y * x⌝ : TERM
```

Since the term is recognised as in the image of the Z embedding and Z is the preferred language in the present context, the term is displayed using the Z pretty printer, and we are left in the dark about the details of the interpretation. However, we can get information about the mapping by taking the term to pieces.

SML

```
dest_z_term zle;
```

```
val it = Zλ (⌜ML⌜decl_of⌜[x : ℕ; y : ℤ]⌝⌝⌝, ⌜x > 1⌝, ⌜y * x⌝) :Z_TERM
```

SML

```
dest_simple_term zle;
```

```
val it =
  DApp
    (⌜⌜$"Z'λ[2]"⌝⌝,
      ⌜⌜λ x y
      • ⌜(d ≙ ML⌜decl_of⌜[x : ℕ; y : ℤ]⌝⌝, p ≙ x > 1, t ≙ (x, y),
           v ≙ y * x)⌝⌝⌝) : DEST_TERM
```

Before we go any further I had better explain the meaning of the Z lambda expression. A lambda expression denotes a function, which is, in Z, a set of ordered pairs or 2-tuples, where a 2-tuple is in fact the same as a binding with two components named "1" and "2". Of the three parts of the lambda expression the first two between them determine the domain of the function, which in this case is also a set of ordered pairs of integers of which the first must be greater than 1. The value of the function at some point in the domain is given by the value of the third part of the lambda expression.

The following conversion shows for our lambda expression the equivalent set comprehension. A conversion is a function which takes a term and returns a theorem which is an equation whose left hand side is that term. Conversions are used extensively in rewriting, particularly for providing a set of equations which cannot be expressed in higher order logic as a universally quantified equation, as in this case.

SML

```
|z_λ_conv ⌜z λ x:ℕ; y:ℤ | x > 1 • y * x⌝;
```

```
|val it =
|   ⊢ (λ x : ℕ; y : ℤ | x > 1 • y * x)
|       = {x : ℕ; y : ℤ
|         | x > 1
|         • ((x, y), y * x)} : THM
```

Note that in a set abstraction in Z the value after the '•' is a proforma for the values which are to be in the set. There is one such value for each combination of values for the bound variables (the ones in the signature) which satisfy the predicate implicit in the signature and the predicate explicit after the '|' symbol. So the set here is the set of ordered pairs of which the first element (an argument to the function) is an ordered pair of integers with the first greater than 1 and the second (the value of the function for that argument) is the element is the product of those two integers.

To see how the mapping of the lambda expression works we need to look at the definition of the semantic constant with the rather weird name ⌜$"Z'λ[2]"⌝. It has this strange name to ensure that it does not clash with names which might be used by people in their specifications. We can retrieve the definition (which is one of various classes of definitions which are produced automatically by the Z support system) from the theory database in the following way:

SML

```
|z_get_spec ⌜z⌜$"Z'λ[2]"⌝⌝;
```

```
val it =
  ⊢ ⌜∀ pack
    • ⌞z⌜$"Z′λ[2]" pack⌝⌝
        = {x
        |∃ a1 a2
          • ⌞z⌜pack a1 a2⌝.d⌝
            ∧ ⌞z⌜pack a1 a2⌝.p⌝
            ∧ ⌞z⌜pack a1 a2⌝.t⌝ = ⌞z x.1⌝
            ∧ ⌞z⌜pack a1 a2⌝.v⌝ = ⌞z x.2⌝}⌝ : THM
```

This a definition in HOL which includes fragments which are recognisably in the image of the Z to HOL mapping, and which therefore are formatted for display using the Z pretty printer. This makes the definition more readable than it would be printed purely as HOL. The constant is applied to a 'pack' which is the package containing all the semantically significant information in the constituents of the Z lambda expression. The pack is also responsible for doing the necessary variable binding and for ensuring that the scope rules for the variables are correctly implemented. As it happens, in Z variable binding constructs all the constituents are in the scope of the bindings (including the entire declaration part) so the variable binding takes place on the outside of the pack, and the pack is always a lambda expression (the only primitive variable binding structure in HOL). The body of the lambda expression must simply contain the translations into HOL of all the semantically significant constituents of the Z phrase in question (in this case a lambda expression). These have to be combined together into a single value in some convenient way, and the most convenient way to do this is by using a Z binding whose component names give some clue to what they are.

If you examine the definition above, observing carefully the language quotes, you will see that the only things quoted as Z are the selection of components from bindings. The definition tells us that the value of a Z lambda expression is a set of ordered pairs, the abstraction in the definition is a HOL set abstraction not a Z abstraction. The variable x ranges over these ordered pairs and the body tells us which ordered pairs are in the set. Note that the x here is not the same as the variable x in the Z expression, it is a bound variable in the definition of the semantic constant. The variables bound in the lambda abstraction are the ones over which the pack is an abstraction, and the values of these variables in the case in question are those supplied to the pack in this definition as 'a1' and 'a2'. The defining property of this set is that when instantiated to these particular values of the bound variables:

- the predicate implicit in the declaration of the lambda expression is true
- the explicit predicate is true
- the 'tuple' expression is equal to the left element of the pair
- the body of the lambda expression is equal to the second element of the pair.

5 Some Applications

5.1 Security

The first application of ProofPower was in a project carried out for the Defence Research Agency (DRA) in 1993 and 1994. A group led by Simon Wisemen at DRA were undertaking a programme of research in secure systems and had specified for a secure relational database system and its query language. ICL undertook to produce a formal model of the system and its critical security properties and to verify that the model satisfied those properties.

The result amounts to the verification that a programming language (the database query language) satisfies certain information flow constraints. These constraints are expressed as properties on behaviours on an abstract execution machine. Omitting the 1,000 or so lines of specification that give it meaning, the actual theorem proved is:

$$\vdash \mathit{behaviours}\ \mathit{SSQLam} \in \mathit{secure}$$

The proof scripts (which can still be replayed over 10 years later) run to about 14,000 lines and took about 2 person/years of effort to construct. The work was carried out in the ProofPower-HOL language using the Z-like library to make the specifications as accessible as possible to a readership with a good knowledge of Z but little knowledge of HOL.

5.2 Code Analysis

[Note: This will be a very brief precis of my paper "Analysis of Compiled Code".]

[Note: I think you should mention some of your mathematical stuff.]

6 DAZ

The initial development of ProofPower was motivated by the apparent demand for formal verification against specifications in Z at the extremes of high assurance in secure computing, and by the need for tools to support that process. The development supplied tools which enabled ICL to complete the design and verification of secure systems, and also gave skills to the formal methods team in the development of proof tools in a context in which external contracts for such development were in prospect. By the time that the FST project was complete, the context had changed. The first setback had been that expected open tenders for development of proof tools for use in secure systems development failed to materialise. The developments did take place, but were undertaken under existing formal methods consultancy contracts. The more serious setback was a complete volte-face in government policy on the development of secure systems. The dominant trend in military computing procurement moved towards "COTS", Commercial Off The Shelf procurements, rather than the more traditional and more expensive development of bespoke systems. For this or other reasons the expected stream of government contracts for the development of highly secure systems dried up.

At the same time as the prospects for formal methods in secure computing were faltering, the application of formal methods to safety critical military systems was being underpinned by Defence Standard 00-55.

The Royal Signals and Radar Establishment at Malvern had pioneered research for the Ministry of Defence in formal methods, and (inter alia) had developed a "compliance notation" which permitted refinement of specifications in Z into programs in a safe subset of Ada. As the FST project came to an end, RSRE, by then part of the Defence Research Agency, put out an open tender for a tool to support the use of their compliance notation in the development of safety critical systems. This would open an alternative marketplace to ProofPower and the ICL High Assurance Team if the contract could be secured and the compliance tool built on ProofPower.

[**Note: Need to say more. Add reference to Chris Sennett's paper and Colin et al.'s subsequent papers.**]

7 ClawZ

8 The Future

In 1997, development and exploitation of ProofPower was taken over by Lemma 1 Ltd. ProofPower is now made available by Lemma 1 Ltd. as

open source software suite which provides support for the application of proof oriented formal methods to the development of information systems. The software and supporting documentation may be downloaded from the `lemma-one.com` web site.

References

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

[2] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–, 1940.

[3] Michael J.C. Gordon. HOL:a proof generating system for higher-order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer, 1987.

[4] Michael J.C. Gordon. Mechanising programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Proceedings of the 1988 Banff Conference on Hardware Verification*. Springer-Verlag, 1988.

[5] Michael J.C. Gordon and Tom F. Melham, editors. *Introduction to HOL*. Cambridge University Press, 1993.

[6] Michael J.C. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF. Lecture Notes in Computer Science. Vol. 78*. Springer-Verlag, 1979.

[7] Z Standards Change Group. Z base standard (version 0.4), 9th December 1991.

[8] L.Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987. Cambridge Tracts in Theoretical Computer Science 2.

[9] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[10] R.Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[11] B. Russell. Mathematical logic as based on the theory of types. *American Journal of Mathematics*, 30:222–262, 1908.

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

[13] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 1989.