Z in HOL - the story of ProofPower

Roger Bishop Jones

Date: 2012/06/03 21:32:27

Abstract

An analysis of the ideas behind the engineering of a proof tool to support the Z specification language by semantic embedding into HOL. From the ideas of Leibniz via the creation of the new academic disciplines, first of Mathematical Logic and then of Computer Science, we trace the roots of one small step in the mechanisation of reason.

http://www.rbjones.com/rbjpub/pp/doc/t013.pdf

Id: t013.doc,v 1.5 2012/06/03 21:32:27 rbj Exp

Contents

1 Introduction **2**
- 1.1 Gottfried Wilhelm Leibniz . . . 2
- 1.2 The Foundations of Mathematics . . . 3
- 1.3 Mathematical Logic . . . 4

2 HOL and its Roots **4**
- 2.1 Russell's Theory of Types . . . 4
- 2.2 The Simple Theory of Types . . . 5
- 2.3 LCF . . . 6
- 2.4 HOL . . . 7

3 Z and its Roots **7**
- 3.1 Zermelo Set Theory . . . 7
- 3.2 The Z specification language . . . 8
  - 3.2.1 Z Without Types . . . 9
  - 3.2.2 Types in Z . . . 10
  - 3.2.3 Schema Types . . . 10
  - 3.2.4 Undefinedness in Z . . . 12

4 Interpreting Z in HOL **14**
- 4.1 What Kind of Embedding? . . . 14
- 4.2 The Type Injection . . . 16
- 4.3 Mapping Formulae and Terms . . . 16
- 4.4 Undefinedness in Z in HOL . . . 18

5 Implementation **19**
5.1 Using Cambridge HOL for Proofs in Z 19
5.2 The FST project 21
5.3 Some Preliminary Decisions 22
5.4 The Implementation of ProofPower HOL 22
5.5 Implementing Z in HOL 22

6 DAZ and CLAWZ **22**

1 Introduction

In 1990 International Computers Limited began 3 year collaborative research and development project, in which its effort was to be devoted to the engineering of a proof assistant for the Z specification language, intended for applications in the development of highly assured secure computing systems.

This paper is a kind of ideological reverse engineering of the software development which ensued, and a cameo showing how ideas formulated before their time can reach across centuries of cultural evolution to inspire effects which could not have been foreseen. It traces back through hundreds of years and several academic disciplines the history of the ideas which shaped the product *ProofPower* which emerged from that project.

The character of the paper has been influenced by the work of Donald MacKensie in his book *Mechanising Proof - Computing, Risk and Trust* [**?**], which provides a broader perspective on the context in which this work took place.

The known history of these ideas goes back of course through millenia before the trail goes cold, and the choice of where to begin is therefore to some extent arbitrary.

1.1 Gottfried Wilhelm Leibniz

The story begins with Leibniz(1646-1716), who I think of as the patron saint of automated of reasoning. Leibniz conceived of a *lingua characteristica* and a *calculus ratiocinator*. The first was a universal formal language, in which all things could be expressed with mathematical precision. The second was a calculus which permitted any question formulated in this language to be reliably answered by mechanical computation.

Though Leibniz also contributed to the mechanisation of such computations through his work on calculating machines, digital computers as we know them were hundreds of years into the future. Not only were Leibniz's aspirations ahead of the technology of his day, they depended on advances in symbolic logic which were not to be realised for another quarter of a millenium.

Leibniz was in other respects of his time. Galileo's new scientific methods, involving the use of mathematics in formulating precise models of aspects of physical reality, ushered in a new era for science and stimulated new developments in mathematics to support this new kind of science. A fundamental building block for the kind of applicable mathematics which was needed by the new science and engineering was what we now call *the calculus*, on whose development many mathematicians of the time were engaged. Though many other mathematicians contributed, Newton and Leibniz are generally credited with independently fitting the story together (though the question of precedence was a source of bitter controversty between them).

The invention of the differential and integral calculus was however just the starting point for a period in which the new kinds of mathematics opened up by the calculus were progressed under continuing

stimulus from the development of science. In periods such as this it is practical achievement in the establishment of applicable mathematical methods which drive the development of the subject. Rigour takes a back seat, uncertainties are put aside for so long as new mathematical developments prove to be effectively applicable.

Doubts there were, at the very beginning of this process, notably a corrosive critique by the Irish philosopher Berkeley. The difficulties appear most conspicuously for Leibniz's approach to the calculus in his use of infinitesimal quantities. As a result of developments in the 20th century it is now known how infinitesimals can coherently be used in the development of analysis (this yields what is now called "non-standard analysis", "analysis" being the part of mathematics in which the differential and integral calculi now belong). But when the calculus was first introduced, and for a couple of centuries during which there was continuous rapid growth of mathematics based on the calculus, neither the real numbers nor the non-standard reals were adequately understood to make a rigorous foundation for this mathematics possible.

Only when we come to the nineteenth century does the rush to exploit the calculus calm sufficiently for mathematicians to pause and consider more carefully its foundations. Once mathematicians began to look toward the foundations of analysis, a new trend begins which leads in due course to the new discipline of mathematical logic, and to the advances in logic necessary for any real progress towards Leibniz's universal language and calculus of reason.

Prominent in the results of these foundational studies we find the ancestors of Z and HOL, Zermelo's set theory, and Russell's theory of types.

1.2 The Foundations of Mathematics

The first step in the rigourisation of analysis was to find a way of doing without infinitesimals (though it is now known how one can rigorously do *with* them, this was not to be discovered until much later, and non-standard analysis has not supplanted the "standard" methods whose development now concerns us).

The concept of "limit" was the key to defining key concepts such as "continuous", "differential" and "integral". Once these concept are defined as limits there is immediately a problem about when these things exist, which places the focus on the number system in which this work is taking place. The next step is therefore to clarify the concept of "real number", giving a conception of number in which limits exists when it seems natural and convenient that they should. It is not needed that all series of numbers should have a limit, the kinds of series which one would like to have limits are the *convergent* series.

To realise a number system in which convergent series have limits, numbers may be thought of as "cuts" in the rational numbers. A cut divides the rationals into two non-empty collection, a lower and an upper, such that every rational is in just one of these collections and every rational in the lower collection is smaller than every rational in the upper collection, and such that the upper collection has no least member. The limit of a series is then the set of rationals which are less that at most finitely elements in the series and it can be shown that every convergent series of rationals has a limit in this number system, and that every convergent series of these new *real* numbers has a limit which is a real number.

We now see that in our search for rigour we have found it desirable to speak of sets. The real numbers are defined as sets of rationals, the rationals are readily understood as constructions from the whole numbers of arithmetic, and for a mathematician acquainted with the ideas of Leibniz it is natural to ask whether one further step reducing the whole to a formal logic might be possible.

1.3 Mathematical Logic

Another thread in the mathematics of this period, alongside the rigourisation of analysis consists in attempts to treat logic mathematically. In Leibniz's conception of a calculus ratiocinator the computations involved were numerical computations, his lingua characteristica involving numerical coding of concepts and propositions. In the nineteenth century mathematicians found new more abstract ways of studying number systems, the methods of abstract algebra. From the perspective of abstract algebra the truth functional logic of propositional connectives *and*, *or* and *not* could be seen as an example of an algebra, called a *boolean* algebra after the mathematician George Boole.

This approach to a mathematical treatment of logic was limited. It did not connect well with the conception of logic as concerned with formal deductive systems.

2 HOL and its Roots

HOL is an acronym for Higher Order Logic. There are many variants of higher order logic, but in this essay the acronym HOL will be used specifically for the version of higher order logic implemented at Cambridge University as a proof assistant, also called HOL.

HOL is a direct descendant of Bertrand Russell's *Theory of Types*, details of which were first published in 1908, and the line of descent is simple enough for a reasonable account to fit into these pages.

In brief the story begins with Russell's original *ramified* theory of types[4], which is then simplified by removal of the ramifications to give the *simple theory of types*. The next step, due to Alonzo Church[2] is a further simplification obtained by basing the system on functional abstraction. This ends the line of descent belonging to mathematical logic leaving only those further elaborations which were thought necessary for application of the logic to hardware verification.

2.1 Russell's Theory of Types

Russell devised his theory as a logic in which the reduction of mathematics to logic could be carried through. The main problem which Russell had to solve in devising his logic was to avoid in a philosophically satisfactory way the antinomies which had already been noted in the foundations of mathematics, including but not limited to the one which now bears his name. On examining all the known foundational paradoxes Russell observed that they had all in common "the assumption of a totality such that, if it were legitimate it would at once be enlarged by new members defined in terms of itself", and proposed the rule that "no totality can contain members defined in terms of itself". This is Russell's *vicious circle principle* and is the basis of his type theory, in which context it assumes the more technical form "whatever contains an apparent variable cannot be a possible value of that variable".

In the resulting theory the universe of discourse is considered to be partitioned into a countably infinite sequence of *types*. The first type consists of individuals and the rest of propositional functions. A propositional function should be thought of as a proposition containing real (i.e. free) variables, and which therefore may be considered as a function which yields a proposition for each of the possible values which these free variables might take. A propositional function may also be expressed using apparent variables (variables bound by quantifiers). In the ramified type system the type of such a propositional function is the least type greater than that of any variable (real or apparent) which occurs in it.

In this *predicative* type theory only limited parts of mathematics can be derived. Russell therefore found it expedient to adopt his *axiom of reducibility*, which states that every propositional function

is co-extensive with a *predicative* function. A predicative function is one in which the type of every apparent variable is no greater than the type of a real variable, so the effect of the axiom of reducibility is to negate the effect of the "ramification" (i.e. of taking into effect the types of apparent variables).

2.2 The Simple Theory of Types

It was inevitable that Russell's theory of types would be simplified. This was first proposed by Ramsey [3] after Peano had observed that the ramifications were relevant only to the resolution of the semantic paradoxes (such as *the liar*), which, even without the ramification to the type system, are not reproducible in this kind of logical system.

After a bit of cleaning up this becomes the standard textbook presentation of higher order logic. But alongside the establishment of this logical system investigations into founding logic more radically and thoroughly on functions lead to a reformulation of the simple theory of types both more elegant and better suited to the applications in computer science which concern us here. This formulation of the simple theory of types is due to Alonzo Church [2], and is simple enough to present here.

In Church's paper the system is described using concrete syntax considering syntactic entities as strings. The system presented here is the same logical system, but is presented instead using *abstract syntax* an innovation in the description of languages which arose in Computer Science.

They type system of Church's formulation of the simple theory of types consists of the two types i and o which should be thought of respectively as the types of *individuals* and of *propositions*. It has a single binary type constructor which given two types α and β yields the type $(\beta\alpha)$ which is a type of (total) functions whose domain is the type α and whose co domain is the type β. The abstract syntax of the type system is the single sorted free algebra with this signature.

The terms of the language are either *variables*, *constants*, *lambda abstractions* or *function applications*.

A variable is a name together with a type, as is a constant, and is written as the name subcripted with the type. Lambda abstraction is a ternary term constructor, whose operands are a name n, a type *ty* and a term *tm*, and is written as the letter λ followed by the *bound variable* whose name is n and whose type is *ty*, followed by the term *tm*. Function application consists of the juxtaposition of a term whose type is compound on the left with another term on the right, subject to the constraint that the type of the first term is a function type of which the second argument is the type of the second argument. Application of this constraint of course depends upon the assignment of types to arbitrary terms which is done as follows.

The type of a variable or a constant is the type from which it was formed. The type of an abstraction is the function type of which the first argument is the type of the term which is the body of the abstraction, and the second is the type of the bound variable in the abstraction. They type of a function application which complies with the above mentioned constraint is the type of the co-domain of the first term of the application.

The logical system is a Hilbert style deductive system in which the role of formulae is undertaken by terms of type o.

The system has two primitive constants and two families of primitive constants which are:

- The constant whose name is 'N' and whose type is '(oo)'.
- The constant whose name is '$\vee$' and whose type is '$((oo)o)$'.
- The constants whose names are '$\forall$' and whose types are '$(((o\alpha)\alpha))$' for each type α.
- The constants whose names are 'ι' and whose types are '$(\alpha(o\alpha))$' for each type α.

These constants should be thought of respectively as negation, disjunction, universal quantification and description.

The axioms and inference rules of the system also involve various additional constants which may be defined in terms of the above primitive constants. The constants defined by Church are conjunction, implication, equivalence, existential quantification, equality, inequality, the identity function, the combinator K, the numerals (defined as function iterators) the successor function, the property of being a natural number,

The deductive system consists of six rules and eleven axioms or axiom schemata.

The first three rules are rules of lambda conversion, viz. renaming of bound variables, lambda-reduction and its inverse. The fourth rule is a limited rule of substitution, the fifth modus ponens and the last is the rule of generalisation permitting the introduction of universal quantifiers.

Of the eleven axiom schemata the first four provide for the propositional calculus, the fifth allows specialisation of universally quantified formulae, and the sixth allows universal quantifiers to be pushed in over disjunctions (subject to appropriate conditions). Axioms seven and eight provide for arithmetic, the first asserting that there is more than one individual, and the second that no two distinct numbers have the same successor. Schema nine provides for descriptions by asserting that the description operator, when applied to a propositional function which is true of exactly one value, yields that value. Schema ten asserts that functions are extensional, and eleven is an axiom of choice which upgrades the description functions to choice functions.

2.3 LCF

The details of the HOL logic and its implementation as the HOL proof assistant arise from adapting a system originally devised to support a quite different logic to provide support for an elaboration of Church's system.

The original logic and the software system which supported its use were both called LCF. LCF stands for *Logic for Computable Functions*. The logic was devised by the logician Dana Scott for reasoning about denotation semantics. It is a first order logic in which the terms are the terms of a typed lambda calculus, construed as functions in "Scott Domains" (continuous partial orders), supported by reasoning using fixed point induction. In the logic as formulated by Scott, the type system for the terms is the same as that used by Church in his Simple Theory of Types, except that the primitive types differ. In Scott's LCF formulae were distinct from terms and there was therefore no need for a type of propositions. Scott was at pains to point out that the variables in his presentation ranging over types were variables in the metalanguage, not variables in the object language LCF.

The first tool providing support for this logic was *Stanford LCF*, implemented at Stanford University as a tool which would assist users in constructing and checking proofs in the LCF. The team developing the Standford LCF tool was lead by Robin Milner, who then moved to Edinburgh and with the benefit of experience in applying that tool came up with some radical new ideas for a successor which would be know as Edinburgh LCF.

One of those ideas was to develop a new *typed* functional programming language for use both in implementing the proof tool and as a metalanguage for interacting with the users of the tool when constructing and checking proofs. Stanford LCF had been implemented in LISP, which is an untyped functional programming language. It was immediately clear that if a typed functional language was to be substituted for LISP, the type system would have to be polymorphic. (Otherwise, for example, all functions over lists would have to be coded up many times, once for every different type of list for which they were needed).

The closeness between such a typed functional programming language and the language of terms

in LCF made it obvious to consider whether, if polymorphism was required in the metalanguage, it should not also be permitted in the object language. Why was Scott at pains to point out that his type variables were in the metalanguage, were there doubts about the consistency of the logic if they were admitted in the object language, or was just a question he did not want to consider?

Anyway, the upshot was that polymorphism, in the form of free type variables which could not be bound but which could be instantiated, was included not only in the new Metalanguage, ML, but also in the version of LCF which was supported by Edinburgh LCF. When later the LCF system was modified to support HOL, polymorphism in the object language was naturally retained.

The effects of LCF on HOL were not confined to the introduction of type variables. It was natural in switching from LCF to HOL to make minimal changes to the code of the LCF system, and hence to the LCF deductive system, which had been developed with efficiency in mind at a time when computers where several orders of magnitude slower, and a great deal less common, than they now are. The LCF system es further developed at Cambridge University, and it is the Cambridge version of LCF which was adapted to make the HOL system. The version of LCF supported by Cambridge LCF was called PPLAMBDA and was documented by Larry Paulson in technical report TR-36.

The key features of PPLAMBDA were:

substitution

2.4 HOL

3 Z and its Roots

3.1 Zermelo Set Theory

In the same year that Russell published his *Theory of Types*, Ernst Zermelo published a paper[7] in which he offered a resolution to the set theoretic paradoxes through an axiomatisation of the theory of sets.

This paper is less ambitious than Russell's. It seeks to present a consistent basis for the further development of set theory but neither attempts to formalise nor to provide a philosophical justification for the chosen axiom system. The introduction to the paper makes clear that Zermelo has very specifically aimed to produce a set of axioms which are sufficient for the derivation of set theory as it was then understood, but which is immune to the paradoxes. It is this paper which provided the foundation for twentieth century mathematics, needing relatively modest changes to make it suitable for most modern mathematics.

Subsequent philosophical rationales for Zermelo's axiom system have revolved either around the principle of limitation of size, or on the interactive conception of set. However, the relationship between this system and the type theories we have discussed is perhaps more clearly seen in terms of the principle of well-foundedness.

Russell's saw the paradoxes as arising from circularity in definitions. However, if definitions are conceived as picking out individuals from an already well established domain of discourse more liberal principles of definition than those proposed by Russell may be admitted. The can be accomplished by moving the constraint against circularity from the definitions to the membership relation. This is accomplished if the membership relation is required to be well-founded. Zermelo does not himself place this constraint on sets, though it will later be added. However, his theories are consistent with the constraint that sets be well-founded and the concept of well-founded set can be made intuitively reasonably clear through the idea of the iterative or cumulative hierarchy, in which sets are conceived

as ordered by a rank which corresponds to the stage at which it appears in an iterative process of set formation which may be considered to populate the domain of discourse.

Zermelo's axioms were as follows:

I Extensionality - two sets are the same if they have the same members

II That an empty set exists, and, for each set its unit set, and for any two sets a set which has just those sets as members.

III Separation - for each set and for each "definite property" of sets, there exists a set which contains just those elements of the first set which satisfy the property.

IV Power set - for each set there exists a power set whose members are the subsets of the set.

V Union - the union of the members of a set is a set.

VI Choice - for each set of non-empty sets whose members are pairwise disjoint there exists a choice set containing exactly one element from each of the members of the set.

VII Infinity - there is a set which contains the empty set and contains the unit set of each of its members.

This set of axioms appears to be consistent with an interpretation whose elements are the well-founded sets, which may be thought to be justified by the iterative or cumulative conception of set. It appears immune to the known paradoxes, while being sufficient for most of the set theory known at the time. This it achieves without the complexity and inconvenience of a type system.

The principle weaknesses of the system are:

1. Because this is an *informal* axiom system it is not easy to make precise the notion of *definite property* which appears in the axiom of separation.
2. It lacks sufficiently strong existence principles for some kinds of mathematics (but is sufficient for arithmetic and analysis).

These weaknesses were eventually remedied, yielding the first order set theory now known as ZFC, which was widely regarded over the main part of the last century as providing a safe context in which mathematics could be undertaken. It is the formalisation of the axiom system in first order logic which permitted the notion of *definite property* to be made definite. The axiom schema of replacement was added permitting many more collections to be shown to exist. Of these, for applications in formal methods for computer science the former was essential but the latter was less crucial since the branches of mathematics depending on the axiom of replacement are much less likely to be required for applications in computing.

3.2 The Z specification language

The Z specification language is in superficial respects much richer than the language of first order set theory. The richness is what computer scientists call "syntactic sugar", which can be unravelled to render the same message in the simpler language of first order set theory.

In this section I will attempt to describe the key features of Z by comparison both with Zermelo set theory and also with higher order logic, informally but in sufficient detail to permit subsequent description of the interpretation of Z in HOL.

To understand Z and its interpretation in HOL it is best first to understand the kinds of objects which populate the universe of discourse of Z and then to know what can be done with them.

Unlike Zermelo set theory, which is pure untyped first order theory, Z has a type system, and is in this respect more like a type theory or a higher order logic. This facilitates the interpretation of Z in HOL, and permits a first relationship to be established by consideration of the type systems of the two languages.

However, the type system of Z operates *behind the scenes*, and a first understanding of Z is perhaps better realised through an untyped interpretation.

3.2.1 Z Without Types

A Z specification consists of a sequence of paragraphs which serve to define various "global variables" ("global variable" is Z-speak for "constant", the difference between them and local variables is not just that they have global scope, if they really were global variables in a first order logic then one could generalise assertions involving them, which is definitely not intended in Z).

At this stage I will describe just three of these kinds of paragraph, the given set paragraph and the axiomatic specification paragraph.

A given set paragraph consists in a list of given set names. These are the names of sets whose interpretation is left loose, i.e. they are new "global variables" about which nothing is known except that they denote sets.

An axiomatic specification introduces one or more new global variables, which are specified in two ways. Firstly they are specified in a signature which supplies for each of these variables a set of which it is required to be a member. This signature looks like a type constraint, insofar as the membership of the new name in the set is written using a colon sign, but the expression on the right can be an arbitrary non-empty set. In addition to the signature a formula is supplied in which the new global variables may appear as free variables. This formula is called the *predicate* of the specification.

The effect of this specification paragraph is to introduce a new axiom which asserts the conjunction of the constraints imposed by the signature and the predicate.

The third kind of paragraph is the schema box, which introduces a new global variable which denotes a set of bindings. A binding is like an ordered tuple except that the components are named rather than merely ordered. A schema box, like the axiomatic specification, contains a signature and a predicate. Instead of introducing new global variable with the names in the signature, it introduces a single global variable whose name is declared on the schema box, and whose value a the set of bindings whose components are the component names in the signature. The set is the set of those bindings which satisfy the conjunction of the signature and the predicate. The schema box is therefore a kind of set comprehension, in which the signature of the schema serves to identify a set from which the required set can be obtained by separation.

The use of signatures which appear similar to type constraints but in fact constrain names to fall within arbitrary sets is pervasive in Z, and appears not only in the paragraphs which provide top level definitions of global variables, but also in all the variable binding constructs which are permitted in the formulae or terms of the language. The effect is that all the abstractions which are legal in Z are sufficiently constrained that they can be interpreted as separation in an untyped first order set theory such as Zermelo set theory.

Though signatures are in fact set constraints rather than type constraints, and almost all [1] the

[1] The exceptions are the schema operations which involve negation, viz. schema negation itself, schema implication and equivalence, which, because they take a complement would not be interpretable if they were not constrained by a

language can be interpreted in an untyped set theory, Z is in fact a typed set theory.

3.2.2 Types in Z

The type system of Z is more elaborate than that of STT. Users of the language are encouraged to make use of uninterpreted types of entities which Z calls *given sets*. The types of Z are the given sets introduced by a specification, the type Z of integers, and types which can be formed from other types by applications of the following type constructors:

Powerset - the type whose members are sets of objects of some type.

Tuple - for each tuple of n types there is a type of n-tuples in which the nth element has the nth type

Schema - for each signature (identifier indexed finite family of types) a schema or labelled product type is available whose members are *bindings* with the given signature. A binding is like a tuple but the elements are identified by tags or component names rather than a numeric position in the tuple.

All these kinds of construction can be undertaken in Zermelo set theory, provided that the given sets are all pure sets, using a variety of coding tricks, but in Z these coding tricks are hidden. For the Z user every type is a set, but the elements of that set will not in general be known to be sets (they are known to be sets only for the powerset construction).

In these type constructors there is some redundancy, since n-tuples are considered identical with schemas with corresponding numeric component labels. The types are therefore the smallest set which can be constructed from the given sets by application of power set and schema type constructions.

One further elaboration is the type genericism in Z. The paragraphs constituting a Z specification may be parametrised by sets, and the objects defined can then be instantiated by supplying actual sets in place of these parameters. The type of the Z entities defined must then reflect this set genericism. A generic type is then a finite sequence of parameter names together with a monotype in which these names may appear as if they were given sets.

The most important of these types in shaping the character of the Z language is the schema type, whose role I will therefore discuss in greater detail in the next section.

3.2.3 Schema Types

An early stage in the development of set theory for use in pure mathematics is the selection of a method for representing ordered pairs (of sets) as sets. This is normally done using a method devised by Sierpinski, though other methods are available. Once order pairs are available the construction can be iterated to give arbitrary n-tuples. Ordered pairs are used in the representation of functions as sets, and thence sequences and a wide variety of complex mathematical (or data) structures.

When mathematics is used for modelling computer systems the use of tuples is pervasive and the problem of remembering the significance of each element in these tuple becomes an impediment to ease in understanding the mathematics. In programming languages (particularly and originally in such commercial programming languages as COBOL) this problem of intelligibility is alleviated by the use of tuples with named components (sometimes called structures or records).

The most important single innovation of the Z specification language relative to first order set theory is the introduction of labelled tuples, which are called in Z *bindings*. Set of these binding, which

type system (on pain of letting in Russell's paradox).

are called *schemas* are ubiquitous in Z specification and give the language its special character. The term *schema* is used both for the type constructor which gives a type of bindings, and for an object whose type is the powerset of a schema type (i.e. for subsets of schema types), as well as for a special kind syntactic object (a box containing a signature and a formula) which denotes a set of bindings.

As well as being used simply for types or sets of data objects with named components, schemas have special roles which exploit the correspondence between sets and properties or predicates. A may be defined in Zermelo set theory using separation of the elements satisfying some definite property from some already available set. Once a type system is imposed on set theory, sets of any type of value can be defined taking the type as the set from which separation takes place, and a bijection is established between typed propositional functions or predicates and typed sets. As a result of this correspondence a set of binding can be taken to represent a property, the property of being a member of that set.

This also provides a convenient way of defining a schema. The schema may be defined by providing:

- a *signature* which identifies the names of the components of the bindings in the schema together with the a set for each component from which the values are drawn (and which determines the type of the values of the component)
- a formula, called in Z the *predicate* in which the names of the components may appear as free variables, and which is required to be true for every binding in the schema when the components of the binding are assigned to the appropriately named free variables in this formula. This is analogous to the ability in some programming languages to use structured objects which can be "opened" in some context, effectively assigning the value of each component to a local variable of that name and making it possible to access these values without reference (in the relevant context) to the structure from which they are drawn.

As well as using formulae with free variables as a way of defining a schema having bindings with a matching signature, the opposite effect is possible. A name whose value is a schema can be used in a formula as a shorthand for the formula defining the schema. When this is done, something logically strange happens, a formula contains for logical purposes occurrences of free variables which do not explicitly occur in the formula. This is a very special feature of Z, making the kinds of specification written in Z much more concise than they might otherwise be. It does however present some special difficulties from the point of view of embedding the language into some more conventional language such as HOL, and for defining or implementing a sound deductive system for the language.

To explain (as we will do shortly) how this exotic use of covert or implicit variables can be properly interpreted in HOL it is convenient to describe how its use in Z can be eliminated to yield a Z specification in which all occurrences of variables are explicit. To do this it is necessary to have a notation for an explicit binding display, i.e. a notation in which the desired values of the components of some binding can be combined to yield a binding having those components. Such a notation was not available in Z at the time of the development we are describing. The notation we use here was added to the Z implemented in ProofPower, a different notation was eventually added to Z during the ISO standardisation process.

For our discussion a binding display will be presented as a bracketed sequence of comma separated pairs, each pair consisting of the name of a component of the binding and the value which that component is to take, with the $\widehat{=}$ symbol appearing between the two.

Thus a binding with two components one of which is called 'one' and has the value 1 and the other 'two' having the value 2 is written:

$$\left|(one \mathrel{\widehat{=}} 1,\ two \mathrel{\widehat{=}} 2)\right.$$

It should be understood that the names in the above schema display are not variables.

There are two notations in Z which introduce covert variables and which we therefore wish to explain in terms of equivalent Z with explicit variables.

The first is the "theta term". A theta term is the name of a schema (i.e. a variable whose type is the power set of a schema type) immediately preceded with the Greek letter theta, e.g. θTWO. If TWO is a name which has been given to the type of bindings to which the above schema display belongs, then this can be understood as an abbreviation for the following binding display:

$$(one \mathrel{\widehat=} two,\ two \mathrel{\widehat=} two)$$

In this expression the occurrences of *one* and *two* to the right of the definition symbols are variables (free in this context). Thus the expression θTWO must be considered as containing two hidden or implicit occurrences of variables, and its value will be determined by the values of these variables in the context in which the theta term appears.

The second kind of notation in which variables may be implicit is the "schema reference". A schema reference is simple the use of the name of a schema as if it were a formula in which the names in the signature of the schema type occur as free variables.

This should simply be understood as an abbreviation of the formula which asserts the membership of the corresponding theta term in the schema. Thus, if the name TWO when declared as described above, is used in a context where a formula would be expected, then it will be understood as the formula $\theta TWO \in TWO$, which in turn should be construed:

$$(one \mathrel{\widehat=} two,\ two \mathrel{\widehat=} two) \in TWO$$

3.2.4 Undefinedness in Z

Z was originally construed as a first order set theory, with added types (which makes it higher order) but certain constructs in the language are naturally construed as having no value. For example a definite description which is not satisfied by exactly one value. There are ways to treat these features within the context of a first order set theory, but when the first explicit accounts of the semantics of Z were published they were inconsistent with the classic first order interpretation.

Before looking at these it will be useful to explain the natural treatment in first order logic. The key to these is the common method of introducing the axiom of choice using a choice function. In this method a new function symbol, let us write it ϵ (Hilbert's epsilon), is introduced and the axiom of choice is expressed:

$$\forall s \bullet (\exists e \bullet e \in s) \Rightarrow \epsilon\ s \in s$$

The choice function is a function over all sets which when applied to a non-empty set yields an element of the set. From the axiom it is not possible to determine which element is chosen for a set which has more than one element, or the value of the function when applied to the empty set. This means that these details of the choice function will vary across different models of the theory. But nevertheless, in every model (indeed in every interpretation, by definition) the choice function is a total function defined over all sets.

In some early formulations of Z the function μ was described as a choice function However, in Spivey's publications it is merely definite description, so that the axiom describing it is more like:

$$\forall s \bullet (\exists_1 e \bullet e \in s) \Rightarrow \epsilon\ s \in s$$

Using definite description application of a function represented by a set which is its graph to some argument can be defined, and function application, even for partial functions, becomes a total function. Function application having been rendered as a loose total function all other term constructions (such as set or function abstraction, become unproblematic and all the normal behaviour of two valued classical logic is preserved.

In Spivey's treatment definite description is give no value if applied to other than a unit set. Term constructors are *strict*, i.e. an undefined value supplied to such a construction renders the result of the construction undefined. The classical two-valued logic is then recovered by the behaviour of the primitive predicates equality and membership, which when supplied with undefined arguments are deemed false.

Though this has some aesthetic appeal it is inconvenient. It is inconvenient especially in an proof tool which makes extensive use of equational rewriting, for we lose the universal reflexivity of equality. The effects ripple through the logic and we find that many kinds of elementary reasoning the definedness of the terms involved must be proven, though in a more purely classical logic the reasoning would go through without need to establish well definedness.

The following tables illustrate the effects.

The following tables summarise the treatment of partiality as it would be if Z were viewed as a first order set theory and as it is presented in the two books by Spivey [5] and [6].

The aspects of the language which characterise its treatment of partiality:

UDP are there undefined predicates (formulae)

UDT are there undefined terms

ROFV what is the range of the free variables (does it include undefined values)?

$=$ How does equality behave with undefined operands?

$\in$ How does membership behave with undefined operands?

$\{\}$ How are undefined values treated in set abstractions?

λ How are undefined values treated in functional abstractions?

μ How are definite descriptions treated in functional abstractions?

The following additional abbreviations are also used in the table:

D free variables range over defined values only

U free variables range over defined values and 'undefined',

C 'classical'

S strict (for terms) false on undefined (for predicates)

LO loose

LI liberal (not strict)

ISSUE	*FOL*	*Blue book*	*Red book*
UDP	*No*	*No*	*No*
UDT	*No*	*Yes*	*Yes*
ROFV	*D*	*U?*	*U?*
$=$	*C*	*S*	*S*
$\in$	*C*	*S*	*S*
$\{\}$	*C*	*S*	*S*
λ	*C*	*S*	*S*
μ			

HOL Constant

...

...

4 Interpreting 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 purpose. 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*[5]. 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.

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.

4.1 What Kind of Embedding?

It is possible to approach this in several different ways, and not very easy to second guess 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 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. Each sentence in Z would then be translated into the sentence in HOL which asserts the truth of the sentence in Z.

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. In this case fragments of the semantics of Z are coded in HOL by a translation written in the metalanguage ML. For each constructor in the abstract syntax of Z a constant in HOL is defined which captures the semantics of that constructor, and phrases in Z which are made with that constructor are mapped to terms in HOL which are applications of the constant which captures its semantics. Thus, in a shallow embedding, the detail of the semantics of Z is coded into HOL constants, but the top level of the semantics where the values corresponding constructor are combined into a single semantic function for the phrase type in question are implemented in effect in the metalanguage rather than the object language. 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 alternatives:

- deep embedding
- shallow embedding into small number of types
- shallow embedding with type injection

Particular difficulties and benefits attend each of these alternatives. Among these are:

- A deep embedding permits reasoning about the embedding (i.e. about the semantics of Z) in HOL, but for that reason requires a non-conservative extension to strengthen 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. 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.
- A shallow embedding into a small number of types provides yet another combination of advantages and disadvantages, but was in fact considered at the time.

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

One advantages of a deep embedding is that it permits formalised metatheory, i.e. reasoning about the semantics of Z, whereas the shallow embedding is oriented to reasoning in Z about things specified in Z, more closely analogous to the kinds of application (e.g. in hardware verification) for which the Cambridge HOL system was developed. Use of a type injection results in a better correspondence between type correctness in the two languages (a Z specification is type correct iff its image under the mapping is type correct HOL. When working within *the LCF paradigm*, computations involving syntactic values of the object language are routine, and when a term is constructed from its constituents the typing rules are checked so that no type-incorrect terms can be constructed. With a shallow embedding of Z into HOL in this kind of context, computations involving the HOL terms which represent Z formulae or terms are automatically type checked as a result of the checks on the underlying HOL. More important than whether type checking comes for free, is the extent to which type checking may be pushed from the metalanguage into the object language as a result of a deep embedding.

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 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 the HOL function type in which the domain type is a tuple of type variables corresponding to the formal generic variables, and the range type is the image under this injection of the Z mono-type.

4.3 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. Sometimes it is desirable to put in a dummy constant which is in effect a name for the identity function so that tools processing a construct will be aware of the language in which it occurs even though the constant is the same in more than one language. This happens for example, for the universal quantifier. Though the Z universal quantifier does map to a HOL universal quantifier, the form of the body of a Z universal quantifier is special, and so it is desirable to inhibit processing the HOL universal quantifier in the normal way by wrapping it in a special constant serving only to mask it from procedures not intended for Z.

In general the pattern for the mapping involves defining a new constant or family of constants for each constructor in the abstract syntax of the Z language which correctly captures the semantics of that part of the Z language. Complications to this arise mainly in connection with the Z constructs in which variable occurrences are implicit, and the variable binding constructions of the Z language (which also allow for these implicit variables).

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. 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. 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 Z lambda expression, but since the syntactic form of the HOL lambda expression has no such semantic significance, this information must be made explicit in 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 letter used above in enumerating 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 a set of values for the 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*.

4.4 Undefinedness in Z in HOL

Z was originally supposed to be a convenient syntax for first order set theory, and its logic was considered a classical first order logic in which there are exactly two truth values and every term denotes some value from the domain of the interpretation.

There was however no systematic published account along these lines, and when the first reasonably complete accounts of the semantics of Z were published some small deviations from this position were evident. The accounts in question were, firstly that in Michael Spivey's doctoral dissertation, published as the book *understanding Z*[5], and then two editions of *The Z Notation*[6] by the same author. The first edition of the latter appeared in between the availability of the doctoral dissertation and its publication in book form.

In brief the differences between these accounts in relation to the question of undefinedness are as follow Firstly, in his doctoral dissertation Spivey broke with full conformance with a first order semantics by allowing that some terms have no denotation. The logic remains two valued, undefinedness disappearing at the primitive relations (equality and membership) which were deemed false if either of their operands failed to denote. In *The Z Notation* Spivey modifies his position, and leaves the value of the primitive predicates undetermined if either operand is undefined.

Its worth noting how this would work in a completely standard first order set theory. In such a theory every term must always denote a value in every interpretation of the language. Of a function such as function application is included in the language, then an application of a function to a value which is not in its domain must yield in every interpretation some value in the domain of the interpretation. One way to fully define such a function would be to say that the value of a function when applied to an argument for which it is not single-valued is the value of the function itself (this has the advantage that because of the well-foundedness of Zermelo set theory this could not possibly be the value of the function at any point for which it is single valued). However, it is not desirable that an arbitrary choice of this kind should be made definite so that it can be exploited in specifications. It is preferable to leave this aspect of the application function undetermined, saying that it is indeed a function, and does indeed yield a value in every case, but declining to say what that value is in the cases where the function is not single valued (or has no value).

This looks like a fine case of hair splitting, so I had better make haste to explain why this is in fact of practical significance.

When we come to implement a proof tool for this language, especially as in our case, based on HOL which makes extensive use of rewriting with equations, the unadulterated first order logic is significant. It is significant because it guarantees with absolutely no exceptions that for every term "t", "t = t" is true and is a theorem. If we insist that every term denotes, then universal quantification embraces all terms, otherwise we find that when a universally quantified result is instantiated it is necessary to prove that the term to which it is instantiated denotes some value. There are very many logical rules which can be applied without consideration of whether the terms under consideration are well defined, *provided that* the semantics holds fast to first order logic, but which cannot be applied without first demonstrating that the terms involved denote if non denoting terms are admitted.

5 Implementation

5.1 Using Cambridge HOL for Proofs in Z

International Computers Limited was a company put together by the British government in an attempt to rescue a declining UK computer industry. It was formed by taking all the parts of UK industry which were involved in the design and manufacture of commercial computers and making a single manufacturer. In the process of putting the company together a deal was struck between the various companies involved, that they would not compete with ICL in the commercial sector for a certain period of time (must have been about 20 years) and in return ICL agreed not to go for the non-commercial sector (mainly military computing).

As this period drew to a close ICL geared up to enter its new markets with some enthusiasm. It had in its portfolio some products, such as its distributed array processor (DAP), which had clear military applications. A Defence Technology Centre was established to undertake the research and development which was though essential to success in defence.

At this time the UK government was engaged in the promotion of capability in UK industry for the development of secure computer systems by methods similar to those advocated and under development in the USA. To this end the government were placing contracts with UK contractors to undertake research and development in the area of highly secure computer systems. The application of formal methods was thought to be essential, the aim being to specify formally the critical security properties of such systems, to formally specify the relevant aspects of the design of such systems, and to mathematically prove that a system implemented according to the design would meet the security requirements. Accordingly ICL established in its Defence Technology Centre a Formal Methods Unit intended to provide the necessary technical skills for developing systems in the approved manner.

It was generally supposed that the application of mathematics to the verification of computer systems would require machine checked formal proofs, rather than the more conventional less formal proofs found in mathematical journal and checked by peer review. Consequently, computer based proof assistants were considered essential, and as early as possible the formal methods team sought practical experience in the application of such tools in their problem domain. At this time the government had already chosen to promote the use of a single formal specification language to be used for this kind of security work. Having taken advice from its consultants they had selected for this purpose the Z specification language.

No proof tool was then available which supported Z, so ICL sought a tool with which experience could be gained by manually translating specifications from Z into a language supported by the tool. Time was of the essence, and not much of it was available for surveying proof tool. The choice was made between NQTHM, otherwise known as "the Boyer-Moore theorem prover" after its principle authors, and HOL. NQTHM was thought to have a more powerful automated proof capability, but

the language supported by this tool was an impoverished first order logic and the translation of Z specifications into this language would be likely to yield a specification quite different in form to the original. The Cambridge HOL proof assistance supported higher order logic, a language much closer in expressiveness to Z, and thought the level of proof automation was not so high as NQTHM, the implementation via *the LCF paradigm* made the system seem more adaptable and made it seem more likely that the tool could readily be adapted to use for reasoning about specifications originally written in Z.

ICL therefore began to experiment with the application of Cambridge HOL to its Z specifications of secure systems. This was done in a rather *ad hoc* way. Firstly some facilities were coded using the metalanguage ML to facilitate use of the kinds of information structures which were common in Z. The most pervasive of these is the schema type, which is a kind of labelled product, which could be introduced in HOL by introducing a new type represented by an iterated ordered pair in relation to which appropriate constructors and projections were automatically introduced. The definition of schemas in Z is then readily translated into the definition of properties over these labelled product types. A second important kind of construction to be translated was the Z free type, which in its non-recursive applications is a labelled disjoint union and could be supported in similar ways. There was at this time no support for recursive datatypes in HOL.

The other major feature of the Z language, at the paragraph level, for which support was needed, was the axiomatic specification. An axiomatic specification consists of the introduction of one or more new "global variables" (which translate into HOL as constants) together with an axiom jointly constraining these new global variables. There is here a small clash of cultures. The culture around Cambridge HOL reflected its ancestry as a foundation system for mathematics, in which mathematics was expected to be undertaken by the use of definitions only, not new axioms. The more liberal attitude of the Z community to the use of axioms seemed less appropriate for the intended applications, in which a great deal of weight was to be placed on formal reasoning about the system. In order to guarantee that the consistency of the logical system could not be compromised by errors in the specification, ICL sought to translate the axiomatic specifications by conservative means. This was done by translating them using the choice function in HOL. A Z axiomatic specification was translated into a definition in which a new constant was defined as *the* value which is a tuple whose components satisfy the required predicate.

Use of the choice function in this way was safe, so far as consistency was concerned but was not an entirely faithful translation of the Z, in two respects, one desirable and the other regrettable. The desirable lack of faithfulness concerned the translation of non-conservative axiomatic specifications. If the consistency of the predicate is not provable in HOL, then the desired properties of the newly defined constants cannot be established. This is what makes it safe. The undesirable feature is that some identities are provable which should not be. This occurs when two different global variables are introduced using the same loose (not uniquely satisfied) property. For example in two separate paragraphs, two distinct global variables a and b are introduced each of which is require to be an even integer but is otherwise unspecified.

When this is done in Z, one knows only that a and b are both even integers. If these loose specifications are translated into HOL using the choice function, it becomes possible to prove that the two constants have the same value. Support for faithful translation of such specifications was later realised by the introduction of a new facility into Cambridge HOL, called constant specification. In ProofPower this was to be the only primitive form of constant definition.

Certain other cosmetic measure were adopted to make the correspondence between a specification translated into HOL and the original Z specification easier to see. The first of these was simply limited presentation of the HOL specification as printed via LaTeXin a style similar to that of Z paragraphs, which appear in various kinds of box. The second concerns the presentation of logical symbols, which in HOL are given using ASCII characters, but in Z use non-ASCII mathematical symbols.

A new character set was devised which contained the most frequently used of the special symbols, documents were prepared using this font. The special characters were translated for printing into appropriate LaTeXmacros, they were translated when presented to HOL into the ASCII characters required by HOL. This hack, was later cleaned up to realise a compromise between a fully graphical interface and a purely ASCII interface which is now found in ProofPower.

Practical experience in reasoning about Z specifications via these experimental hacks was invaluable in securing a practical understanding of the semantics of the Z specification language and its relationship with HOL. Without this prior experience the FST project under which ProofPowerwas developed would not have been possible in the available timescales.

Throughout this process we were aware that in principle a completely faithful interpretation of Z in HOL was possible. It was clear that such a mapping could not be used in practise without more radical changes to the underlying HOL system. In general, in a systematic mapping, the resulting HOL terms would be too complex to be recognisable or intelligible, and a systematic mapping would need parsers, pretty printers and proof facilities implemented specifically to support the Z language. This could not be undertaken on the shoe-string budget under which these experiments in translating Z into HOL had been conducted.

5.2 The FST project

FST is an acronym for the project number 1563 assigned to the project by the Department of Trade and Industry which partially funded the work, chosen arbitrarily at a significant saving in creative effort.

The purpose of the project as a whole was to further the development of capability for formal machine checked verification of software and or digital hardware. The objective of ICL, which was the leader and largest contributor to the project was to completely re-engineer proof support for HOL, the language supported by the proof tool of the same name developed at Cambridge University, and to use this re-engineered tool to provide best achievable support for proof in the Z specification language by semantic embedding into HOL. The name of the software product which ICL was to produce under the project was not decided until later. My first choice was to call the tool *Principia* recognising its logical system as a direct descendant of that of *Principia Mathematica*[1], but none else seemed enthusiastic about that name. At that time ICL had a range of software products with names such as *OfficePower*, *DecisionPower* and so on, so for a new ICL product something with *Power* at the end was an easy choice, and a tool for doing proofs might as well be called *ProofPower*. This proposal met with little enthusiasm or resistance, so that was it.

The other participants in the project were the Universities of Cambridge and Kent, and Program Validation Limited.

The University of Cambridge was to undertake under the supervision of Martin Hyland, some theoretical investigations into various type theories, which ICL was to work with an established type theory, Cambridge would look into more elaborate type theories which might at some time in the future provide a superior base for this kind of undertaking.

The University of Kent, under the lead of Keith Hanna was to continue its work on the development and application of tools supporting formal specification and verification of digital hardware using classical dependent type theories, Keith Hanna's positive achievements with his VERITAS system had already been instrumental in persuading Mike Gordan to take up a higher order logic for his work on hardware verification, but no clear consensus had yet emerged on the relative merits of the simple type theory which Gordon adopted against the dependent type theories in use by Hanna.

Program Validation Limited (PVL) had extensive experience in development of software to support

static analysis and formal verification of programs written in the SPARK subset of Ada. PVL were to undertake further development of their own proof tool.

My concern in this essay is exclusively with the ICL part of the project leading to proof support for Z in HOL.

5.3 Some Preliminary Decisions

We had three years to produce an effective proof tool for the Z specification language, so we didn't really have time to debate the main features of the approach, which had already been laid out in the proposal.

5.4 The Implementation of ProofPower HOL

5.5 Implementing Z in HOL

6 DAZ and CLAWZ

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] Frank Plumpton Ramsey. *The Foundations of Mathematics and Other Essays*. Humanties Press, New York, 1931.

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

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

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

[7] Ernst. Zermelo. Investigations in the Foundations of Set Theory I. 1908.