

SOFTWARE VERIFICATION RESEARCH CENTRE
SCHOOL OF INFORMATION TECHNOLOGY
THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 97-34

Approaches to proof in Z
– or –
Why effective proof tool support for Z is
hard

Andrew Martin

November 1997

Phone: +61 7 3365 1003

Fax: +61 7 3365 1533

<http://svrc.it.uq.edu.au>

Note: Most SVRC technical reports are available via anonymous ftp, from `svrc.it.uq.edu.au` in the directory `/pub/techreports`. Abstracts and compressed postscript files are available via `http://svrc.it.uq.edu.au`

Approaches to proof in Z

— or —

Why effective proof tool support for Z is hard

Andrew Martin*

Abstract

Various attempts at supporting proof in Z are described in the literature. This paper presents a survey of these approaches, and the underlying semantic issues which make proof in Z a non-trivial task. The draft Z Standard is used as a normative reference. Special care is given to an account of the peculiarities of Z schemas. The proof tools surveyed divide into two groups: custom-made implementations for supporting Z, and encodings of a Z logic within some other logical framework. The latter are further subdivided into ‘deep’ and ‘shallow’ embeddings. The broad conclusion is that none of these approaches is a clear winner at present, but that each may be able to benefit from the others.

Keywords

formal proof, semantics, proof tools, Z notation, schemas

1 Introduction

In the software engineering world, Z (Nicholls, 1995) has gained some significant respect as a notation for specification (see for example, (Bowen et al., 1997)), but *very few* projects undertake reasoning about their specifications—either to validate them, or to refine them formally towards code. The reason usually cited for choosing not to conduct such reasoning is one of diminishing returns: formal specifications are widely believed to improve the quality of delivered systems without significant increase in cost; formal validation and refinement appears to be very expensive, with relatively few benefits to be gained.

No doubt, a major reason why there is little use of Z in this way is because there are few strong tools for proof in Z. (Specification, by contrast, is well-supported by some excellent type-checking tools.) Conversely, few have been willing to invest in the production of such tools because there appears to be little market for them. Other formal methodologies have been more successfully applied with extensive tool support.¹ Why have proof tools for Z been slow in appearing?

It is worth noting in passing that type-checking in Z has been very successful as a simple semantic check of well-formedness for specifications. This is largely because Z has a simple decidable Hindley-Milner type system (Spivey and Sufrin, 1990) which

*SVRC, School of Information Technology, The University of Queensland, Brisbane 4072, Queensland, Australia. apm@it.uq.edu.au

¹B (Abrial, 1996), for example, has developed alongside tools for its application.

can be statically checked. Moreover, a tool like *f*UZZ is able to use Z's abbreviation mechanisms to construct a system of subtypes, for helpful error diagnostics (Spivey, 1996). Z's approach with a decidable type system may be contrasted with that of PVS (see Sections 2.2.1 and 3.1.9).

To the casual observer, the Z notation is merely a style for writing terms of first-order predicate calculus and set theory. This is of course true in principle. Z is a first-order language, and a Z specification can be mechanically transformed into a collection of classical predicates. Classical logic and set theory are well-understood topics, and thus producing proof tools for Z does not sound like an interesting theoretical research problem.

We appear to be faced here with a paradox. Z is constructed using an easy theory, yet it is too expensive to produce useful tools for it. Many research projects have considered aspects of proof tool support for Z, and a few product-quality tools have been produced. Whilst hard figures are not available, anecdotal evidence suggests that uptake of these tools is small.

1.1 Semantics for Z

That Z is not quite the easy theory it first appears to be is evidenced by the number of different accounts of the semantics of Z which are available. The seminal work on Z semantics comes from Spivey (1988). This has been adapted considerably to give the semantics now presented in the Z Standard. The standard itself may be regarded as presenting two views of the semantics: the definitional (normative) part of the document gives a denotational model for Z using the relational calculus; an informative annex offers a logical theory which includes a deductive system for reasoning about Z specifications; it might be regarded as an axiomatic semantics in its own right. This second account is intended to be consistent with the first, though not equivalent (since it takes decisions over matters which the denotational description deliberately leaves loose—the issue of undefinedness, for example). A formal proof of soundness (modulo these issues) is envisaged, though not published at present.

Each proof tool described in this paper also effectively defines its own semantics for Z. For many, the correspondence with the standard is very close, though a proof of soundness or faithfulness is not usually offered. For others, the resemblance with Standard Z will merely be a passing likeness. Many texts also provide elements of a semantics for Z, offering either a (paper-based) proof system, or some form of model theory. Again, some of these are close to the Standard, (Woodcock and Davies, 1996), others approach from a very different angle, e.g. (Hodges, 1995).

1.2 Outline of the paper

The next section will explain the main challenges encountered in supporting proof in Z. Above all, the Z *schema notation* (which is the mechanism by which complexity is managed in Z specifications) appears to be the most problematic. Z schemas are unlike the usual terms of logic and set theory, and providing deep support for them is non-trivial.

A schema consists of a *declaration* of some typed variables and a (collection of) *predicate(s)* over those variables (and other variables of the specification). It corresponds to a generalisation of a record type in programming, but in popular Z style may also for example describe the global variables of a system, operations on those variables, or the promotion of one such (sub-)system into another.

Transformation into pure first-order predicate logic is possible (see the accounts of various systems which do this, below), but almost entirely obfuscates the original specification, and doing so soundly is again non-trivial. This semantic gap can be seen in something as seemingly trivial as how *substitution* works.

The second part of the paper surveys and attempts to classify the leading approaches to these problems. These may be classified into two major groups—those which support Z by encoding its semantics within an existing system (a logical framework, or generic theorem-prover); and those which have been implemented from scratch, especially for the task of supporting Z. In both groups we find a variety of experimental projects, as well as product-quality tools.

One of the original goals of the Z standardisation activity was to promote a widely-accepted means of reasoning about Z specifications. In a final section, we consider whether the emerging Z standard may assist in solving some of the problems outlined here.

Spivey and the Standard Some of the problems which will be described here are a result of the liberal approach taken by the Z Standard. The Z of Spivey's Reference Manual (1992) is more restrictive (in the use of arbitrary schemas as expressions, for example) and thereby avoids some (though not all) of these issues.

To some extent, the restrictions imposed by the Z Reference Manual correspond to good stylistic advice for aiding readability and simplifying proof. For a variety of reasons, the Standards committee has chosen to take a more liberal approach, and at length a reasonable semantics for it has been defined. It will be for the users to decide if the greater freedom of expression is worthwhile.

2 Challenges in supporting proof in Z

This section explains some of the reasons why supporting Z is not trivial. The first subsection deals with direct usability issues; the second with more profound semantic matters.

2.1 Problems of Syntax and Methodology

Much of Z is a stylised way of writing traditional mathematics. The notation is terse because mathematicians have generally aimed for conciseness of notation in order to ease both the discovery of proofs and their recording. To reproduce the notation accurately requires a moderately advanced typesetting/word-processing system. Other specification notations are deliberately text-based (e.g. the annotations in SPARK Ada (Barnes, 1997)). Many systems use a special syntax for specification, and another for proof. This complicates their use, and increases the potential for mistakes.

The concrete syntax of the specification/proof is of no theoretical interest. However, in the software engineering context it is quite significant, for two reasons:

1. Changes of notation are a barrier to understanding. Whilst most people are able to cope reasonably well with different syntaxes for input, printed/typeset specifications, and tool interactions, the cost of learning all three is high, and the scope for mistakes very great. For example, two closely-related tools are used regularly by the author; each uses a different binding priority for certain infix operators. The result is that many hours have been wasted because terms which look the same in both systems actually have radically different meanings.

2. Z specifications are commonly used in high-integrity developments. Here, it is most important that procedures for configuration management and version control are followed. Changes are inevitable during a project; either due to changing requirements, or the discovery of errors. The propagation of changes through specification documents, validations, refinements, and code must be recorded and traceable.

If a collection of tools is used which do not share common input formats, the development process will be hard to manage. Automated translation from one form to another is usually possible, but the retention of comments/line numbers etc. may be very *ad hoc*. Moreover, the status of proofs needs to be part of the managed configuration, so that the extent of changes in requirements and specifications can be accurately determined.

These problems are not unique to Z, but do distinguish the proof activity in software engineering from the more general concerns of the proof tool community. (Because proofs in group theory, say, are unlikely to be subject to changing assumptions.) They are the subject of ongoing research.

These concerns seem to mitigate against use of off-the-shelf proof tools.

Notice how well a tool like *fUZZ* fares against these criteria. It can be inserted in the process with virtually no overhead (provided the specifiers use *L^AT_EX*) and it can be run as a batch job in order to achieve ‘regression testing’ for specification and design documents.

An alternative approach is to use a text-based version of Z for specification and interaction with tools. The Standard offers an *email lexis* for such purposes, but this is yet to be widely taken up. The Cogito project (Bloesch et al., 1994) defines its own Z subset² (called *Sum*) which is written using simple ASCII characters. Unfortunately, interaction with Cogito’s theorem prover *Ergo* uses a different (though closely-related) syntax.

A common method for managing complexity in development is to use a module system. Various module schemes have been proposed for Z,³ but none has achieved widespread use. The Z Standard has a notion of *section*, which provides the simplest possible form of modularity but offers no formal support for the separate development of modules—in particular, there is no separation of name-spaces.

Cogito has an extensive module system (Nickson, Traynor, and Utting, 1996), with the module structure of the specification reflected in a theory structure in the proof tool. Nevertheless, this modularity does not support hiding, and so is not able to guarantee safe data refinement without restrictions on module usage.

2.2 Technical (Semantic) Problems

As we have observed, there are a number of different accounts of the semantics of Z. The issues raised here are discussed relative to the Z Standard’s account, which in most cases coincides with Spivey’s (1988). Of course, the problems encountered will depend to some extent on the underlying framework (logical or computational) in which the semantics are to be expressed. The genericity of most of the operators in the Z toolkit, for example, is problematic in some systems.

²Sum is not strictly a subset of Z: it also extends the language with annotations to support validation, and an extensive module system.

³One way to view *object orientation* is as a style for modularity. Various object-oriented Z notations have been proposed, most are surveyed in (Stepney, Barden, and Cooper, 1992).

The issues highlighted here are problems common to most systems for reasoning about Z specifications.

2.2.1 Partial Functions

Common Z style makes heavy use of partial functions — as does the Z mathematical toolkit. It is therefore important that the user of Z should ensure that whenever a function is applied, it is defined at its point of application.⁴ Z/EVES has made such a check largely automatic—thus permitting it to be part of a test of well-formedness, like type-checking. (Of course, in general, the test is undecidable, but for many practical specifications, this is not a problem.) Apparently, few real specifications pass this test (Saaltink, 1997).

Most proof systems for Z incorporate the well-definedness check as part of the rule for function application. In the logic of the Standard, for example, we have

$$\frac{\Gamma \vdash (e, u) \in f \wedge \forall y : f \bullet y.1 = e \Rightarrow y.2 = u}{\Gamma \vdash u = f e} \quad [y \notin \phi e \cup \phi u]$$

Such frequent checks often serve to complicate proofs, but present no theoretical difficulty. A good theorem-prover is even able to store such results for reuse wherever they are encountered.

It is worthwhile to contrast here the approach taken in PVS, where every function must be total. This is achieved using a system of subtypes, so the problem is transformed into one of type-checking (and so the PVS type system is undecidable; demonstrating type-correctness typically entails proof).

If Z partial functions are applied outside their domains, the result depends on the particular flavour of Z semantics being used. A common error is to apply the set cardinality function # to a set which is not known to be finite:

$$\frac{set : \mathbb{P}\mathbb{N}}{\#set < 10}$$

The function # is partial: its result is not defined for infinite sets, so the truth value of #set < 10 is not determined. The standard has been careful to avoid giving a value to such expressions, so that any reasonable interpretation can be used. The following principle, succinctly described by Peter Lupton (1991), is strongly encouraged

Undefinedness should not be exploited in specifications.

The logic presented in the standard offers one particular way to deal with this issue.⁵ All terms denote values (so for any expression t , we have that $t = t$), but in order to replace a function application with its result, the expression being applied must be shown to be functional at the point of application (as in the rule above). A related resolution is that embodied by the Z/EVES principle *don't ask*.

Observe that Z differs from VDM in this area in that Z's predicates are classical—either *true* or *false*—whereas VDM uses a carefully-crafted logic of partial functions, with an extra 'non-value' logical value.

⁴More accurately, a sufficient check is that whenever a *relation* is applied to an argument, it is *defined* and *functional* at that point.

⁵Observe that the logic has the status of 'informative' in the standard. This is contrasted with the 'normative' (definitional) parts, wherein function application is given a deliberately loose definition.

Although based on Z, Cogito follows VDM more closely in this regard. An extra value is added to the underlying model. The value \downarrow ('bad') is taken by any improper term. It is not a member of any type; indeed, it is not even a ZF set. Most operators are strict with respect to 'bad', but in logical terms it forms an equivalence class with 'false'.

A detailed study of the possible treatments of undefinedness in specification notations is outside the scope of this paper. Our aim here is merely to flag it as an issue which complicates reasoning in Z. The problem is one both of logic *and* methodology. A good survey of approaches can be found in (Brien, 1995).

2.2.2 Z Schemas

Z schemas are certainly the most interesting feature of the notation. They might be viewed simply as macros giving a name to a combination of declarations and predicates over the declared variables.⁶ However, in most accounts of Z semantics, schemas have a meaning of their own. In their greatest generality, Z schemas are quite unlike any terms normally arising in classical logics. This unusual nature of schemas is what makes producing a logical system for Z using familiar components into quite a challenging task.

Roles for Schemas Z has three major syntactic classes: declarations, expressions, and predicates. In general, these are strictly non-overlapping (there are no boolean-valued functions, for example, to be used as predicates), *but* a schema may appear in any of these roles.

A simple Z schema might be written

$$\frac{S}{\begin{array}{|l} x : \mathbb{N} \\ \hline x < y \end{array}}$$

Writing such a declaration in a Z specification is equivalent to defining⁷

$$S == [x : \mathbb{N} \mid x < y]$$

The right-hand side of this declaration is a schema in 'horizontal format'. This is an instance of a schema used as an *expression*. A schema used as an expression denotes a set of *bindings*—see below. Schema expressions may be combined using the operators of the *schema calculus*, so if we have schemas T_1 and T_2 , we may define schema T , say, as

$$T == T_1 \vee T_2$$

This approach is commonly used when schemas describe operations— T_1 may denote the successful operation, and T_2 the error case(s).

A schema can also be used as a declaration:

⁶This was, initially, precisely the meaning, according to some. A paper by Brien (1994) surveys the historical development of Z.

⁷Observe that in (Spivey, 1992) schema declarations are introduced using $\hat{=}$, and 'abbreviations' using $==$. Standard Z uses the second symbol for both purposes.

$$\frac{x : \mathbb{N}}{x < y}$$

This ‘axiomatic declaration’ uses the same schema as before, but now instead of giving a name to the schema, it is used directly to define a global variable x , and place a constraint on its value. The same declaration occurs between the quantifier and the \bullet in the following predicate:

$$\forall x : \mathbb{N} \mid x < y \bullet x = 0$$

Given the declaration above, this predicate may also be written

$$\forall S \bullet x = 0$$

Thirdly, a schema can be used as a predicate. Again, using the declaration above, we may write

$$\forall x : X \bullet S$$

This would be equivalent (in most cases) to writing

$$\forall x : X \bullet x \in \mathbb{N} \wedge x < y$$

Free variables, etc. In these various roles and forms, the schema has different properties. The variables declared in a schema are ‘free’ when the schema is used as a predicate, and something akin to ‘bound’ when it appears in an expression. The terms ‘bound’ and ‘free’ do not have quite their traditional meanings, however.

The standard uses the notations $\phi(e)$ and $\Phi(p)$ to denote the free variables of expression e and predicate p respectively. We have

$$\phi[x : \mathbb{N} \mid x < y] = \{y\}$$

(Strictly, \mathbb{N} and $(_ < _)$ are also free variables, but we shall overlook that detail here.) Whereas, $\phi(S) = \{S\}$. Moreover,

$$\Phi[x : \mathbb{N} \mid x < y] = \{x, y\}$$

but $\Phi(S) = \{S, x\}$.

There are two points to note here. First, the name of the schema is a proper Z name. It may be subject to quantification, for example (as a simple variable, as in $\forall S : \mathbb{P}\mathbb{N} \bullet b \in S$, or even as a schema: $\forall S : \mathbb{P}[x : \mathbb{N} \mid x < y] \bullet b \in S$), and so its meaning must be determined by context, and not by a global substitution (like a macro). Secondly, the use of y in the definition of S is fixed as the y which was in scope at the point of declaration. Thus when S is used as a predicate, y is not a free variable.

The consequence of this is that a schema reference (i.e. use of a schema name like S as a declaration, predicate, or expression) *cannot* in general simply be replaced by its definition.

The challenge, then, in supporting this notation in a system set up to deal with traditional notions of bound/free variables is that the free variables of a term are dependent on context. Logical frameworks which provide facilities for efficient evaluation of free variables but do not offer access to the context will not be able to support the semantics of Z in its greatest generality. Approaches to this problem are surveyed in Section 3.1.

Bindings An alternative approach to understanding Z schemas is to consider bindings. When a schema is used as an expression, it denotes a set of bindings, mappings from *names* to *values*. For a schema T with component variables t_1, \dots, t_n , we would write an element of T as

$$\langle t_1 == e_1, \dots, t_n == e_n \rangle$$

Thus, if y had the value 3 when S was declared above, we would have

$$S = \{ \langle x == 0 \rangle, \langle x == 1 \rangle, \langle x == 2 \rangle \}$$

Bindings are a simple generalisation of tuples (a *labelled* product rather than a cartesian product), but, again, they are not common in logical frameworks. A distinctive binding for a schema is that one which maps each name to its corresponding value in the current context. For schema T , this is θT .

$$\theta T = \langle t_1 == t_1, \dots, t_n == t_n \rangle$$

When a schema appears as an expression, then, it denotes a set of bindings—those bindings which have the same alphabet (i.e. which declare/use the same set of names) as the schema and give values to the variables in such a way as to make the predicate part of the schema equivalent to true. When a schema T appears as a predicate, it denotes the predicate

$$\theta T \in T$$

This predicate may be read as saying simply that the binding which maps the variables defined in T to their values in the current context is one which satisfies the declaration and predicate parts of T .

A schema which appears as a declaration can be viewed as introducing a binding of that schema type, and using that binding as a local definition.

$$(\forall S \bullet P) \Leftrightarrow (\forall b : S \bullet (\mathbf{let} \ b \bullet P))$$

(subject to side-conditions to prevent variable capture). See below for a discussion of local definitions and substitution.

Semantics This view of a schema as a set of bindings is made concrete in the Standard's semantics for schemas, where for a schema S , we have

$$\llbracket S \rrbracket^M : Env \leftrightarrow Situation$$

Here *Env* is the *environment*, that is, a mapping of names to values, and *Situation* is a mapping from names to type-value pairs (where the value must be a member of the type). Thus in any given environment, the schema may denote a number of situations. Bindings are a simplification of situations in that they map simply from names to values (omitting the type part).

The denotation of a predicate is the set of environments in which that predicate is true. A schema S used as a predicate is true in an environment iff that environment contains a situation of the schema S .

Substitution Whilst bindings serve to complicate the language considerably, they are also very useful for describing substitution in Z. This is a major contribution of the paper of Woodcock and Brien (1992). Because the notion of free variables in Z is context-dependent, so too must be any formulation of substitution.

Z uses a dot to denote binding selection, so $\langle x == 3, y == 4 \rangle.y$ takes the value 4. We may promote this selection operator so that $\langle x == t \rangle_{\odot} e$ denotes the result of replacing every free occurrence of x in expression e by t , and $\langle x == t \rangle_{\odot} p$ denotes the result of replacing every free occurrence of x in predicate p by t .⁸ More generally, an arbitrary binding can be used as a substitution. This notation is merely a shorthand for the local declaration which Spivey writes using **let**. He notes that **let** can be written using other Z operators, so that for expressions

$$\begin{aligned} (\mathbf{let} \ x_1 == e_1; \dots; x_n == e_n \bullet e) \\ = (\mu \ x_1 : t_1; \dots; x_n : t_n \mid x_1 = e_1; \dots x_n = e_n \bullet e) \end{aligned}$$

and for predicates

$$\begin{aligned} (\mathbf{let} \ x_1 == e_1; \dots; x_n == e_n \bullet p) \\ \Leftrightarrow (\exists_1 \ x_1 : t_1; \dots; x_n : t_n \mid x_1 = e_1; \dots x_n = e_n \bullet p) \end{aligned}$$

(subject to certain side-conditions in each case).

The short forms mentioned above permit the description of inference rules for quantifiers as follows:

$$\frac{\Gamma \vdash \forall S \bullet P \quad \Gamma \vdash b \in S}{\Gamma \vdash b \odot P} \quad [\text{AllE}]$$

This is a generalisation of the traditional form, which would use replacement $P[e/x]$ in the conclusion of the rule. The rule says that a universal quantification may be specialised by providing a binding b which satisfies its declaration part, assigning values to each of the variables. The specialised version of the predicate is obtained by using b as a substitution into P .

Moreover, this version is context-sensitive in that $b \odot P$ is to be evaluated in the context of the declarations in Γ .⁹ Clearly, for proof, a context-sensitive account of how to simplify these substitution terms is required. This is presented in the Standard.

3 Possible Approaches to Supporting Proof in Z

A significant number of proof tools aiming to support proof in Z have been described in the literature. The author would be pleased to hear of any not already mentioned here.

One way to classify the various proof tools for Z is to distinguish between those which encode/embed Z within an existing system, and those which have been implemented especially for supporting Z. The latter clearly have a better chance of meeting the concerns of Section 2.1. The dividing line is not always entirely sharp (see, for example, the description of *Ergo* in Section 3.1.6 below).

Z tool summaries have been published previously by Parker (1991) and Stegges and Hulance (1994).

⁸The symbols differ (\odot vs \odot) in order to help disambiguate substitution into a schema expression and a schema predicate: the results also differ, in general.

⁹ Γ is a sequence of Z paragraphs—schemas declarations, given sets, predicates, etc. The sequent form $\Gamma \vdash P$ should be read as ‘the specification Γ is sufficient to derive the predicate P .’

3.1 Encodings within a more general system

Many highly configurable proof support tools are available today. Some allow the user to define a logical theory for a particular problem domain within a well-understood logic (e.g. the HOL system); others—the ‘logical frameworks’—permit the definition of the logical system itself from scratch (e.g. Isabelle). This detail need not concern us.

However, the *level* of the embedding is of interest. Some encodings are at the syntactic level. A language of Z terms (term algebra) is defined within the host system, as are inference rules, such as those presented in the Z standard (logical frameworks are best suited to this task). Alternatively, the embedding may be semantic, providing a mapping from (something close to) Z into some pre-existing logical theory, such as higher-order logic. Semantic embeddings may be further subdivided into ‘deep’ and ‘shallow’ embeddings.

The paper of Bowen and Gordon (1994) discusses the issues involved in various levels of embedding. In general there will be a continuum of possible levels, rather than a sharp division. Typically, the deeper the embedding, the more abstract the results which may be proved. (A deep embedding might permit a proof of the commutativity of schema conjunction, say.) A shallow embedding may be good for proving results about a particular specification, but nothing more general. The embeddings which I have called ‘syntactic’ are somehow the deepest of all, in that the logical rules of (something like) Z are encoded, and little reliance is placed on the semantics of the host logic.¹⁰

A good reason for choosing to encode support for Z within a more general system is that these systems tend to be well-designed and efficient. They have sizeable user communities with much experience of their use, and large libraries of tactics and example theories available for direct re-use or minor adaptation.

Conversely, if one is concerned to demonstrate the soundness of one’s encoding, or compliance to the Z Standard, such an encoding may present a major burden. A system such as HOL will ensure the soundness of proofs which it produces, but this is useful only if the axiomatisation of Z introduces no inconsistency. Moreover, the transformation from Z into an alternative semantic framework must be trusted (or the chosen semantic model shown consistent with that of the Standard). In the area of schemas, as discussed in Section 2.2.2, this can be particularly problematic. Finally, the majority of such frameworks will require considerable work if support for Z syntax—and user interaction using the same notation—is to be arranged.

3.1.1 *zedB*

zedB (Neilson and Prasad, 1992) was an early attempt to support proof in Z using the **B**-Tool. Because **B**’s set theory and logic is essentially identical to Z ’s, the manipulation of Z expressions using **B** is very profitable. **B** does not support the schema notation, however, so some tricks were employed to permit the simplification of schema expressions. The soundness of these was never fully explored.

B has now become much less general-purpose, so this is not a viable future implementation route.

¹⁰The term ‘syntactic embedding’ is the author’s invention. Notice that it is at the opposite end of the spectrum from a syntactic *transformation*, which generally accomplishes a shallow embedding.

3.1.2 Z/EVES

EVES is a relatively highly automated proof tool (compared to the other proof systems described here) for predicate calculus and (untyped) set theory. The Z/EVES system (Saaltink, 1997) translates Z specifications into the core language of EVES, and supports the production of proofs about those specifications. The results of the application of proof steps (typically large steps, due to the high degree of automation) are transformed back into the Z notation.

As indicated above, considerable effort has been put into checking (in an automated manner wherever possible) that partial functions are applied within their domains—with the perhaps surprising result that very many specifications checked using the tool have failed.

An earlier paper on Z/EVES (Saaltink, 1992) explains some of the difficulties encountered in translating expressions involving schemas and bindings into the untyped base language of EVES, as one would expect from the account above. No full solution is presented there, but one has since been discovered so Z/EVES now covers virtually the whole of Z. As with many of the systems described here, Z/EVES displays poor performance when the number of variables in a schema becomes large (which may happen surprisingly easily, due to Z's structuring schemes).

3.1.3 Z-in-Isabelle

Kraan and Baumann (1995) have produced a ‘deep semantic embedding’ of Z the logical framework Isabelle. They have produced a deductive system based on that in the Z Standard (v1.0) using Isabelle’s simple logical theory *Lkz* as a basis (since *Lkz* is a sequent calculus in the same style as that of \mathcal{W}). The object language (Z) is represented in Isabelle using simply typed lambda calculus. This provides automated support for product and power types (though not bindings) so all deductions are automatically type-correct. Quantifiers and other binding constructs (such as set comprehension) are expressed using lambda-abstraction. The predicate $\forall x : X \bullet P$ is represented as $\forall(\lambda x \bullet x \in X \wedge P)$. Using this style, Isabelle hides much of the detail involved in proof; usual side conditions about variable capture are automatically respected (indeed, rules are not permitted to have side-conditions), and alpha-conversion is automatic where necessary.

The paper cited explains how this approach necessitates the expansion of schema references at the outset; this is done in a preprocessing step. As we have seen, to do this entails strong assumptions about the name spaces used in the specification. Clearly, it also results in very long formulae, and thus has a high impact on efficiency. The former problem is mitigated by Isabelle’s pretty printer—the concrete syntax for the expanded schema is retained as simply its name. Preprocessing is also used to eliminate a potential problem arising from the use of schemas as both predicates and expressions. Approximately, variables arising in schemas as predicates should be free; those in schema expressions should be bound. Bound variables are not appropriate because *in Z the names matter*: lambda-bound variables are anonymous, making schema calculus operations impossible. The Z-in-Isabelle solution is to replace all occurrences of schemas as expressions S with $\{ S \bullet \theta S \}$, before replacing these instances of S by their definitions.

By making these simplifications, Z-in-Isabelle is able to provide a useful proof environment for a large class of reasoning problems in Z.

3.1.4 Z in Isabelle/HOL

An indication of the great generality of Isabelle is that a second encoding of Z in Isabelle has been produced by Kolyang, Santen, and Wolff (1996), and this has little in common with Z-in-Isabelle. This encoding is built using Isabelle’s HOL theory. Its chief contribution is to provide an encoding which still allows schemas to be modelled as logical entities (in fact, as predicates). It is claimed that this is done in a manner which ‘essentially conforms with latest draft of the Z standard’. The encoding is nevertheless described as ‘shallow’ because not all aspects of Z semantics are represented in logical terms—Z’s distinctive treatment of *names* is not represented in the logic; the parser keeps track of these, and schema component variables are essentially anonymous (i.e. schema equivalence is preserved only up to alpha-conversion).

Schemas are denoted by characteristic functions. A schema is represented as a predicate over a tuple of variables. A schema and its representation are shown below.

S	
$x : \mathbb{N}$	
$y : \mathbb{Z}$	
$f : \mathbb{Z} \rightarrow \mathbb{Z}$	
$P \ x \ f \ y$	
	<pre> const S :: "int <=> int * int * int => bool". def S == SB f.SB x.SB0 y.[x:ℕ & y:ℤ & f:ℤ>-->ℤ P x f y] </pre>

The term SB is defined so that it behaves like a quantifier, and is suppressed by the pretty-printer; that is, a schema is denoted by a lambda-abstraction. The parser keeps track of schema signatures internally, so that the reference to schema S in a context may be replaced by an application $S \ x$, where x represents those variables of the surrounding context which are to be identified with those declared in S . Observe that at this level (hidden from the user) the identifiers in the schema’s signature are sorted lexicographically, to facilitate matching.

This representation permits user reasoning about schemas—staying at the level of the schema calculus, say—without expanding the schema definitions. In this way, Kolyang et al. achieve similar results to those presented in the logic of the Standard. Again, large schemas present performance problems.

3.1.5 Jigsaw v1

Jigsaw v1 (Martin, 1993) uses the 2OBJ logical framework to implement a proof tool based on \mathcal{W} (Woodcock and Brien, 1992). In so far as \mathcal{W} is sound,¹¹ this is a successful attempt to provide a proof tool faithful to the Z standard. However, the overheads involved in supporting the context-sensitive notions of free variables and substitution using 2OBJ (which was itself a very experimental tool) rendered the resulting system cripplingly inefficient.

3.1.6 Cogito

Cogito (Nickson et al., 1996) offers an integrated methodology and tool-set for Z-style developments from specification through to executable code (in Ada, at present). A variant of Z, called *Sum* is used as the specification notation, and the accompanying proof tool is called *Ergo*.

¹¹ \mathcal{W} turns out to place heavy restrictions on the naming of variables, and so is not very practical.

Ergo is a generic theorem-prover, but Cogito is its biggest user, and has thus influenced the design of both the core tool and its libraries. Sum is modelled using Ergo's Zermelo-Fraenkel theory of sets. The modelling follows the semantics of the standard, as mentioned in Section 2.2.2. Therefore, when a Sum specification is translated into Ergo, the environment in which each predicate is to be evaluated is made explicit. Schemas are characteristic functions, returning '*true*' if the environment contains a situation of the schema, and '*false*' otherwise. The resulting predicates look rather forbidding, but advanced tactics simplify them into more recognisable forms.

This approach has elements in common with the approach of Z in Isabelle/HOL (Section 3.1.4) in the use of characteristic functions to model schemas, but here the names are used within the logical theory to match schema components to the current environment, whereas in Isabelle/HOL this work is accomplished entirely in the parser.

The result is a potentially faithful model of the context-sensitive schema semantics presented above, though in a rather impenetrable form. Bindings are not supported, and so schemas as types are not allowed, though it would be a straightforward task to add this facility within the existing model.

3.1.7 Z/HOL

A paper by Bowen and Gordon (1994), (1995) describes a simple “shallow” semantic embedding’ of Z in the HOL system. This approach makes no special concessions to Z's unusual use of names, though it offers limited support for bindings as tuples of name-value pairs. The paper serves as a good description of the issues surrounding semantic embeddings (in HOL, in particular).

The paper offers the following shibboleth as a means of distinguishing shallow and deep embeddings: only in the latter will the property of commutativity of schema conjunction be provable. In a shallow embedding the schema conjunction will be converted into something else (logical conjunction, typically), rendering the question meaningless.

3.1.8 ProofPower

In contrast, ProofPower is quite a deep embedding of Z in HOL, complete with extensive support for reasoning about Z specifications. Early papers on ProofPower include those by Arthan (1991) and Jones (1992); recent information is to be found on a web site (Arthan, 1997). In ProofPower Z, bindings are accurately modelled as logical objects, and the type system is extended to support this use.

Bowen and Gordon (1994) remark, however that this encoding is still insufficient to be able to prove the commutativity of schema conjunction. This is largely because there is no way to write a postulate about ‘all schemas’; the modelling has sufficient depth. They further remark that there is no single semantic function defined in the logic that maps Z syntax into its meaning.

ProofPower is committed to compliance with the emerging Z standard, though it does not appear that a formal result about the faithfulness of its logic with respect to the Standard's semantics is expected. Of all the tools described in this paper, ProofPower is probably the most mature.

3.1.9 PVS and Z

As part of the ProCos project, Engel and Skakkebæk (1994) describe the application of the PVS system to Z specifications (particularly those in a timed variant of Z, for use with the duration calculus). We might call this a ‘super-shallow’ embedding, since it actually entails a translation from Z into the specification language of PVS. All proofs are carried out in PVS, with no special reference to the Z specifications from which the terms originate.

In this case, the mapping from Z schemas to PVS theories is described as being quite straightforward; there is a close, though not perfect match between the two. The language of Z expressions is a greater challenge, however, as PVS is based on a theory of total functions, in contrast to Z’s set theory and partial functions. A faithful model of Z functions in PVS is presented, but as a consequence, the resulting PVS terms are rather complicated, and cannot utilise much of the power of the automation provided in PVS for specifications written in its own native style. PVS and Z remain antithetical in their approaches.

Also relevant, since it deals with B’s abstract machine notation, which is closely related to Z, is work by Pratten (1995) on using PVS in place of the B tool.

3.1.10 Z in LEGO/Type Theory

Maharaj (1994) describes a careful study of how to encode the schema calculus using the unifying theory of dependent types (UTT), as implemented in the LEGO proof-checker. Various possible representations are explored, taking account of some of the issues raised in Section 2.2.2. The use of schemas to define types is explicitly excluded, and since UTT employs an intuitionistic logic (whereas Z is generally taken to be classical), the encoding necessarily covers only a restricted part of Z. Earlier work (1990) describes ‘implementing’ Z using LEGO.

3.2 Special Implementations

Clearly, the alternative to using a logical framework or general-purpose proof tool is to construct one from scratch. In doing so, one can avoid all interface problems, by arranging for tool interfaces to permit real Z specifications to be read, written, and manipulated. Moreover, in this way it is possible to side-step problems of soundness by implementing precisely a published account of a logic for Z, such as the one in the Z standard.¹² The implementation is able to provide full support for bindings and schemas as first-class objects, and to evaluate accurately Z’s context-sensitive notions of bound and free variables.

The chief drawback of this approach is the inherently small base of theories, tactics, and user experience on which to draw. A longer implementation time is also to be expected, of course, and potentially poorer performance of the finished product (though **Jigsaw1**, implemented with 2OBJ, was between two and three orders of magnitude *slower* than **Jigsaw2**, implemented directly in Haskell).

¹²As no proof of the soundness of that system is offered, the question of whether or not this is wise is moot. Moreover, as soon as the implementor chooses to change/enhance the published account, a new problem of soundness arises.

3.2.1 JigsaW v2

A second version of *JigsaW* has been constructed, implemented directly in Haskell. Some of its features are described in (Brien and Martin, 1995), as it has been developed with the specific aim of supporting *precisely* the logic described in the Z Standard. Because the Z Standard has yet to stabilise, the tool is not yet in a generally-usable state.

3.2.2 Zola (Balzac)

Zola (a commercial version of the tool developed as *Balzac* (Harwood, 1991; Ashoo, 1992)) is based on a custom-designed logic, closely related to \mathcal{W} .

The tool represents a considerable number of man-years of effort, and so is relatively mature. It has been developed in close co-operation with the Standards activity, and may be expected to conform with the Z Standard.

3.2.3 CADiZ

The CADiZ tool development has gone through a number of stages. Early versions supported type-checking and typesetting of Z specifications (Jordan, McDermid, and Toyn, 1991; Toyn and McDermid, 1995). More recently, CADiZ has begun to support proof (Toyn, 1996). The tool has a highly visual user interface, and is based on \mathcal{W} . An *ad hoc* implementation of substitution is used, but conformance (or convergence) with the Standard is claimed. A tactic language similar to that used in *JigsaW* is available for writing proof procedures.

4 Concluding Remarks

This paper has surveyed some of the features of Z which make the provision of proof tool support an interesting task. It has also considered the approaches which have been taken to date in meeting that challenge.

The key feature of Z which complicates tool support is its use of names, and the consequential complexity of the semantics of Z schemas. This has a wide impact because it means that substitution in Z is not readily implemented using the replacement/rewriting tools which come with most proof frameworks.

4.1 The state of the art

If a generic theorem-proving tool is used, we have seen that a difference of basic concepts may render the resulting system somewhat distant from Z. In some cases this is quite marked—PVS, for example, uses entirely the wrong paradigm—in others it is less immediately obvious, but lacking bindings as first-class objects, or context-sensitive free variable calculations, is a significant problem. By making shallow embeddings, such problems can be avoided for large classes of specifications, and a useful tool constructed. However, the translation to a host logic is necessarily a complicated step, requiring much of the apparatus of a Z logic itself. Having made the translation, the user must reason in something akin to a model theory for Z, which has attendant problems. The success of Z as a specification notation can be attributed largely to the structuring made possible by schemas. If this structuring is lost or obscured in the proof activity, proof will quickly become difficult and/or error-prone.

Specially-constructed tools are starting from ‘further back’ and still have a lot of catching-up to do. They have the potential to support Z substitution, schemas and bindings at a high level, but much of the benefit of doing so is yet to be realised.

In both cases, soundness with respect to the Standard semantics of Z has not been formally addressed. In this regard, the specially-constructed tools possibly have an advantage, since they incorporate a single logical system which is amenable to a proof of soundness. In all but the deepest embeddings, by contrast, some elements of the proof of soundness will rely on the work of the translator, and some on the way that terms are embedded in the host logic.

This form of soundness is often termed ‘faithfulness’. There may be little doubt that the host logic is itself sound, but that is not a guarantee that the transformation of Z terms into that system produces a sound (faithful) means of reasoning about Z specifications.

For the time being, there is a spectrum of options, some of which inspire higher confidence in their probable soundness, some of which are obviously supporting Z (as distinct from something which appears similar to Z , but in fact has quite different semantics), and some of which are actually useful for doing industrially-relevant proofs.

4.2 Future Directions

One way to mitigate questions about soundness, whilst retaining access to existing proof tools, might be to develop a framework in which a well-established proof tool is used to discover proofs, and a small, highly-trusted custom-built proof-checker is used to validate them.

In any case, it is highly desirable to achieve libraries of theories, lemmas and tactics for the Z world. There is every reason to suppose that such libraries could be made portable across various proof tools, since most work from a common collection of definitions in the Z mathematical toolkit. Spivey’s account of the mathematical toolkit includes a large number of laws relating the constructs defined there. A library of such laws *together with tactics describing their proofs* should be portable enough to be incorporated into any proof tool supporting Z .

Reasoning about schemas is, as we have seen, more problematic than proving toolkit properties. Since most Z specifications will not use the most exotic properties of schemas, however, even the systems which take the greatest liberties with schema semantics will often deliver correct results. Therefore, we might go further and consider proof strategies appropriate to common Z activities (standard theorems about specifications, etc.) which could again be used in both interactive tools, and trusted proof-checkers.

In conclusion, whilst no single proof tool technology is a clear winner at present, given that we have general agreement on the semantics of Z and the definitions to be used in the mathematical toolkit, there is every reason to suppose that we should be able to extend the Z mathematical toolkit into a proof toolkit. If we can achieve this, the implementation technology may eventually become irrelevant.

References

Abrial, J.-R. (1996). *The B-Book: Assigning Programs to Meanings*, Cambridge University Press.

- Arthan, R. D. (1991). Formal specification of a proof tool, in S. Prehn and W. J. Toetenel (eds), *VDM'91: Formal Software Development Methods*, Vol. 551 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 356–370.
- Arthan, R. D. (1997). The ProofPower web pages.
URL: <http://www.trireme.demon.co.uk/>
- Ashoo, K. (1992). The Genesis Z tool – an overview, *BCS-FACS FACTS Series II*, 3(1): 11–13.
- Barnes, J. (1997). *High Integrity Ada: The SPARK Approach*, Addison-Wesley.
- Bloesch, A., Kazmierczak, E., Kearney, P., and Traynor, O. (1994). The Cogito methodology and system, *Asia-Pacific Software Engineering Conference '94*, pp. 345–355.
- Bowen, J. P. and Gordon, M. J. C. (1994). Z and HOL, in Bowen and Hall (1994), pp. 141–167.
URL: <http://www.comlab.ox.ac.uk/archive/z/zum94.html>
- Bowen, J. P. and Gordon, M. J. C. (1995). A shallow embedding of Z in HOL, *Information and Software Technology* 37(5-6): 269–276.
- Bowen, J. P. and Hall, J. A. (eds) (1994). *Z User Workshop, Cambridge 1994*, Workshops in Computing, Springer-Verlag.
URL: <http://www.comlab.ox.ac.uk/archive/z/zum94.html>
- Bowen, J. P., Hinchey, M. G., and Till, D. (eds) (1997). *ZUM'97: The Z Formal Specification Notation, 10th International Conference of Z Users, Reading, UK, April 1997, Proceedings*, Vol. 1212 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin Heidelberg.
- Brien, S. M. (1994). The development of Z, in D. J. Andrews, J. F. Groote, and C. A. Middelburg (eds), *Semantics of Specification Languages (SoSL)*, Workshops in Computing, Springer-Verlag, pp. 1–14.
- Brien, S. M. (1995). *A Model and Logic for Generically Typed Set Theory (Z)*, D.Phil. thesis, University of Oxford. New version expected 1997.
- Brien, S. M. and Martin, A. P. (1995). A tutorial on proof in Standard Z, *Technical Monograph PRG-120*, Programming Research Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK. Presented at ZUM'95.
- Engel, M. and Skakkebak, J. U. (1994). Applying PVS to Z, *ProCoS II Technical Report IT/DTU ME 3/1*, Department of Computer Science, Technical University of Denmark.
URL: <ftp://ftp.id.dth.dk/pub/ProCoS/Marcin.Engel/IDDTH-ME-3-1.ps.Z>
- Harwood, W. T. (1991). Proof rules for Balzac, *Technical Report WTH/P7/001*, Imperial Software Technology, Cambridge, UK.
- Hodges, W. (1995). The meaning of specifications I: Domains and initial models, *Theoretical Computer Science* 152: 67–89.

- Jones, R. B. (1992). ICL ProofPower, *BCS-FACS FACTS Series III*, 1(1): 10–13.
- Jordan, D., McDermid, J. A., and Toyn, I. (1991). CADiZ – computer aided design in Z, in Nicholls (1991), pp. 93–104.
URL: <http://www.dcs.gla.ac.uk/springer-verlag/50.html>
- Kolyang, Santen, T., and Wolff, B. (1996). A structure preserving encoding of Z in Isabelle/HOL, *1996 International Conference on Theorem Proving in Higher Order Logic*, Springer-Verlag.
- Kraan, I. and Baumann, P. (1995). Implementing Z in Isabelle, in J. P. Bowen and M. G. Hinchey (eds), *ZUM'95: The Z Formal Specification Notation*, Vol. 967 of *LNCS*, Springer-Verlag, pp. 355–373.
- Lupton, P. J. L. (1991). Z and undefinedness, *Technical Report PRG/91/68*, Z Standards Panel / Programming Research Group.
- Maharaj, S. (1990). *Implementing Z in LEGO*, Msc thesis, The University of Edinburgh.
- Maharaj, S. (1994). Encoding Z-style schemas in type theory, in H. Geuves (ed.), *TYPES '93: Types for Proofs and Programs*, Vol. 806 of *Lecture Notes in Computer Science*, Springer-Verlag.
URL: <http://www.cs.stir.ac.uk/~sma/publications/SchemasinUTT.ps>
- Martin, A. (1993). Encoding W: A logic for Z in 2OBJ, in J. C. P. Woodcock and P. G. Larsen (eds), *FME'93: Industrial-Strength Formal Methods*, Vol. 670 of *Lecture Notes in Computer Science*, Formal Methods Europe, Springer-Verlag, pp. 462–481.
- Neilson, D. S. and Prasad, D. (1992). zedB: A proof tool for Z built on B, in Nicholls (1992), pp. 243–258.
URL: <http://www.dcs.gla.ac.uk/springer-verlag/21.html>
- Nicholls, J. E. (ed.) (1991). *Z User Workshop, Oxford 1990*, Workshops in Computing, Springer-Verlag.
URL: <http://www.dcs.gla.ac.uk/springer-verlag/50.html>
- Nicholls, J. E. (ed.) (1992). *Z User Workshop, York 1991*, Workshops in Computing, Springer-Verlag.
URL: <http://www.dcs.gla.ac.uk/springer-verlag/21.html>
- Nicholls, J. (ed.) (1995). *Z Notation*, Z Standards Panel, ISO Panel JTC1/SC22/WG19 (Rapporteur Group for Z). Version 1.2, ISO Committee Draft; CD 13568.
URL: <ftp://ftp.comlab.ox.ac.uk/pub/Zforum/ZSTAN/drafts>
- Nickson, R., Traynor, O., and Utting, M. (1996). Cogito Ergo Sum: Providing structured theorem prover support for specification formalisms, in K. Ramamohanarao (ed.), *Proceedings of the Nineteenth Australasian Computer Science Conference (ACSC'96)*, Vol. 18(1) of *Australian Computer Science Communications*, pp. 149–158.
- Parker, C. E. (1991). Z tools catalogue, *ZIP project report ZIP/BAe/90/020*, British Aerospace, Software Technology Department, Warton PR4 1AX, UK.

- Pratten, C. H. (1995). An introduction to proving AMN specifications with PVS and the AMN-PROOF tool, in H. Habrias (ed.), *Z Twenty Years on – What is its Future?*, IRIN (Institut de Recherche en Informatique de Nantes), Université de Nantes, France, pp. 149–165.
- Saaltink, M. (1992). Z and Eves, in Nicholls (1992), pp. 223–242.
URL: <http://www.ora.on.ca/biblio.html#mark:z-and-eves>
URL: <http://www.ora.on.ca/biblio.html#mark:z-eves>
- Saaltink, M. (1997). The Z/EVES system, in Bowen et al. (1997), pp. 72–85.
- Spivey, J. M. (1988). *Understanding Z: A Specification Language and its Formal Semantics*, Vol. 3 of *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press.
- Spivey, J. M. (1992). *The Z Notation: A Reference Manual*, second edn, Prentice-Hall.
- Spivey, J. M. and Sufrin, B. A. (1990). Type inference in Z, in D. Bjørner, C. A. R. Hoare, and H. Langmaack (eds), *VDM'90: VDM and Z—Formal Methods in Software Development*, Vol. 428 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 426–451.
- Spivey, M. (1996). Richer types for z, *Formal Aspects of Computing* **8**(5): 565–584.
- Steggles, P. and Hulance, J. (1994). Z tools survey. Imperial Software Technology Ltd. / Formal Systems (Europe) Ltd.
URL: <ftp://ftp.ist.co.uk/pub/doc/zola/ztool-survey.ps>
- Stepney, S., Barden, R., and Cooper, D. (eds) (1992). *Object Orientation in Z*, Workshops in Computing, Springer-Verlag.
URL: <http://www.dcs.gla.ac.uk/springer-verlag/30.html>
- Toyn, I. (1996). Formal reasoning in the Z notation using CADiZ, *Proc. 2nd Workshop on User Interfaces to Theorem Provers*, York.
URL: ftp://ftp.cs.york.ac.uk/hise_reports/cadiz/uitp.ps.Z
- Toyn, I. and McDermid, J. A. (1995). CADiZ: An architecture for Z tools and its implementation, *Software—Practice and Experience* **25**(3): 305–330.
- Woodcock, J. C. P. and Brien, S. M. (1992). *W: A Logic for Z*, *Proceedings 6th Z User Meeting*, Springer-Verlag.
- Woodcock, J. C. P. and Davies, J. (1996). *Using Z: Specification, Refinement, and Proof*, Prentice-Hall, Europe.

Acknowledgements

The perspectives described here have arisen out of many conversations with colleagues over many years. Stephen Brien and Jim Woodcock have been the most influential in helping my understanding. Other members of the Z Standards committee and the SVRC have contributed insight, too. Special thanks to Ian Toyn, Ina Kraan, Mark Saaltink, and Thomas Santen for discussions about their respective proof tools, and to Peter Kearney for comments on an earlier draft of this report.