Mechanizing Proof for the Z Toolkit*

R.D. Arthan

Lemma 1 Ltd.
2nd Floor, 31A Chain Street,
Reading UK RG1 2HX
rda@lemma-one.com

Abstract. This paper reports on theorems and proof procedures for working with the Z mathematical toolkit developed using the ProofPower system. This development has taken place in parallel with work on ProofPower itself over the last five or six years.
Since the underlying mathematics is not completely trivial and is largely independent of the theorem-proving technology, the body of theorems proved and proof procedures found useful should be relevant to other work on proof in Z. The work has also influenced the design of the toolkit itself. For example, the definitions of the arithmetic operators currently proposed for the draft ISO standard for Z are largely based on our formulation.

1 Introduction

1.1 Background

Much work on proof automation has relied on the use of problem descriptions tailored to meet the needs of the theorem-prover. When formal methods are used in a systems engineering context, the style used to specify a problem is often outside the control of the person attempting to carry out the proofs. To be effective in such a context, a theorem-proving system must assist the user with all the mathematical idioms that are likely to be encountered. This is not an easy challenge.

In the context of the Z notation [12], a common feature of nearly all specifications is that they make intensive use of the library of mathematical notions known as the *Z mathematical toolkit* (or just *the toolkit* for short). Even such a thing as the union operator for sets is defined in the toolkit rather than provided as a fixture of the Z language. To be useful in practice, any theorem-proving system for Z must therefore provide support for the toolkit.

The description of the toolkit in [12] does discuss proof in Z and includes a number of laws about the objects in the toolkit. The discussion is concerned with informal reasoning rather than formal machine-checked proof. The laws provide a useful source of test problems — many of the more elementary laws are amenable to completely automatic proof.

* This paper was presented at the Oxford Workshop on Automated Formal Methods 19–21 June, 1996.

1.2 ProofPower

For well over 5 years now, ICL have been engaged in the development and exploitation of a theorem-proving system called ProofPower. ProofPower began as a re-engineering of the Cambridge HOL tool [4]. It supports the same abstract logic as HOL and incorporates many features that were felt to be important as a result of ICL's experience using HOL for proofs of specification-to-model correspondence in highly assured secure systems development.

ProofPower can support object languages other than HOL by a technique known as semantic embedding. This concept is originally due to M.J.C Gordon in the context of logics for programming languages [5]. The method is just as appropriate for specification languages, and, given strong customer demand for the Z notation, ICL's work on ProofPower has always been aimed at supporting proof in Z. In 1993, ICL was able to begin to market a system offering support for a reasonably fully-featured dialect of the Z language and including many theorems and proof procedures for the toolkit.

ProofPower has evolved over the years, most recently in response to the requirements of a tool commissioned by the Defence Research Agency, Malvern. This *Compliance Tool*, implemented on top of ProofPower supports a notation designed by the Defence Research Agency [11, 8]. The *Compliance Notation* is used for the specification and verification of Ada programs in Z. The verification conditions generated by the Compliance Tool pose a nice challenge for proof automation, and this has been the target of much of our work in this area over the last two years.

1.3 Guide to this Document

In this paper, we attempt to survey the ProofPower treatment of mechanized proof with the toolkit. We will assume a nodding familiarity with the Z notation as described in [12], but a detailed knowledge of the subtler points of the language is not required. No knowledge of HOL is required, although at a couple of points the methods described involve a little bit of mixed-language working.

ProofPower is an interactive tool. It is controlled using the functional programming language Standard ML as a command language (*the metalanguage*). The ProofPower paradigm for developing specifications and proofs is the evolutionary, interactive, construction of documents containing both formal material and narrative text. The formal material is checked and processed by the tool as it is constructed. When development is complete, the file containing the document can be processed off-line, e.g., to prove and store the theorems about a section of the toolkit when a new version of the tool is built.

This paper is just such a ProofPower document. The formal material is highlighted by a bar in the right margin. Input to the tool is introduced by the tag "SML" in a small font. Output from the tool is introduced by the tag "ProofPower Output". Material with a marginal bar and no tag is simply being quoted by way of information. Here's an example which just shows ML in action as a programming language:

SML

```
(curry BasicIO.output std_out o implode o map chr)
    [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 33, 10];
```

The above commands (which assuredly and deliberately violate every tenet of good software engineering practice!) produce the following output:

ProofPower Output

Hello World!

The plan of this document is as follows. First of all, section 2 gives an overview of how proofs are conducted in the Z language using ProofPower; then sections 3 to 9 survey the facilities provided for reasoning about the toolkit under the following headings: *Sets, Relations, Functions, Arithmetic, Finiteness, Sequences* and *Others*. Finally, section 10 gives some concluding remarks including some tool-independent lessons which may be learnt from our work.

The organisation of the material in this document is slightly idealised and does not reflect the precise way the toolkit is modularised in ProofPower or in [12]. For example, the total function space arrow is discussed here under the heading *Functions*, but in practice has to be defined very early on since many of the definitions concerned with sets and relations use it.

Sections 2 to 6 contain many examples of simple proofs. These examples are intended to give a flavour of how ProofPower is used and are chosen for simplicity and to illustrate methods rather than any mathematical interest. In many cases, a fact whose proof involves several steps when done from first principles can be proved automatically using more advanced methods.

2 Reasoning in the Z Language

In sections 2.1 to 2.4 below, we describe briefly how reasoning is carried out in Z using ProofPower. The description is given under the following headings: *Propositional Calculus*, *Predicate Calculus*, *Z Expression Constructs*, and *Equational Reasoning*. It may seem striking that we make no mention of schemas or the schema calculus. However, schemas are not used in the toolkit and so are not relevant to our immediate concerns here. In fact, schemas and the schema calculus are fully supported in ProofPower and most of the tools and methods described here handle them in a uniform way.

2.1 Propositional Calculus

Almost any proof in Z will involve some reasoning about the propositional connectives. Any adequate proof tool will make such work trivial, and our only reason for going into it here is to introduce the style of proof supported by ProofPower. ProofPower is in the LCF tradition [6]; as with other LCF-based systems such as HOL [4], proofs are generally found using an interactive subsystem, the *subgoal package* based on ideas due to Paulson [10].

In an LCF-style system, the main user interface is provided by a strongly-typed functional programming language serving as the command language, or *metalanguage*. In the case of ProofPower, the metalanguage is Standard ML [13] extended to allow fragments of logical syntax to be quoted in a convenient fashion. To see this in action for Z, let us consider a very simple tautology: $\alpha \Rightarrow \alpha \lor \beta$. To embark on the proof of this, we execute a metalanguage command to invoke the subgoal package:

SML
```
set_goal([], ⓩα ⇒ α ∨ β⌝);
```

The symbols ⓩ and ⌝ act as brackets to delimit the Z predicate $\alpha \Rightarrow \alpha \lor \beta$. The system responds by echoing back the problem which is our goal:

ProofPower Output
```
Now 1 goal on the main goal stack

(* *** Goal "" *** *)

(* ?⊢ *)  ⓩα ⇒ α ∨ β⌝
```

We can now attempt to progress the proof by applying tactics. A *tactic* is a metalanguage function which attempts to reduce a goal by finding a list of zero or more subgoals which entail it. Behind the scenes, the tactic also computes a function value which can be used to verify this entailment once the subgoals have been verified.

Like many other LCF-style systems, ProofPower provides a tactic implementing a process referred to as *stripping*; this serves as a Swiss army knife for whittling away at goals by analysis of their principal connective (in this case $\Rightarrow$). We tell the subgoal package to apply this tactic (*strip_tac*) using the metalanguage function called *apply_tactic* or *a* for short:

SML
```
a(strip_tac);
```

Omitting some of the red tape, the system's response to this is:

ProofPower Output
```
(*  1 *)  ⓩα⌝

(* ?⊢ *)  ⓩα ∨ β⌝
```

Here we have a goal with a simpler *conclusion* $\alpha \lor \beta$ and an *assumption* α. So the tactic has implemented the usual way of proving an implication: to prove $A \Rightarrow B$, assume A, and then prove B on the basis of that assumption. Continuing our attack, another application of *strip_tac* transforms the goal to:

ProofPower Output

```
(* 1 *) ⌜α⌝

(* ?⊢ *) ⌜¬ α ⇒ β⌝
```

Here, rather than generating a new assumption, *strip_tac* has rewritten the disjunction as an implication, which turns out to fit in well with the other transformations it knows how to effect.

A final application of *strip_tac* completes our search for a proof:

ProofPower Output

```
Tactic produced 0 subgoals:
Current and main goal achieved
```

What has happened here is that *strip_tac* has attempted to transform the goal in the usual way for an implication, by adding the antecedent $\neg\ \alpha$ to the assumptions and reducing the conclusion to the succedent β; as it attempts to add the new assumption, it checks, *inter alia*, whether the new assumption contradicts an existing assumption, finds that it does, and so deduces that the transformed goal is an instance of the propositional axiom: $A, \neg A \vdash B$, and so the proof is complete.

In the usual LCF style, we can now extract the theorem we have proved for subsequent use. This step can be thought of as invoking the verification function computed behind the scenes by the tactics. This causes a theorem, i.e., a value of metalanguage type *THM*, to be computed. The LCF architecture ensures that any value of this type is a valid consequence of the axioms which are currently in force. The following metalanguage fragment causes the theorem to be bound to a metalanguage variable for future reference and to be saved away in a database:

SML

```
val thm1 = save_pop_thm"thm1";
```

The system responds with:

ProofPower Output

```
Now 0 goals on the main goal stack
val thm1 = ⊢ α ⇒ α ∨ β : THM
```

Mere iteration of the tactic *strip_tac* is a complete decision procedure for propositional tautologies. Here for example, is the proof of a less trivial tautology, using a so-called *tactical*, *REPEAT*, to carry out the iteration.

SML

```
set_goal([],⌜
        (φ x ∨ ψ x) ∧ (φ y ∨ ψ y) ∧ (φ z ∨ ψ z)
    ⇒       (φ x ∧ φ y) ∨ (φ x ∧ φ z) ∨ (φ y ∧ φ z)
        ∨   (ψ x ∧ ψ y) ∨ (ψ x ∧ ψ z) ∨ (ψ y ∧ ψ z)  ⌝);
a(REPEAT strip_tac);
val thm2 = save_pop_thm"thm2";
```

ProofPower Output

```
Now 0 goals on the main goal stack
val thm2 = ⊢ (φ x ∨ ψ x) ∧ (φ y ∨ ψ y) ∧ (φ z ∨ ψ z)
    ⇒ φ x ∧ φ y ∨ φ x ∧ φ z ∨ φ y ∧ φ z
    ∨ ψ x ∧ ψ y ∨ ψ x ∧ ψ z ∨ ψ y ∧ ψ z : THM
```

The output from the tool when a proof is complete brings joy to the eyes of an interactive user, but doesn't import much new information when seen on the printed page. The output just tells us that the proof is complete and shows the ML binding of a name to the new theorem. In the sequel, I will generally suppress the less informative parts of the output and just mark the tactic which completes a proof with an ML comment "(∗ *Done*! ∗)". As the final steps in some of the proofs are also suppressed, the names, *thm1*, *thm2* etc. used for the example theorems do not form a consecutive sequence in the printed document.

2.2 Predicate Calculus

The most elementary method provided by ProofPower for working with the predicate calculus in Z augments the approach to the propositional calculus described above with automatic treatment of existential assumptions and universal conclusions. Tactics are provided for working with universal assumptions and existential conclusions. Let's see this in action on propositions *10.5 and *10.51 from *Principia Mathematica* [1].

SML

```
set_goal([],(∗ ∗10.5 ∗) ⓩ(∃x:V• φx ∧ ψx) ⇒ (∃y:V• φy) ∧ (∃z:V• ψz)⌝);
a(REPEAT strip_tac);
```

This gives us two subgoals, the first being as follows:

ProofPower Output

```
(∗  3 ∗)  ⓩx ∈ V⌝
(∗  2 ∗)  ⓩφ x⌝
(∗  1 ∗)  ⓩψ x⌝

(∗ ?⊢ ∗)  ⓩ∃ y : V • φ y⌝
```

Here stripping has automatically carried out a natural style of reasoning: to prove $(\exists x{:}V\bullet\ A) \Rightarrow B$, assume $x \in V$ is such that A holds of it and prove B on the basis of that assumption. In this case, B is a conjunction and stripping continues by splitting the problem into a subgoal for each conjunct. What stripping cannot do is invent the existential witness required to complete the proof. However, we can easily see that x will do the job and the tactic *z_∃_tac* allows us to progress the proof further.

SML

```
a(z_∃_tacⓩx⌝);
```

This transforms the conclusion of the goal to

ProofPower Output

```
(* ?⊢ *)  ⌜x ∈ V ∧ true ∧ φ x⌝
```

and repeated stripping will complete the proof, since the goal is now a propositional tautology. (The apparently spurious "∧ *true*" here comes from part of the existential quantification which has been elided in this problem. The general form in Z is "∃ *declarations* | *predicate* • *predicate*", and here we see the default value taken when the first predicate is omitted.)

The treatment of universal assumptions is seen in the treatment of *10.51:

SML

```
set_goal([],(* *10.51 *) ⌜¬(∃x:V• φx ∧ ψx) ⇒ (∀y:V• φy ⇒ ¬ ψy)⌝);
a(REPEAT strip_tac);
```

The stripping process pushes negations through quantifiers before they arrive in the assumptions, so this produces the following goal:

ProofPower Output

```
(*  3 *)  ⌜∀ x : V • ¬ (φ x ∧ ψ x)⌝
(*  2 *)  ⌜y ∈ V⌝
(*  1 *)  ⌜φ y⌝

(* ?⊢ *)  ⌜¬ ψ y⌝
```

We now need to specialise the universally quantified assumption to y. This is done as follows:

SML

```
a(z_spec_nth_asm_tac 3 ⌜y⌝)            (* Done! *);
```

which immediately completes the proof. What has happened is that the result, $\neg\ (\phi\ y \wedge \psi\ y)$, of specialising assumption 3 has been stripped into the assumptions automatically. *En route* this predicate will have been converted into $\neg\ \phi\ y \vee \neg\ \psi\ y$ causing a case split, each case then being an instance of a propositional axiom.

The style of predicate calculus proof discussed above gives a complete method for theorems of the predicate calculus (relying of course on human ingenuity to find witnesses). For simple examples such as the above, one would hope that the system would provide more automated assistance. A completely automatic approach to simple predicate calculus problems is provided by a simple resolution-based decision procedure, which we can access via a tactic interface called *prove_tac* or via a rule called *prove_rule*. The following is the one line proof of *10.51:

SML

```
val thm5 = save_thm("thm5",
    prove_rule[]⌜(∃x:V• φx ∧ ψx) ⇒ (∃y:V• φy) ∧ (∃z:V• ψz)⌝);
```

ProofPower Output

$$\begin{array}{l} \textit{val thm5} = \vdash (\exists\ x : V \bullet \phi\ x \land \psi\ x) \\ \quad \Rightarrow (\exists\ y : V \bullet \phi\ y) \land (\exists\ z : V \bullet \psi\ z) : \textit{THM} \end{array}$$

2.3 Z Expression Constructs

Support for the various expression constructs of the Z language is provided in ProofPower by tactics and rules appropriate to the construct in question. For example, the tactic *z_app_eq_tac* supports reasoning from first principles, so to speak, about equations involving function applications. Let's use it to begin a proof about application of a singleton function:

SML

```
set_goal([], ⓩ{(1, 2)} 1 = 2⌝);
a(z_app_eq_tac);
```

This results in the following goal:

ProofPower Output

$$\begin{array}{l} (*\ ?\vdash *)\ \ ⓩ(\forall\ f\_a : U \mid (\mathit{1}, f\_a) \in \{(\mathit{1}, \mathit{2})\} \bullet f\_a = \mathit{2}) \\ \qquad\qquad \land (\mathit{1}, \mathit{2}) \in \{(\mathit{1}, \mathit{2})\}⌝ \end{array}$$

I.e., to prove that $\{(\mathit{1}, \mathit{2})\}$ applied to $\mathit{1}$ gives $\mathit{2}$, we must show that $\{(\mathit{1}, \mathit{2})\}$ is single-valued at $\mathit{1}$ and contains the pair $(\mathit{1}, \mathit{2})$.

Many of the rather low-level facilities for the various expression constructs are integrated into more general purpose facilities such as stripping; users typically do not need to worry about the fine details. For example, stripping will complete the above proof, using rules such as $X \in \{Y\} \Leftrightarrow X = Y$ to eliminate the set display which appears in the goal.

The user can control the overall style of a proof step by selecting an appropriate *proof context*. Proof contexts are named packages of parameter settings for the general purpose rules and tactics. So far, we have been working in a proof context which is not particularly "aggressive" in its treatment of equations between sets. For example, consider the following beginning of a proof:

SML

```
set_goal([], ⓩ{x, y} = {1,2} ⇒ x ∈ {1,2,3}⌝);
a(strip_tac THEN strip_tac);
```

This does good a good job of eliminating the membership of the set display, but doesn't help much with the equation:

ProofPower Output

$$\begin{array}{l} (*\ \ \mathit{1}\ *)\ \ ⓩ\{x, y\} = \{\mathit{1}, \mathit{2}\}⌝ \\ \\ (*\ ?\vdash *)\ \ ⓩx = \mathit{1} \lor x = \mathit{2} \lor x = \mathit{3}⌝ \end{array}$$

If we undo the last step, set up a different proof context and try again, we get a little further:

SML

```
undo 1;
set_pc"z_language_ext";
a(strip_tac THEN strip_tac);
```

which results in:

ProofPower Output

$$(*\ \ 1\ *)\ \ {}^{\ulcorner}_{Z}\forall\ x1 : U \bullet x1 \in \{x, y\} \Leftrightarrow x1 \in \{1, 2\}^{\urcorner}$$

$$(*\ ?\vdash\ *)\ \ {}^{\ulcorner}_{Z}x = 1 \lor x = 2 \lor x = 3^{\urcorner}$$

Specialising the assumption to x, and further stripping will now complete the proof.

2.4 Equational Reasoning

The basic facts about many Z language constructs are actually provided as what are called conversions. A *conversion* is an inference rule coded as a function whose argument is an HOL term (e.g., a Z predicate or expression), say t, and whose result is a theorem of the form $\vdash t = t'$ or $\vdash t \Leftrightarrow t'$. For example, let's look at a conversion called *z_×_conv* that expands a cartesian product as a set comprehension:

SML

```
val thm8 = save_thm("thm8",  z_×_conv ⌜Z A × B × C⌝);
```

ProofPower Output

$$val\ thm8 = \vdash A \times B \times C = \{t_1 : A; t_2 : B; t_3 : C\} : THM$$

Conversions like the above one are provided mainly for the programmer rather than the user conducting an interactive proof. However, it can be convenient to have them to hand while doing proofs, particularly when a very fine degree of control is required. The system provides facilities for building new conversions from old, based on ideas originally due to Larry Paulson [9]. These facilities give a powerful general framework for programming equational reasoning.

Like most other LCF style systems, ProofPower provides a repertoire of general purpose term-rewriting tools, packaged as tactics, conversions and rules. These are the most common way of carrying out equational reasoning in an interactive proof. The user's input to these tools is a list of theorems providing equations for use as rewrite rules. Designing theorems which work together as useful rewrite systems is an important part of providing effective proof support for a body of definitions such as the toolkit.

Like so many other facilities in ProofPower, the rewriting tools are parameterised by the proof context. For example, many of the facts about Z language constructs which we saw handled by stripping in section 2.3 above, are also available by rewriting using an empty list of theorems:

SML

```
val thm9 = save_thm("thm9", rewrite_conv[] ⓩx ∈ {a, b, c}⌝);
```

ProofPower Output

$val\ thm9 = \vdash x \in \{a, b, c\} \Leftrightarrow x = a \lor x = b \lor x = c : THM$

3 Sets

The toolkit begins with the basic vocabulary of set theory, together with a few miscellanea. The operators in question are listed in the following table:

$\varnothing$	The empty set
$\neq$, $\notin$	negated equality, negated membership
$\cup$, $\cap$, $\setminus$	(binary) union, intersection and set difference
$\subseteq$, $\subset$	non-strict and strict inclusion
$\bigcup$, $\bigcap$	distributed union and intersection
first, *second*	First and second elements of a pair
$\mathbb{P}_1$	set of non-empty subsets of a set

Like many of the operators defined in the toolkit, these operators are generic. Here, for example, is the defining property of the union operator expressed as a ProofPower theorem (in a slightly simplified form):

$$
\begin{aligned}
&val\ thm10 = \\
&\quad \vdash ⓩ[X]\quad ((\_ \cup \_)[X] \in \mathbb{P}\ X \times \mathbb{P}\ X \to \mathbb{P}\ X \\
&\qquad \land\ (\forall\ S, T : \mathbb{P}\ X \bullet (\_ \cup \_)[X]\ (S, T) = \{x : X \mid x \in S \lor x \in T\}))
\end{aligned}
$$

Here, the "$[X]$" introducing the generic theorem acts as a universal quantifier, with X ranging over sets. The full name of the union operator is "$\_ \cup \_$" according to the Z conventions and, in principle, the meaning of "$(\_ \cup \_)[A]$", could depend crucially on the value of A. Now, in practice, no-one ever explicitly provides a generic actual parameter for an operator like union. Instead, one just writes things like "$\{1, 2\} \cup \{3,4\}$" and leaves the Z type checker to infer that what one means is "$(\_ \cup \_)[\mathbb{Z}]\ (\{1, 2\}, \{3, 4\})$". This is mostly harmless from a mathematical point of view, since the actual value of the generic parameter is immaterial for most operators providing it is big enough.

The universal quantification appearing inside the defining property for the union operator is slightly problematic in that it introduces a *conditional* rewrite rule. Before one can use the equation which is the core of the definition to rewrite a term of the form $(\_ \cup \_)[X]\ (A, B)$, one must establish that A and B

are indeed subsets of X. This is a general problem with equational reasoning in Z — a universally quantified equation will always give a conditional rewrite rule, although in many useful cases, the condition is mathematically, if not logically, trivial. In ProofPower, the situation is eased by the use of a special generic constant U, which is used to denote the set of all values of some type. For example, when a given set G is introduced its defining property is $G = U$. The rewriting tools are aware that a declaration of the form $v : U$ does not impose a condition when used in a universal quantification. So quantification over U allows us to express unconditional rewrite rules. Here, for example, is the U-instance of the defining property for union:

$$\begin{array}{l} \mathit{val\ thm11} = \\ \quad \vdash \quad (\_ \cup \_) \in \mathbb{P}\ U \times \mathbb{P}\ U \to \mathbb{P}\ U \\ \quad \land \quad (\forall\ S,\ T : \mathbb{P}\ U \bullet S \cup T = \{x : U \mid x \in S \lor x \in T\}) : \mathit{THM} \end{array}$$

(Note how the U-instance of the union operator is now written in the familiar infix form). This form of the defining property can now be used as a rewrite rule, as we may see by executing the following fragment of ML:

SML

```
val thm12 = rewrite_conv[thm11] ⓩ{1, 2} ∪ {3,4}⌝;
```

ProofPower Output

$$\begin{array}{l} \mathit{val\ thm12} = \vdash \{1,\ 2\} \cup \{3,\ 4\} = \\ \qquad \{x : U \mid (x = 1 \lor x = 2) \lor x = 3 \lor x = 4\} : \mathit{THM} \end{array}$$

The first step in handling a typical body of generic definitions in Z is to derive rewrite rules to expand away those definitions, at least in the most common cases. As we have seen, in ProofPower, this can typically be done by proving the U-instances of the definitions. However, one finds that blindly expanding definitions is rarely the preferred method of proof in practice. What one must do is identify the forms of conjecture which occur most often and provide methods exploiting domain-specific information to help with those forms. For the elementary theory of sets, it is natural to focus on conjectures of the form $t \in A$, $A = B$, $A \subseteq B$, and $A \subset B$, where A and B are expressions formed using the vocabulary of set theory. Proof contexts are provided to help with problems of this type. These are based on theorems such as the following:

$$\vdash \forall\ z : U;\ s : U;\ t : U \bullet z \in s \cup t \Leftrightarrow z \in s \lor z \in t$$

$$\vdash \forall\ z : U;\ s : U;\ t : U \bullet z \in s \cap t \Leftrightarrow z \in s \land z \in t$$

$$\begin{array}{l} \vdash \forall\ a : U \bullet\ a \cup \{\} = a \land \{\} \cup a = a \land \\ \qquad a \cup U = U \land U \cup a = U \land a \cup a = a \end{array}$$

$$\begin{array}{l} \vdash \forall\ a : U \bullet\ a \cap \{\} = \{\} \land \{\} \cap a = \{\} \land \\ \qquad a \cap U = a \land U \cap a = a \land a \cap a = a \end{array}$$

The styles of reasoning supported are a so-called algebraic style, in which set notation is only traded in for logic when the form of the problem strongly suggests doing so, and a so-called extensional style, in which equations and other relations between sets are aggressively converted into logic. The extensional style may be seen in the following proof:

SML

```
set_goal([], ⓩ(A ∪ B) = (A \ B) ∪ (A ∩ B) ∪ (B \ A)⌝);
set_pc"z_sets_ext";
a(rewrite_tac[]);
```

This produces the following goal:

ProofPower Output

```
(* ?⊢ *)  ⓩ∀ x1 : U
              • x1 ∈ A ∨ x1 ∈ B
              ⇔ x1 ∈ A ∧ ¬ x1 ∈ B ∨
              x1 ∈ A ∧ x1 ∈ B ∨
              x1 ∈ B ∧ ¬ x1 ∈ A⌝
```

Note how all the set-theoretic vocabulary has been eliminated, resulting in a predicate calculus conjecture (which can easily be solved by stripping). In fact, the extensional proof context provides a decision procedure for a useful class of set-theoretic trivia, and when such are needed (e.g., as lemmas to help in a more complex proof), they can readily be proved automatically. For example, if the above lemma were required, one might use:

SML

```
val thm14 = save_thm("thm14", pc_rule1 "z_sets_ext" prove_rule[]
    ⓩ(A ∪ B) = (A \ B) ∪ (A ∩ B) ∪ (B \ A)⌝);
```

which uses the proof context to carry out an automatic proof resulting in:

ProofPower Output

```
val thm14 = ⊢ A ∪ B = (A \ B) ∪ A ∩ B ∪ B \ A : THM
```

The algebraic style of reasoning is the preferred style when one wants to retain the vocabulary of set theory during a proof. This paradigm is very common when one is building up a theory. First of all, to develop the basic facts of the theory, one needs methods like the extensional style for set theory, which eliminate the vocabulary of the theory in favour of more primitive notions. When one moves on to more complex theorems either within the theory itself or in building another theory, one generally wants to preserve the vocabulary of the theory and so conduct proofs at a higher conceptual level.

Natural examples of the algebraic method would take up too much space in the present document. By way of a contrived example, let us prove a theorem which one feels ought to be an easy consequence of *thm14* above. First of all, let us continue to work with the extensional proof context:

SML

```
set_goal([], ⓩA ∩ B = {} ⇒ (A ∪ B) = (A \ B) ∪ (B \ A)⌝);
a(REPEAT strip_tac);
```

ProofPower Output

```
(* *** Goal "2" *** *)

(*  2 *)  ⓩ∀ x1 : U • x1 ∈ A ∩ B ⇔ x1 ∈ {}⌝
(*  1 *)  ⓩx1 ∈ B⌝

(* ?⊢ *)  ⓩ¬ x1 ∈ A⌝


(* *** Goal "1" *** *)

(*  3 *)  ⓩ∀ x1 : U • x1 ∈ A ∩ B ⇔ x1 ∈ {}⌝
(*  2 *)  ⓩx1 ∈ A⌝
(*  1 *)  ⓩx1 ∈ B⌝

(* ?⊢ *)  ⓩ¬ x1 ∈ A⌝
```

Now these subgoals can readily be proved, but our hope of using *thm14* above to help would have to be abandoned. Let's go back and work in the algebraic proof context instead:

SML

```
undo 1;
set_pc"z_sets_alg";
a(REPEAT strip_tac);
```

ProofPower Output

```
(*  1 *)  ⓩA ∩ B = {}⌝

(* ?⊢ *)  ⓩA ∪ B = (A \ B) ∪ B \ A⌝
```

We should now be able to use *thm14* as a rewrite rule to progress the proof:

SML

```
a(rewrite_tac[thm14]);
```

ProofPower Output

```
(*  1 *)  ⓩA ∩ B = {}⌝

(* ?⊢ *)  ⓩ(A \ B) ∪ A ∩ B ∪ B \ A = (A \ B) ∪ B \ A⌝
```

Rewriting with the assumptions and the algebraic laws about sets embodied in the proof context now completes the proof:

SML

```
a(asm_rewrite_tac[])            (* Done! *);
```

4 Relations

This section of the toolkit deals with binary relations represented as sets of pairs. The operators introduced are listed in the following table (in which operands R, X, Y are provided as needed to suggest the syntax of the operator):

$\mapsto$	maplet (a variant syntax for pairing: $x \mapsto y = (x, y)$)
$X \leftrightarrow Y$	set of all relations between X and Y
dom, ran	domain and range of a relation
id	identity relation
o, $\fatsemi$	right-first and left-first relational composition
$\lhd$, $\rhd$	range and domain restriction
$\ntriangleleft$, $\ntriangleright$	range and domain anti-restriction
$\oplus$	relational overriding
$R^{\sim}$	relational inverse
$R(\!\mid X \mid\!)$	relational image
R^{+}, R^{*}	transitive and reflexive-transitive closure

The treatment of the relational operators in ProofPower follows a similar strategy to the treatment of the set-theoretic operators described in the previous section. First of all, theorems are proved which enable the relational vocabulary to be eliminated in favour of set theory and logic. In conjunction with the support for set-theoretic operators, these theorems are used to provide an extensional proof context, which will prove many of the basic facts about relations automatically and will assist in semi-automatic proof of less tractable problems. As with sets a less expansionist, so-called algebraic proof context is also provided.

As an example of the extensional method for relations, let's prove a simple fact about the two relational composition operators:

SML

```
set_pc"z_rel_ext";
set_goal([], ⓩR ⨾ S = S o R⌝);
a(strip_tac);
```

This produces the following subgoal:

ProofPower Output

```
(* ?⊢ *)  ⓩ∀ x1 : U; x2 : U • (x1, x2) ∈ R ⨾ S ⇔ (x1, x2) ∈ S o R⌝
```

Here extensionality has been used to eliminate the equality of two relations in favour of a universally quantified membership assertions. Note how the system has introduced a pair $(x1, x2)$ of variables rather than a single variable; introduction of a single variable ranging over pairs turns out to lead one down blind alleys in many cases, so the pair of variables is much preferred. Let us proceed a bit further with the proof:

SML
```
a(REPEAT_N 6 strip_tac)            (* strip 6 times *);
```

At this stage we have a goal split:

ProofPower Output
```
(* *** Goal "2" *** *)

(* ?⊢ *)  ⌜(x1, x2) ∈ S o R ⇒ (x1, x2) ∈ R ⨾ S⌝


(* *** Goal "1" *** *)

(* ?⊢ *)  ⌜(x1, x2) ∈ R ⨾ S ⇒ (x1, x2) ∈ S o R⌝
```

The next step of stripping will effectively expand the definition of ⨾ and transfer the resulting information to the assumptions.

SML
```
a(strip_tac);
```

ProofPower Output
```
(* *** Goal "1" *** *)

(*  2 *)  ⌜(x1, y) ∈ R⌝
(*  1 *)  ⌜(y, x2) ∈ S⌝

(* ?⊢ *)  ⌜(x1, x2) ∈ S o R⌝
```

Another stripping step will expand the definition of o and so reduce the problem to pure logic (in the sense that, while the goal still contains pairing and membership, the meanings of those operators no longer affect the truth of the goal):

SML
```
a(strip_tac);
```

ProofPower Output

```
(* *** Goal "1" *** *)

(*  2 *)  ⓩ(x1, y) ∈ R⌝
(*  1 *)  ⓩ(y, x2) ∈ S⌝

(* ?⊢ *)  ⓩ∃ y : U • (x1, y) ∈ R ∧ (y, x2) ∈ S⌝
```

Supplying the obvious witness y and stripping will now complete this branch of the proof:

SML

```
a(z_∃_tacⓩy⌝ THEN REPEAT strip_tac);
```

The other branch of the proof is very similar; but, there is an easier way:

SML

```
a(prove_tac[])                (* Done! *);
```

In fact, the resolution-based proof procedure, when used in an appropriate proof context, can carry out a completely automatic proof of a wide class of theorems similar to the one we have just proved:

SML

```
val thm17 = save_thm("thm17", prove_rule[]ⓩ R ⨾ S = S o R ⌝);
```

ProofPower Output

```
val thm17 = ⊢ R ⨾ S = S o R : THM
```

The methods outlined above work well for everything in this section of the toolkit except for the closure operators. The inductive nature of the definition of the two closure operators necessarily complicates all but the most trivial reasoning about them. Support for these operators is provided in the shape of theorems which make it easy to expand their definitions. These have been found adequate to date, although it must be pointed out that our applications of Z have not often made extensive use of these operators.

5 Functions

This section of the toolkit deals with functions viewed as a special case of binary relations. The only operators introduced are the function space arrows that are such a distinctive feature of the Z notation. The arrows provided are shown in the following table:

$X \rightarrowtail\!\!\!\!\!\!+\ Y$	set of all partial functions from X to Y
$X \rightarrow Y$	set of all total functions from X to Y
$X \rightarrowtail\!\!\!\!\!+ Y$	set of all partial injections of X into Y
$X \rightarrowtail Y$	set of all total injections of X into Y
$X \twoheadrightarrow\!\!\!\!\!\!+\ Y$	set of all partial surjections of X onto Y
$X \twoheadrightarrow Y$	set of all total surjections of X onto Y
$X \rightarrowtail\!\!\!\!\!\twoheadrightarrow Y$	set of all bijections between X and Y

An important role played by the function arrows is in justifying reasoning about function application. Very often the fact that an object belongs to some function space is given to us by virtue of the declaration introducing the object. We need convenient means of exploiting this feature of the usual Z style.

In section 2.3, we have already done some reasoning about a function application from first principles using the tactic *z_app_eq_tac*. To reason about a function application $f\ x$, this tactic requires us to verify a a direct expression in logic of the fact that f is functional at x. In principle, we can always appeal to the definitions of the arrows to deduce the theorems required by the tactic. However, in practice it is tedious and verbose to do so.

A family of theorems is provided to help exploit assumptions concerning membership of function spaces. These work well with a technique called forward chaining. Here's an example:

SML

```
set_goal([], ⌜Z f ∈ X → {y} ∧ x ∈ X ⇒ f x = y ⌝);
a(REPEAT strip_tac);
```

This gives us the following subgoal:

ProofPower Output

```
(*  2 *)  ⌜Z f ∈ X → {y}⌝
(*  1 *)  ⌜Z x ∈ X⌝

(* ?⊢ *)  ⌜Z f x = y⌝
```

To attack this, we will use the theorem *z_fun_∈_clauses*, which is the following:

```
⊢ ∀ f : U; x : U; X : U; Y : U
  • ((f ∈ X → Y ∨ f ∈ X ↣ Y ∨ f ∈ X ↠ Y ∨ f ∈ X ⤖ Y) ∧
     x ∈ X ⇒ f x ∈ Y)
  ∧ ((f ∈ X ⇸ Y ∨ f ∈ X ⤔ Y ∨ f ∈ X ⤀ Y) ∧
     x ∈ dom f ⇒ f x ∈ Y)
```

This logically entails a number of implications:

```
⊢ ∀ X : U; Y : U; f : U; x : U • f ∈ X → Y ⇒ x ∈ X ⇒ f x ∈ Y
⊢ ∀ X : U; Y : U; f : U; x : U • f ∈ X ↣ Y ⇒ x ∈ X ⇒ f x ∈ Y
    ...
```

Forward chaining is a technique which takes implicative theorems such as the above, and searches in the assumptions for facts which match the antecedents of the implication. A successful match gives rise to a consequence which is an instance of the succedent of the implication. In this case, $f \in X \to Y$ matches the first assumption (with Y instantiated to $\{y\}$) and $x \in X$ matches the second assumption (as is). The form of forward chaining tactic we will now use strips any consequences it is able to derive into the list of assumptions. In this case that completes the proof.

SML

```
a(all_asm_fc_tac[z_fun_∈_clauses])              (∗ Done! ∗);
```

To see this in slow motion let us go back a step and use a variant of forward chaining which puts the derived consequences as antecedents of an implication in the conclusion of the goal:

SML

```
undo 1;
a(ALL_ASM_FC_T (MAP_EVERY ante_tac) [z_fun_∈_clauses]);
```

This gives:

ProofPower Output

```
(∗  2 ∗)  ⌜Z f ∈ X → {y}⌝
(∗  1 ∗)  ⌜Z x ∈ X⌝

(∗ ?⊢ ∗)  ⌜Z f x ∈ {y} ⇒ f x = y⌝
```

Stripping $f\ x \in \{y\}$ into the assumptions will now transform it into $f\ x = y$ *en route*, and so turn the goal into a tautology.

A body of theorems along the same lines as *z_fun_∈_clauses* has long proved useful in many situations where reasoning about function application is required in practice. Some types of reasoning covered by these theorems is amenable to greater automation. In particular, a prolog-like backwards-chaining algorithm can prove many useful conjectures of the form $e{\in}A$ by analysis of the function applications and other language constructs which make up e. An algorithm of this type has been implemented but is not yet on general release. Such an algorithm has the potential for automating many proofs that the arguments of functions lie in their domains. This is not dissimilar to the role of the so-called type inference process in the Boyer-Moore theorem prover [2]. A limited interface to the algorithm is made available in the Compliance Tool, where it is used to automate proofs that the Z translations of Ada expressions belong to the Z sets which model the corresponding Ada type. The algorithm and its interfaces have not yet been tuned and tested for general purpose use.

As part of building up the theory of finiteness we have begun to develop an approach to two issues with the function arrows. The first issue is that the total function arrows other than $\to$ itself are defined in terms of the partial function arrows. For example, the defining property of $\rightarrowtail$ is:

$\vdash [X,\ Y](X \rightarrowtail Y = (X \mapsto\!\!\!\!\!\!\rightarrowtail Y) \cap (X \rightarrow Y))$

In proofs, it is generally easier to work with total functions rather than partial ones (and one always can, since any function is total on its domain).

The second issue is that within the definitions of some of the functional properties such as injectivity, the toolkit tends to use a mixture of the idioms $x \mapsto y \in f$ or $(x,\ y) \in f$ or $f\ x = y$. For example, here are some of the defining properties:

$$
\begin{array}{l}
\vdash [X,\ Y](X \rightarrow Y = \{f : X \leftrightarrow Y \mid \forall\ x : X \bullet \exists_1\ y : Y \bullet (x,\ y) \in f\}) \\
\vdash [X,\ Y](X \nrightarrow Y = \\
\quad \{f : X \leftrightarrow Y \mid \\
\qquad \forall\ x : X;\ y1,\ y2 : Y \bullet x \mapsto y1 \in f \land x \mapsto y2 \in f \Rightarrow y1 = y2\}) \\
\vdash [X,\ Y](X \mapsto\!\!\!\!\!\!\rightarrowtail Y = \{f : X \nrightarrow Y \mid \\
\qquad \forall\ x1,\ x2 : dom\ f \bullet f\ x1 = f\ x2 \Rightarrow x1 = x2\})
\end{array}
$$

The idiom involving the function application tends to lead to some unnecessary work. Theorems are provided for use as rewrite rules serving: *(a)* to eliminate partial arrows in favour of total ones; *(b)* to simplify total arrows to the ordinary total arrow together with a succinct statement of any extra information in the arrow; and *(c)* to avoid unnecessary uses of function application. Here, for example, are some of them:

ProofPower Output

$$
\begin{array}{l}
\vdash \forall\ A : U;\ B : U;\ f : U \bullet f \in A \nrightarrow B \Leftrightarrow f \in dom\ f \rightarrow B \land dom\ f \subseteq A \\
\vdash \forall\ A : U;\ B : U;\ f : U \bullet \\
\qquad f \in A \rightarrowtail B \\
\quad \Leftrightarrow f \in A \rightarrow B \\
\qquad \land (\forall\ x,\ y : U;\ z : U \bullet (x,\ z) \in f \land (y,\ z) \in f \Rightarrow x = y) \\
\vdash \forall\ A : U;\ B : U;\ f : U \bullet f \in A \twoheadrightarrow B \Leftrightarrow f \in A \rightarrow B \land B \subseteq ran\ f
\end{array}
$$

This latter approach has not yet been worked through comprehensively, but enough has been done to facilitate derivation of theorems such as the following composition rules for bijectivity:

$$
\begin{array}{l}
\vdash \forall\ A : U;\ B : U;\ C : U;\ f : U;\ g : U \\
\quad \bullet f \in A \rightarrowtail\!\!\!\!\!\twoheadrightarrow B \land g \in B \rightarrowtail\!\!\!\!\!\twoheadrightarrow C \Rightarrow g\ o\ f \in A \rightarrowtail\!\!\!\!\!\twoheadrightarrow C \\
\vdash \forall\ A : U;\ B : U;\ C : U;\ D : U;\ f : U;\ g : U \\
\quad \bullet f \in A \rightarrowtail\!\!\!\!\!\twoheadrightarrow B \land g \in C \rightarrowtail\!\!\!\!\!\twoheadrightarrow D \land A \cap C = \{\} \land B \cap D = \{\} \Rightarrow \\
\qquad f \cup g \in A \cup C \rightarrowtail\!\!\!\!\!\twoheadrightarrow B \cup D
\end{array}
$$

Theorems like these are required in developing the theory of finiteness and are surprisingly tedious to establish without a systematic approach. For the second of the two, blind expansion of definitions would produce 48 distinct subgoals.

6 Arithmetic

This section of the toolkit provides the sets and operators for integer arithmetic listed in the following table:

$\mathbb{Z}$, $\mathbb{N}$, $\mathbb{N}_1$	The sets of integers, naturals, and positive naturals
$\sim$, $+$, $-$	arithmetic negation, addition, subtraction
$*$, *div*, *mod*	multiplication, division and modulus
$<$, $\leq$, $>$, $\geq$	ordering relations
abs	absolute value
succ	successor function
R^i	relational iteration
i .. j	integer range (or interval)

The theory of these operators is sufficiently complex to merit some internal structure in its description. We shall describe it in sections 6.1 to 6.7 below under the following headings: *The Definitions, Additive Structure, Multiplicative Structure, Division and Modulus, Computation, Linear Arithmetic, Other Operators.*

6.1 The Definitions

A first difficulty with this section of the toolkit is that defining properties for the basic arithmetic operators are not given in [12]. It is clear what the operators are intended to be (since [12] does give laws that fix the meaning of *div* and *mod* and there's not much doubt about the rest). However, for our purposes the omitted definitions must be supplied.

The defining properties chosen for use in ProofPower are fairly succinct, work well in practice and can be easily related to appropriate mathematical theory. A treatment closely based on this approach is currently tabled for inclusion in the evolving standard for Z.

The following four properties are used to characterise the additive structure of the integers.

$$
\begin{array}{ll}
\vdash & \forall\, i, j, k : \mathbb{Z} \bullet \\
& \quad (i + j) + k = i + j + k \\
& \wedge \quad i + j = j + i \\
& \wedge \quad i + \sim i = 0 \\
& \wedge \quad i + 0 = i \\
\vdash & \forall\, h : \mathbb{P}\, \mathbb{Z} \bullet 1 \in h \wedge (\forall\, i, j : h \bullet i + j \in h \wedge \sim i \in h) \Rightarrow h = \mathbb{Z}) \\
\vdash & \mathbb{N} = \bigcap \{s : \mathbb{P}\, \mathbb{Z} \mid 0 \in s \wedge \{i : s \bullet i + 1\} \subseteq s\} \\
\vdash & \sim 1 \notin \mathbb{N} : \mathit{THM}
\end{array}
$$

The first of these properties says that the integers form a group under addition; the second says that any subgroup h of the integers containing 1 is equal to the whole group; the third says that the natural numbers form the smallest set containing 0 and closed under addition of 1; the final property says that ~ 1 is not a natural number. Together these properties imply that the additive group of the integers is an infinite cyclic group generated by 1; this condition completely characterises the additive structure of the integers (see, e.g., [3] for a proof of this characterisation).

Note in connection with the first of these properties that the binary operators in the ProofPower dialect of Z are right-associative. I.e., $1 + 2 + 3$ means $1 + (2 + 3)$ not $(1 + 2) + 3$. It has been recognised that this departure from the traditional view is inappropriate and we hope for later versions of ProofPower to allow both left- and right-associative operators.

Subtraction is defined in terms of addition and negation:

$$\vdash \forall\ i, j : \mathbb{Z} \bullet i - j = i + \sim j$$

The ordering relations for the integers can be characterised in terms of the addition, subtraction and $\mathbb{N}$. The definitions used in ProofPower define $\leq$ in this way and then define the other forms in terms of $\leq$:

$$\begin{array}{ll}
\vdash \forall i, j : \mathbb{Z} \bullet & \\
& (i \leq j \Leftrightarrow j - i \in \mathbb{N}) \\
\land & (i < j \Leftrightarrow i + 1 \leq j) \\
\land & (i \geq j \Leftrightarrow j \leq i) \\
\land & (i > j \Leftrightarrow j < i))
\end{array}$$

The multiplicative structure of the integers is then characterised as follows:

$$\begin{array}{ll}
\vdash \forall\ i, j, k : \mathbb{Z} \bullet & \\
& (i * j) * k = i * j * k \\
\land & i * j = j * i \\
\land & i * (j + k) = i * j + i * k \\
\land & 1 * i = i
\end{array}$$

This says that multiplication is the (necessarily unique) operation which together with addition makes the integers into a commutative ring with unit element 1.

Space prevents us listing the defining properties for the remaining operators. However, there is plenty of interest in developing the theory of what we have already described and particular points about the other operators will be mentioned as they are encountered in the sequel.

6.2 Additive Structure

Under the additive structure of the integers we include the sets $\mathbb{N}$ and $\mathbb{N}_1$, the operators, $+$, $-$ and $\sim$ and the ordering relations, $\leq$, $<$, $>$ and $\geq$.

The main objective is to develop support for *(a)* induction and *(b)* reasoning by cancellation of like terms. Of these two, cancellation laws (and convenient ways of using them) are actually much more widely applicable than induction. However both are important as are sundry other methods (e.g., using order-theoretic properties of the ordering relations).

The first step is to provide, as theorems for use as rewrite rules, the associative law for addition. ProofPower treats an equation as a rewrite rule in the left-to-right direction. By providing the following two theorems (*z_plus_assoc_thm* and *z_plus_assoc_thm1*), we are providing the commands: "Brackets to the right!" and "Brackets to the left!", as a by-product of standard conventions of the system:

$$\vdash \forall\ i, j, k : U \bullet (i + j) + k = i + j + k$$
$$\vdash \forall\ i, j, k : U \bullet i + j + k = (i + j) + k$$

(Here $U = \mathbb{Z}$; recent versions of the system would allow us equally well to use $\mathbb{Z}$ instead of U here, but for historical reasons the actual theorems provided are stated using U).

To show the associativity theorems in action, I have asked the system to display more brackets than are strictly necessary to make clear what is going on:

SML

```
val thm19 = save_thm("thm19",
    rewrite_conv[z_plus_assoc_thm] ⌜Z(a + b) + (i + j)⌝);
val thm20 = save_thm("thm20",
    rewrite_conv[z_plus_assoc_thm1] ⌜Z(a + b) + (i + j)⌝);
```

ProofPower Output

```
val thm19 = ⊢ ((a + b) + (i + j)) = (a + (b + (i + j))) : THM
val thm20 = ⊢ ((a + b) + (i + j)) = (((a + b) + i) + j) : THM
```

The next issue to address is the commutative law. The problem with commutativity is that the rewrite rule $x + y = y + x$ will necessarily loop indefinitely. The rewriting tools in ProofPower make two tiny improvements over their predecessors in related systems: firstly, if a term is unchanged by a rewrite, then the rewrite is rejected; secondly, and more importantly here, free variables in a theorem supplied by a rewrite rule are not taken as candidates for instantiation. This latter feature underlies the surprising utility of the following theorem called *z_plus_order_thm*:

$$\begin{aligned}
&\vdash \forall\ i : U \\
&\quad \bullet \forall\ j, k : U \\
&\qquad \bullet ((j + i) = (i + j)) \\
&\qquad\quad \land ((((i + j) + k) = (i + (j + k))) \\
&\qquad\quad \land ((j + (i + k)) = (i + (j + k))))
\end{aligned}$$

The idea here is that if i is specialised to some expression, t, of interest, then rewriting with *z_plus_order_thm* has the very useful effect of making t float round to the beginning of any expression formed using addition in which it is contained:

SML

```
val thm21 = save_thm("thm21",
    z_∀_elim⌜Z99⌝ z_plus_order_thm);
val thm22 = save_thm("thm22",
    rewrite_conv[thm21] ⌜Z(n + m + 99) + m⌝);
```

ProofPowerOutput

```
val thm21 = ⊢ 99 ∈ U ∧ true
    ⇒ (∀ j, k : U
      • j + 99 = 99 + j
        ∧ (99 + j) + k = 99 + j + k
        ∧ j + 99 + k = 99 + j + k) : THM
val thm22 = ⊢ (n + m + 99) + m = 99 + (n + m) + m : THM
```

Note that the rewriting tools preprocess the theorems supplied as arguments in an attempt to squeeze out as many equations as possible. In this case, they will automatically eliminate the vacuous antecedent to the implication in *thm21* and then extract the three equations that remain.

Combined with appropriate cancellation rules for equality and the ordering relations, this rather simple little device for using commutativity and associativity turns out to be effective for proving many of the basic facts about addition one needs. Here is the cancellation rule for equality (*z_plus_clauses*):

```
⊢ ∀ i, j, k : U •
          (i + k = j + k ⇔ i = j)
    ∧     (k + i = j + k ⇔ i = j)
    ∧     (i + k = k + j ⇔ i = j)
    ∧     (k + i = k + j ⇔ i = j)
    ∧     (i + k = k ⇔ i = 0)
    ∧     (k + i = k ⇔ i = 0)
    ∧     (k = k + j ⇔ j = 0)
    ∧     (k = j + k ⇔ j = 0)
    ∧     i + 0 = i
    ∧     0 + i = i
    ∧     ¬ 1 = 0
    ∧     ¬ 0 = 1
```

There are similar rules for the ordering relations.

Subtraction is systematically catered for by using its definition in terms of addition and negation ($x - y = x + \sim y$). Arithmetic negation is easily handled in the additive fragment of the theory by one or two trivial consequences of the definitions such as:

$$\vdash \forall\, i : U \bullet i + \sim i = 0 \wedge \sim i + i = 0$$

To see these simple but effective ideas at work, here is an example adapted from the depths of a proof about the size of a range of integers. For clarity, we have set the system up to display a profusion of brackets in the proof.

SML

```
set_goal([],
    ⌜a + ((x1 + i) + (~ 1)) ≤ a + (j + (((~ i) + 1) + (i + (~ 1))))
        ⇒  (x1 + (i + (~ 1))) ≤ j⌝);
a(rewrite_tac[z_≤_clauses]);
```

Note how standard algebraic collection and cancellation of like terms here makes it clear that the predicates on either side of $\Rightarrow$ here are the same. Our approach to the proof is to rewrite both sides of the implication until they become equal. We have already begun to do this by cancelling the two instances of a:

ProofPower Output

```
(* ?⊢ *)  ⌜(((x1 + i) + (~ 1)) ≤ (j + (((~ i) + 1) + (i + (~ 1)))))
             ⇒ ((x1 + (i + (~ 1))) ≤ j)⌝
```

Moving instances of i to the fore here should give a chance for the 1 and its negation to cancel out:

SML

```
a(rewrite_tac[z_∀_elim⌜i⌝z_plus_order_thm]);
```

ProofPower Output

```
(* ?⊢ *)  ⌜((i + (x1 + (~ 1))) ≤ (i + (j + (((~ i) + 1) + (~ 1)))))
             ⇒ ((i + (x1 + (~ 1))) ≤ j)⌝
```

Now let's make the cancellation happen:

SML

```
a(rewrite_tac[z_plus_assoc_thm, z_plus_minus_thm, z_plus0_thm]);
```

ProofPower Output

```
(* ?⊢ *)  ⌜((i + (x1 + (~ 1))) ≤ (i + (j + (~ i))))
    ⇒ ((i + (x1 + (~ 1))) ≤ j)⌝
```

Finally, let's bring j to the front so that the second i can cancel out with its negation. We can do this in one step:

SML

```
a(rewrite_tac[z_∀_elim⌜Z j⌝
    z_plus_order_thm,z_plus_minus_thm, z_plus0_thm ])
                                            (* Done! *);
```

This makes both sides of the implication the same and completes the proof.

A number of induction tactics are provided. These are implemented by a uniform mechanism which takes as input a theorem stating the induction principle. These theorems are, by convention, given in a mixture of HOL and Z. Here for example is a rather unusual induction theorem which says that if a property holds of *1*, and is preserved under negation and addition, then it holds for all integers m.

```
⊢ ⌜∀ p• p ⌜Z 1⌝ ∧ (∀ i• p i ⇒ p ⌜Z ∼ i⌝) ∧ (∀ i j• p i ∧ p j ⇒ p ⌜Z i + j⌝)
     ⇒ (∀ m• p m)⌝
```

Here the symbols ⌜ and ⌝ delimit an HOL term, within which embedded fragments of Z are delimited by ⌜Z and ⌝.

The induction principle that has been found most widely useful is called *z_≤_induction_tac*. Let's see it in action:

SML

```
set_goal([],
    ⌜Z f k = k ∧ (∀i:U• f(i + 1) = f i + 1) ⇒ (∀j:U• k ≤ j ⇒ f j = j)⌝);
a(REPEAT strip_tac);
```

ProofPower Output

```
(*  3 *)  ⌜Z f k = k⌝
(*  2 *)  ⌜Z ∀ i : U • f (i + 1) = f i + 1⌝
(*  1 *)  ⌜Z k ≤ j⌝

(* ?⊢ *)  ⌜Z f j = j⌝
```

The induction tactic takes as an argument the induction variable, in this case j. It expects a term of the form $x \leq j$ in the assumptions; in this case x is k.

SML

```
a(z_≤_induction_tac ⌜Z j⌝);
```

This gives us two subgoals; the first one is the base case ($j = k$):

ProofPower Output

```
...
(*  2 *)  ⌜Z f k = k⌝
(*  1 *)  ⌜Z ∀ i : U • f (i + 1) = f i + 1⌝

(* ?⊢ *)  ⌜Z f k = k⌝
```

This is a propositional tautology:

SML

```
a(REPEAT strip_tac);
```

ProofPower Output

```
Tactic produced 0 subgoals:
Current goal achieved, next goal is:

(* *** Goal "2" *** *)

(*  4 *)  ⌜f k = k⌝
(*  3 *)  ⌜∀ i : U • f (i + 1) = f i + 1⌝
(*  2 *)  ⌜k ≤ i⌝
(*  1 *)  ⌜f i = i⌝

(* ?⊢ *)  ⌜f (i + 1) = i + 1⌝
```

The proof of the inductive step is completed by rewriting with the assumptions:

SML

```
a(asm_rewrite_tac[])            (* Done! *);
```

In addition to the above material, other types of theorem are also useful, e.g., to justify various sorts of case analysis. Also theorems expressing order-theoretic properties of $<$, $\leq$ etc. are useful. Space prevents us giving detailed examples here , but some typical theorems of these sorts can be exhibited:

$$\vdash \forall\ i : \mathbb{N} \bullet i = 0 \lor (\exists\ j : \mathbb{N} \bullet i = j + 1)$$
$$\vdash \forall\ i, j : U \bullet i \leq j \lor j \leq i$$
$$\vdash \forall\ i, j : U \bullet i < j \lor i = j \lor j < i$$
$$\vdash \forall\ i, j, k : U \mid i < j \land j < k \bullet i < k$$

6.3 Multiplicative Structure

Theorems for working with the associative and commutative properties of multiplication are provided in close analogy to those for addition. Use of these and the distributive law supports a style of proof mimicking the process of “multiplying out and cancelling like terms”. For example:

SML

```
set_goal([], ⌜(x + y) * (x  + y) + ~(x*y) = x*x + x*y + y*y⌝);
a(rewrite_tac[z_times_plus_distrib_thm]);
```

ProofPower Output

$$
\begin{array}{l}
(*\ ?\vdash\ *)\ \ulcorner_Z((x * x + y * x) + x * y + y * y) + \sim (x * y)\\
\qquad\qquad = x * x + x * y + y * y\urcorner
\end{array}
$$

Let us carry out a first stage of additive cancellation:

SML

```
a(rewrite_tac[z_plus_assoc_thm, z_∀_elim⌜Zy*y⌝z_plus_order_thm,
        z_plus_minus_thm, z_plus_clauses]);
```

ProofPower Output

$$(*\ ?\vdash\ *)\ \ulcorner_Z y * x\ = x * y\urcorner$$

We now bring x say to the front of each of the two products here to show that they are the same and so complete the proof:

SML

```
a(rewrite_tac[z_∀_elim⌜Zx⌝z_times_order_thm])          (* Done! *);
```

6.4 Division and Modulus

For historical reasons, ProofPower has a rather idiosyncratic formulation of the operators *div* and *mod*. This formulation makes the value of *mod* always non-negative. This definition has some good mathematical properties, but disagrees with [12]. On balance the formulation in [12] seems to be the one of choice, and ProofPower may well be amended in this respect in a future release.

In any case, the type of theorems one wants about *div* and *mod* are theorems which characterise them in terms of addition and multiplication. The following two theorems have proved very useful (e.g., in developing the theory of the Ada formulation of division and modulus for use in program verification).

$$
\begin{array}{l}
\vdash \forall\ i, j, k : \mathbb{Z}\\
\quad \mid \neg\ j = 0\\
\quad \bullet\ i\ \mathit{div}\ j = k \Leftrightarrow (\exists\ m : \mathbb{Z} \bullet i = k * j + m \land 0 \le m \land m < \mathit{abs}\ j)\\
\vdash \forall\ i, j, k : \mathbb{Z}\\
\quad \mid \neg\ j = 0\\
\quad \bullet\ i\ \mathit{mod}\ j = k \Leftrightarrow (\exists\ d : \mathbb{Z} \bullet i = d * j + k \land 0 \le k \land k < \mathit{abs}\ j)
\end{array}
$$

6.5 Computation

Numerical computation turns out to be tedious or impossible in surprisingly many approaches to automated theorem-proving. Such a position is quite unacceptable both in doing mathematics and in applying a theorem prover to real-world problems. ProofPower provides conversions to carry out evaluation of expressions formed from numeric literals and the arithmetic operators. To demonstrate these, we use the proof context *z_libraries*. This proof context provides a convenient packaging for these computational conversions and also gives an interface to many of the other techniques described in this document. Here it is in action:

SML

```
set_pc"z_library";
val thm26 = save_thm("thm26",
    rewrite_conv[]⌜Z(7*11*13 − 1) * (2000 div 2)⌝);
```

ProofPower Output

```
val thm26 = ⊢ (7 * 11 * 13 − 1) * 2000 div 2 = 1000000 : THM
```

The computational conversions which carry out the evaluation here are, perhaps surprisingly, not oracles, i.e., the conversions derive their results by inference rather than brashly asserting them without any verification. Indeed ProofPower provides no facility for introducing oracles. The built-in rule of HOL on which the conversions are based carries out addition of natural number literals. This rule, which is the only oracle of its sort in the system, is used to implement the usual arithmetic operations for the natural numbers in HOL. These are used in turn to develop computational conversions for a theory of integers in HOL and the Z versions are then implemented via theorems establishing an isomorphism between the HOL integers and the Z ones. The above calculation actually involves some 547 primitive inference steps of which 69 are appeals to the built-in rule for natural number addition. Nonetheless, the performance of this approach seems to be adequate for all the purposes to which it has so far been put.

6.6 Linear Arithmetic

The facilities for semi-automatic arithmetic reasoning that we have discussed so far can be used effectively in many applications. However, some of the techniques are a little tedious to apply and the various labour-saving tricks which can be used to compensate are easy to forget.

A procedure which is occasionally useful is the conversion *z_anf_conv* which implements an arithmetic normal form for expressions formed using the basic arithmetic operators. For example, this conversion will automate the work we did by hand in the example of section 6.3 above:

SML

```
val thm27 = save_thm("thm27",
    z_anf_conv⌜Z(x + y) * (x  + y) + ∼(x*y)⌝);
```

ProofPower Output

```
val thm27 = ⊢ (x + y) * (x + y) + ∼ (x * y)
          = x * x + x * y + y * y : THM
```

Building on the conversion for arithmetic normal forms, a fully automatic approach to a useful class of problems is provided in the shape of a decision procedure for (quantifier-free) linear arithmetic. Here *linear arithmetic* means the theory of the additive arithmetic operators together with multiplication by numeric literals. This is based on the algorithm described by Hodes in [7]. Let's see it in action on part of a larger problem, which is not itself a theorem of linear arithmetic since its truth depends essentially on multiplication by variables.

SML

```
set_pc"z_library";
set_goal([], ⓩ          S 0 = 0 ∧ (∀i:ℕ• S (i + 1) = S i + (i + 1))
        ⇒   (∀j:ℕ• 2∗S j = j ∗ (j + 1))⌝);
a(REPEAT strip_tac);
```

ProofPower Output

```
(∗  3 ∗)  ⓩS 0 = 0⌝
(∗  2 ∗)  ⓩ∀ i : ℕ • S (i + 1) = S i + i + 1⌝
(∗  1 ∗)  ⓩ0 ≤ j⌝

(∗ ?⊢ ∗)  ⓩ2 ∗ S j = j ∗ (j + 1)⌝
```

Note how the condition $j \in \mathbb{N}$ has been normalised by the proof context into $0 \leq j$. This makes it convenient for us to proceed by induction:

SML

```
a(z_≤_induction_tacⓩj⌝);
```

ProofPower Output

```
...
(∗ ∗∗∗ Goal "1" ∗∗∗ ∗)

(∗  2 ∗)  ⓩS 0 = 0⌝
(∗  1 ∗)  ⓩ∀ i : ℕ • S (i + 1) = S i + i + 1⌝

(∗ ?⊢ ∗)  ⓩ2 ∗ S 0 = 0 ∗ (0 + 1)⌝
```

The base case is solved by rewriting with the assumptions (together with the arithmetic evaluation carried out by dint of the proof context that we are using).

SML

```
a(asm_rewrite_tac[]);
```

ProofPower Output

```
Tactic produced 0 subgoals:
Current goal achieved, next goal is:

(* *** Goal "2" *** *)

(*  4 *)  ⌜S 0 = 0⌝
(*  3 *)  ⌜∀ i : ℕ • S (i + 1) = S i + i + 1⌝
(*  2 *)  ⌜0 ≤ i⌝
(*  1 *)  ⌜2 * S i = i * (i + 1)⌝

(* ?⊢ *)  ⌜2 * S (i + 1) = (i + 1) * ((i + 1) + 1)⌝
```

To turn the inductive step into a problem of linear arithmetic, we must apply the conditional rewrite rule given by assumption 3. Forward chaining does the work of specialising the assumption for us:

SML

```
a(ALL_ASM_FC_T rewrite_tac[]);
```

ProofPower Output

```
...
(*  1 *)  ⌜2 * S i = i * (i + 1)⌝

(* ?⊢ *)  ⌜2 * (S i + i + 1) = (i + 1) * ((i + 1) + 1)⌝
```

We now apply the linear arithmetic proof procedure (which is packaged in a proof context called *z_lin_arith*) to complete the proof.

SML

```
a(PC_T1"z_lin_arith" asm_prove_tac[])                (* Done! *);
```

6.7 Other Operators

The arithmetic operators not covered by the methods discussed so far succumb to relatively simple methods

A few basic facts such as the triangle inequality give adequate cover for the absolute value function *abs* and the successor function is easily eliminated in favour of addition.

The integer range function crops up quite often in declarations. Membership of a range, e.g., $x \in i \mathrel{..} j$, can be eliminated in favour of a pair of inequalities, $i \leq x \land x \leq j$.

Relational iteration might be worthy of some more specialised support, but we have not yet felt the need. Direct reasoning from the definition has proved satisfactory to date.

7 Finiteness

This section of the toolkit provides the notions of finiteness and of the size of a finite set. It introduces the operators listed in the following table:

$\mathbb{F}\ X$	set of all finite subsets of X
$\mathbb{F}_1\ X$	set of all non-empty finite subsets of X
$\#\ X$	size (i.e., number of elements) of a finite set X
$X \mathrel{+\!\!\!\!\!\twoheadrightarrow} Y$, $X \mathrel{\rightarrowtail\!\!\!\!\!+} Y$	sets of finite partial functions and finite partial injections

The finite powerset operator, $\mathbb{F}$, has the following generic defining property.

$$\vdash [X](\mathbb{F}\ X = \{S : \mathbb{P}\ X \mid \exists\ n : \mathbb{N} \bullet \exists\ f : 1\ ..\ n \to S \bullet \mathit{ran}\ f = S\})$$

Our first step is to recast this in terms of an inductive definition given by the following theorem called $z\_\mathbb{F}\_thm1$;

$$\vdash [X](\mathbb{F}\ X = \bigcap\ \{u : \mathbb{P}\ \mathbb{P}\ X \mid \{\} \in u \land (\forall\ x : X;\ a : u \bullet a \cup \{x\} \in u)\})$$

Note how this reduces the notion of finiteness to set theory, eliminating all mention of arithmetic. Arguably, this would be a better definition for $\mathbb{F}$, but our approach is to take the toolkit as it comes and use theorem-proving to rectify any shortfalls.

The proof of $z\_\mathbb{F}\_thm1$ involves quite an amount of combinatorial reasoning and a number of useful lemmas about functions and integer ranges are proved *en route*. This all serves as a nice test of the support for the earlier sections of the toolkit.

A fairly immediate consequence of $z\_\mathbb{F}\_thm1$ is an induction principle: to prove that a property, holds of all finite subsets of X, show that it holds of the empty set and is preserved under singleton extensions. Using this induction principle, we can prove the following theorem:

$$\vdash [X](\mathbb{F}\ X = \mathbb{P}\ X \cap (\mathbb{F}\ \_))$$

This theorem allows us to separate concerns when reasoning about finiteness. The idiom $A \in (\mathbb{F}\_)$ (i.e., $A \in \mathbb{F}U$) just says that the set A is finite without regard to what set A is contained in. We use this idiom systematically in the rest of the theory.

Our next step is to attack the size function, $\#$. The following theorem (called $z\_\mathbb{F}\_size\_thm$) underpins most reasoning about $\#$:

$$\vdash \forall\ A : U;\ f : U;\ n : \mathbb{N} \mid f \in 1\ ..\ n \rightarrowtail\!\!\!\!\!\twoheadrightarrow A \bullet A \in (\mathbb{F}\ \_) \land \#\ A = n$$

I.e., to show A is finite with $\#A = n$, it suffices to exhibit a bijection between $1\ ..\ n$ and A.

We can then show that finiteness is preserved by the usual set forming operators, and give rules for computing the size of unions etc. Typical theorems in this part of the development are the following:

$\vdash \forall\ a : (\mathbb{F}\ \_);\ x : U \mid \neg\ x \in a \bullet \#\ (a \cup \{x\}) = \#\ a + 1$

$\vdash \forall\ a,\ b : (\mathbb{F}\ \_) \bullet a \cup b \in (\mathbb{F}\ \_) \wedge \#\ (a \cup b) + \#\ (a \cap b) = \#\ a + \#\ b$

The first of these two can be proved using $z\_\mathbb{F}\_size\_thm$ and a direct combinatorial construction. The second is then proved using the first and finite set induction.

Mainly as a test of the utility of the earlier material the current treatment of finiteness concludes by proving the pigeon-hole principle:

$\vdash \forall\ u : \mathbb{F}\ (\mathbb{F}\ \_) \mid \#\ (\bigcup\ u) > \#\ u \bullet \exists\ a : u \bullet \#\ a > 1$

In words: if u is a finite family of finite sets and the union of u has more elements than u, then some a in u has more than one element.

No decision procedures for finiteness and counting have yet been provided. However, mainly for use with Ada data types in the Compliance Tool, computational conversions are provided for some cases of the size operator.

8 Sequences

This section of the toolkit provides operations for working with sequences represented as finite functions with domains of the form $1\ ..\ n$. It introduces the operators listed in the following table:

$seq\ X$	set of all sequences of elements of X
$seq_1\ X$	set of all non-empty sequences
$iseq\ X$	set of all injective (i.e., repetition-free) sequences
$\frown$, $\frown/$	binary concatenation, distributed concatenation
$head$, $last$, $tail$, $front$, rev	lisp-like operators for sequences viewed as lists
$s \upharpoonright X$	filter non-members of X out of sequence s

The set, $seq\ X$ of all sequences with elements in X, has the following generic definition.

$\vdash [X](seq\ X = \{f : \mathbb{N} \twoheadrightarrow\!\!\!\!\!\!\shortmid\ \ X \mid dom\ f = 1\ ..\ \#\ f\})$

The use of the finite function arrow here is a major source of difficulty in the absence of a well-developed theory of finiteness. Fortunately, the theory described in section 7 above makes it relatively straightforward to prove the following much simpler characterisation.

$\vdash \forall\ X : U \bullet seq\ X = \bigcup\ \{n : \mathbb{N} \bullet 1\ ..\ n \rightarrow X\}$

This theorem and some simple combinatorial arguments entail an induction principle: to prove that a property holds of all sequences, show that it holds of the empty sequence and is preserved under extension of a sequence by concatenating with a singleton sequence. In fact, variants on this induction principle are probably desirable but have not yet been included in the ProofPower treatment.

Elementary properties of concatenation such as associativity may then be proved. Largely due to lack of demand in our applications, the treatment of the remaining operators is vestigial at the time of writing.

It turns out that reasoning about sequences has a rather schizophrenic nature. Sometimes the abstract view of sequences captured in the inductive principle works fine, and then in the very next step one has to delve into the details of a construction of a sequence as an explicit finite function. This is probably indicative of shortfalls in both approaches; articulating and implementing a more comfortable approach is an interesting area for future work.

9 Others

The remainder of the toolkit comprises some miscellanea concerned with indexed families of sets (*disjoint*, *partition*) and sets of numbers (*min*, *max*), and a theory of multisets, which it refers to as *bags*. No special support for these things is currently provided in ProofPower, although much of the support for the earlier material does apply to them; *disjoint* and *partition* at least are very straightforwardly handled just using their definitions.

10 Concluding Remarks

In this final section, I draw some conclusions on what it takes to provide effective mechanical support for proof with the Z mathematical toolkit. These remarks attempt to focus on issues which are not specific to ProofPower.

10.1 Language and Logic

I take it as given that any proof tool for Z will have means for handling the basic constructs of the language and so have concentrated in this paper on the extra facilities required to make good progress with the toolkit. Nonetheless, it should be pointed out that it is important to have systematic facilities for handling the various language constructs. In a system like ProofPower these facilities are largely the domain of people programming new proof procedures.

Purely logical reasoning is often trivial but always necessary and users of a theorem-prover have a right to expect it to be easy. Propositional reasoning should be fully automatic and a systematic approach for predicate reasoning is required. A fully automatic procedure, e.g., based on resolution, is highly desirable, even if it is limited to small problems. In ProofPower, the stripping technique implements an evident goal-oriented sequent calculus approach to the propositional structure of a problem which scales up into a component in a method for predicate calculus reasoning. A resolution based prover is available and widely used.

The treatment of purely logical problems need to be well integrated with the approaches to proof in specific domains. The proof context mechanism achieves this in ProofPower by letting the designer of proof support for a particular theory provide parameter settings which extend the power of techniques like stripping to embrace domain-specific simplifications.

10.2 Sets, Relations and Functions

Sets, relations and functions are the bread and butter of most Z specifications. The basic facts about sets and relations permit a straightforward reduction to pure logic and any theorem-prover should make it convenient for the user to exploit this. The extensional method provided by ProofPower for working with the set-theoretic and relational operators achieves this nicely. A less "expansionist" approach is convenient for working with the operators when they are used in applications of set theory in other theories. In ProofPower the algebraic proof contexts serve this purpose.

It is important to be able to exploit facts about membership of function spaces when reasoning about function application. Theorems allowing this type of reasoning are necessary and semi-automatic or automatic means for using these theorems are highly desirable. The definitions of the function arrows give rise to some mathematically trivial, but often logically tedious, difficulties. These can be ameliorated by a systematic approach to recasting the definitions. A prolog-like prover to solve the "type inference" problem of section 5 has good potential, but in ProofPower currently requires some further engineering work.

10.3 Arithmetic

For basic support of the arithmetic operators, the basic methods of elementary algebra such as multiplying out and cancelling like terms are vital as are many other basic properties. Further automation in the shape of automatic tools providing suitable normal forms for arithmetic expressions are also very important. Facilities for computing numeric literal expressions is essential and the further development of the theory is greatly eased by automation of a significant fragment like linear arithmetic.

10.4 Finiteness, Sequences and the Rest

When working with the later parts of the toolkit, one is putting to the test the support for what has come before. Success with the earlier sections of the toolkit will lead to success here.

Developing the theory of finiteness is mainly a question of carrying out the not entirely trivial combinatorial arguments which underpin the naive theory of counting. It is not yet clear whether any major fully automated techniques have a role to play in this part of the toolkit.

Our work on sequences is in relatively early days. There has been a lot of work in other contexts on automatic reasoning about lisp-like lists. This work does not apply directly to Z because it uses functions to represent sequences. However, the Z approach has an excellent mathematical pedigree. Reconciling the two points of view poses a challenge, but not I think an insoluble one.

10.5 General Strategy

Our approach to each section of the toolkit broadly follows a general paradigm for building a useful theory. First of all one tries to develop systematic methods of tackling the basic vocabulary of the theory "from first principles". In the case of sets and relations, this is via a reduction to logic; in the case of arithmetic it is via theorems which enable standard arguments from elementary algebra to be worked through step by step.

The next step is to provide methods appropriate for applications of the theory. These methods will generally avoid excessive expansion of definitions and other transformations which detract from the user's intuitions about the theory. Methods based on actual and symbolic computation are particularly valuable, if the theory admits them (cf. the discussion of the size operator in section 7 above). If the theory admits useful normal forms, then these should also be supported, but should not be forced on the user (since normal forms sometimes go against the grain of a chosen line of argument).

Identifying useful classes of problem which admit automatic proof is worthwhile throughout the process. The difficulty of providing decision procedures will obviously depend on the problem domain, many domains having been the subject of quite extensive research over the years. Selection of the decision procedures to provide involves several trade-offs. Ease of use and breadth of application should feature high on ones list of criteria for a decision procedure, as should the smoothness of integration with other methods.

Our experience has been that raw implementation of the "big" algorithms (like resolution or the linear arithmetic prover) occupies only a small part of the development effort. The bulk of our work has gone into ensuring a uniform and integrated interface and in trying to minimise the number of blind alleys (or at least dark corners) into which a carelessly designed proof tool can lead its unsuspecting user.

Acknowledgments

I gratefully acknowledge the many contributions to ProofPower and its coverage for the Z toolkit made by the present and former members of the High Assurance Team at ICL: Kevin Blackburn, Adrian Hammon, Andrew Hayward, Barry Homer, Roger Jones, David King, Gill Prout, Geoff Scullard and Roger Stokes.

The original development of ProofPower was jointly funded by ICL and the UK Department of Trade and Industry. Subsequent developments have been funded by ICL and by the Defence Research Agency, Malvern.

References

1. A.N.Whitehead and B.Russell. *Principia Mathematica.* Cambridge University Press, 1910. 3 vols.
2. R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979.
3. P.M. Cohn. *Algebra*, volume 1. John Wiley & Sons, Inc., 1974.
4. Michael J.C. Gordon. HOL:A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer, 1987.
5. Michael J.C. Gordon. Mechanising Programming Logics in Higher Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Proceedings of the 1988 Banff Conference on Hardware Verification.* Springer-Verlag, 1988.
6. Michael J.C. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF. Lecture Notes in Computer Science. Vol. 78.* Springer-Verlag, 1979.
7. Louis Hodes. Solving Problems by Formula Manipulation in Logic and Linear Inequalities. *Proceedins of the 4th International Joint Conference on Artificial Intelligence*, pages 553–559, 1971.
8. D.J. King and R.D. Arthan. Development of Practical Verification Tools. *Ingenuity - the ICL Technical Journal*, To appear, 1996.
9. L.Paulson. A Higher-order Implementation of Rewriting. *Science of Computer Programming*, 3:119–149, 1983.
10. L.Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF.* Cambridge University Press, 1987. Cambridge Tracts in Theoretical Computer Science 2.
11. C. T. Sennett. Demonstrating the Compliance of Ada Programs with Z Specifications. In R.Shaw, editor, *5th Refinement Workshop*, Workshops in Computing, pages 88–118. Springer-Verlag/BCS-FACS, 1992.
12. J.M. Spivey. *The Z Notation: A Reference Manual, Second Edition.* Prentice-Hall, 1992.
13. ECS-LFCS-86-2. *Standard ML.* R. Harper, D.B. MacQueen, and R. Milner, University of Edinburgh, 1986.