Report No. 92006

Report No. 92006

DEFENCE RESEARCH AGENCY
MALVERN

PROOF OF THEOREMS IN Z

Author: A Smith

DEFENCE RESEARCH AGENCY
Malvern, Worcestershire.

March 1992

CONDITIONS OF RELEASE

0131540 312618

MR PAUL A ROBEY
DTIC
Attn:DTIC-FDAC
Cameron Station-Bldg 5
Alexandria
VA 22304 6145
USA

************************** DRIC U

************************** DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

DEFENCE RESEARCH AGENCY, MALVERN

Report 92006

Title: Proof of Theorems in Z

Author: A. Smith

Date: March 1992

ABSTRACT

This paper is concerned with user aspects of proof in the specification language Z. A number of techniques for proving theorems in Z are presented. Some of the techniques are not new, being drawn from the area of general mathematical theorem proving, but are brought together in one place and applied specifically to Z. The techniques have been written so that they can be understood and applied by a person carrying out a pen and paper proof. Some of the techniques may be automated, and would result in a proof tool tailored to the needs of the user; currently a user has to tailor a proof to fit a tool.

CONTENTS

1 Introduction

The specification language Z has been developed to a stage where it is suitable for writing large specifications. The Z reference manual [Spivey] gives a good working definition and the language has been submitted for standardization. There are many examples of its use, the best known being the set of case studies in [Hayes]. There are many introductory texts while [Gravell] and [Macdonald] present techniques for writing clearer specifications, so that they are easier to understand. There are also a number of editors, syntax and type checkers. However, an important process in understanding and validating a specification is to reason about it. There are no guidelines on how a proof should be conducted or presented. In particular there is no standard syntax for theorems in Z. The only published syntax for Z which contains a notation for theorems is [King]. There is also a lack of tools and techniques for carrying out proofs.

This paper presents a number of useful techniques for proving theorems in Z. The paper does not contain a logic for Z (for this see [Woodcock]), but rather a collection of heuristics. Some of the techniques are not new, being drawn from the area of general mathematical theorem proving. But it is useful to see these techniques brought together in one place and applied specifically to Z. The techniques have been written so that they can be understood and applied by a person carrying out a pen and paper proof. However, they could also be automated, resulting in a tool tailored to the needs of the user; currently all too often a user has to tailor a proof to fit a tool [Smith b].

The content of the paper is as follows. Section 2 contains a useful proof technique known as *generalization*. This is where a theorem is strengthened, roughly keeping its original structure, so that the new theorem is more useful, and surprisingly, sometimes easier to prove. The original theorem follows as a special case of the generalized theorem. Section 3 contains techniques for proof by induction. One technique shows how to choose the right induction schema and induction variable, while another shows how to finish the proof once the induction step has been performed. Section 4 explains a style of reasoning known as *window inference*. This is a technique where a user can transform an expression by restricting attention, or windowing, on a subexpression. In this section, window inference is formalized so that it can be automated. Section 5 presents a technique to help check the consistency of a Z specification. The technique is for proving the existence of a recursive function defined over a recursive free type. The technique extends that in [Smith a] so that a wider class of functions are covered. Section 6 discusses reasoning at the schema level, and presents some useful laws for calculating preconditions. Section 7 discusses the conflict between specification and proof. Sometimes a theorem is easier to prove from an alternative specification. But this alternative specification can be harder to understand. Finally section 8 contains the conclusions of the paper, and suggestions for further work.

2 Generalization

Sometimes it is easier to prove a theorem by generalizing it. The original theorem then follows as a special case of the generalized theorem. At first sight this seems odd. How can a more powerful theorem be easier to prove? A reason is that it removes irrelevant detail from the problem. This section contains two techniques for generalization: the first is to replace a subterm in a theorem by a variable; the second is to use *higher order functions*. A generalized theorem is also more useful, because it can be applied to more cases. It can therefore not only be used to obtain the original theorem, but other theorems as well. Since a generalized theorem is more useful and sometimes easier to prove, generalization is of interest to a person carrying out a pen and paper proof. It is also of interest to a tool builder because the first, and part of the second technique, can be automated.

2.1 Replacing a Subterm by a Variable

The first technique for generalizing a theorem is to replace every occurrence of a particular subterm by a variable, thus removing unnecessary clutter. The technique is described in [Bird] and [Brumfitt], and is illustrated in the next example. All examples in this paper start with the word *Example* and end with the symbol Δ.

Example 1

Consider the following specification

```
_+_ : (ℕ × ℕ) → ℕ
─────────────────────────
∀ ..  n : ℕ •
       + m = m
      (n) + m = s(n + m)
```

```
sum : ℕ → ℕ
─────────────────────────
∀ n : ℕ •
    sum 0 = 0
    sum(s n) = s(n) + sum(n)
```

The function + is addition over ℕ, *s* is the successor function, and the expression *sum n* is the sum of the first *n* natural numbers. Now consider the theorem

$$\vdash \forall\ n : \mathbb{N} \bullet sum(n) + 0 = sum(n) \quad (A)$$

This theorem can be proved by induction over ℕ, using the above axioms. But during the proof, a lemma (the associativity of +) is also required. Again this would have to be proved using induction. So the total effort in proving theorem *A* is two induction proofs. Now it is easier to prove a generalization of theorem *A*, namely

$$\vdash \forall\ x : \mathbb{N} \bullet x + 0 = x \quad (B)$$

Notice how the subterm *sum(n)* in theorem *A* (which appears in two places) has been replaced by a simple variable *x*. This has removed the function *sum* which is just clutter in the original theorem. Once theorem *B* has been proved, theorem *A* follows immediately by specializing *x* with *sum(n)*. The proof of *B* is again by induction over ℕ, but does not require a lemma. Not only is theorem *B* easier to prove, it is more useful. It can be used to prove the original theorem *A* without induction, by specializing *x* with *sum(n)*. Similarly it can be used to prove the theorem

$$\vdash \forall\ n : \mathbb{N} \bullet n! + 0 = n!$$

where *!* is the factorial function, by specializing *x* with *n!*. Δ

Another case when a more general theorem can be easier to prove is in proof by induction. Although generalizing might strengthen the induction conclusion, it also strengthens the induction hypothesis. This stronger hypothesis can help in proving the theorem. The next example illustrates this, using the same technique of replacing a subterm with a variable.

Example 2

Consider the following *quick reverse* function for sequences

$$\begin{array}{|l}
[X] \\ \hline
qrev : (\mathrm{seq}\ X \times \mathrm{seq}\ X) \rightarrow \mathrm{seq}\ X \\ \hline
\forall\ x : X;\ s,\ t : \mathrm{seq}\ X \bullet \\
\quad qrev(\langle\rangle,\ t) = t \\
\quad qrev(\langle x \rangle {}^\frown s,\ t) = qrev(s,\ \langle x \rangle {}^\frown t) \\ \hline
\end{array}$$

Consider the following theorem

$$\vdash \forall\ s : \mathrm{seq}\ X \bullet rev\ s = qrev(s,\ \langle\rangle) \qquad (A)$$

where rev is the ordinary reverse function as defined in [Spivey]. Proving this theorem by induction (over sequences) on s does not work. The problem is in the step case, where the induction hypothesis is not strong enough. But proving a generalization of the theorem *is* successful. The generalization is to notice that theorem A can be written

$$\vdash \forall\ s : \mathrm{seq}\ X \bullet (rev\ s) {}^\frown \langle\rangle = qrev(s,\ \langle\rangle)$$

and then generalized to

$$\vdash \forall\ s,\ t : \mathrm{seq}\ X \bullet (rev\ s) {}^\frown t = qrev(s,\ t) \qquad (B)$$

where the subexpression $\langle\rangle$ has been replaced by the variable t. Proving B by induction on s now gives a stronger induction hypothesis, sufficient to prove the theorem. Δ

2.1.1 Strengths and Weaknesses

The technique of replacing a subterm with a variable could be automated, and is therefore also of interest to a tool builder. Indeed, the technique has been automated in the Boyer-Moore theorem prover [Boyer]. Unfortunately, if the subterm is chosen arbitrarily, the generalized version might not be a theorem. In order to avoid this, an understanding of the problem is necessary. The technique is still safe, because the user would not be able to *prove* the generalized version. The next example illustrates this.

Example 3

Consider the theorem

$$\vdash \forall\ x : X \bullet rev\ \langle x \rangle = \langle x \rangle$$

Replacing the subterm $\langle x \rangle$ by a variable y gives

$$\vdash \forall\ y : \mathrm{seq}\ X \bullet rev\ y = y$$

which is obviously nonsense. Δ

2.2 Using Higher Order Functions

A second technique for generalizing theorems is to use *higher order functions*. This technique is described in [Bird] and [Brumfitt]. In this section, a higher order function will be one which takes a function as argument, and gives a function as result.

In general, a theorem contains a number of objects, for example functions, relations or sets. These objects have a number of properties, but the theorem will only depend on some of these properties. The theorem will also hold for other objects, provided they too have these required properties. Higher order functions allow these required properties to be separated from the irrelevant properties. For example, consider a theorem concerning sequences. Suppose that the theorem only depends on properties of sequences (*collections* of elements), and not on properties of *individual* elements of sequences. A higher order function can be used to abstract, or lift, the properties about the collection of elements from properties about individual elements. There is an analogy here with the technique of *promotion* in Z. This is where a *framing* schema (analogous to a higher order function) lifts the properties about collections of objects away from properties about individual objects. The next example illustrates the use of higher order functions to generalize a theorem.

Example 4

Consider the function $square\_seq$ which squares every element of a sequence of integers. For example, $square\_seq\ \langle 5, 1, -2\rangle = \langle 25, 1, 4\rangle$. Formally

$$\begin{array}{|l}
\hline
square\_seq : seq\ \mathbb{Z} \rightarrow seq\ \mathbb{Z} \\
\hline
\forall\ i : \mathbb{Z};\ s : seq\ \mathbb{Z} \bullet \\
\quad square\_seq\ \langle\rangle = \langle\rangle \\
\quad square\_seq(\langle i\rangle \frown s) = \langle i * i\rangle \frown (square\_seq\ s) \\
\hline
\end{array}$$

Consider the following theorem

$$\vdash \forall\ s, t : seq\ \mathbb{Z} \bullet \\ \quad square\_seq(s \frown t) = (square\_seq\ s) \frown (square\_seq\ t) \qquad (A)$$

If the function $square\_seq$ in the above theorem is replaced by the function add_one_seq (which adds one to every element of a sequence), then the theorem still holds. In fact, the operation on the *individual* elements of the sequence (for example squaring and adding one) is irrelevant. This fact can be captured by using the higher order function map, familiar from functional programming (see for example [Bird]), and defined as

$$\begin{array}{|l}
\hline
map : (\mathbb{Z} \rightarrow \mathbb{Z}) \rightarrow (seq\ \mathbb{Z} \rightarrow seq\ \mathbb{Z}) \\
\hline
\forall\ f : \mathbb{Z} \rightarrow \mathbb{Z};\ i : \mathbb{Z};\ s : seq\ \mathbb{Z} \bullet \\
\quad map\ f\ \langle\rangle = \langle\rangle \\
\quad map\ f\ (\langle i\rangle \frown s) = \langle f\ i\rangle \frown (map\ f\ s) \\
\hline
\end{array}$$

Notice the similarity between the definition of map and that of $square\_seq$. Notice how the particular operation on the individual elements, $i * i$, has been replaced by the general operation $f\ i$. The functions $square\_seq$ and add_one_seq, for example, can be written

$$\begin{array}{l}
square == \lambda\ i : \mathbb{Z} \bullet i * i \\
add\_one == \lambda\ i : \mathbb{Z} \bullet i + 1 \\
\\
square\_seq == map\ square \\
add\_one\_seq == map\ add\_one
\end{array}$$

The generalized version of theorem A is then obtained by replacing $square\_seq$ by $map\ f$, and universally quantifying over f, to obtain

$$\vdash \forall\ f : \mathbb{Z} \rightarrow \mathbb{Z};\ s, t : seq\ \mathbb{Z} \bullet \\ \quad map\ f\ (s \frown t) = (map\ f\ s) \frown (map\ f\ t) \qquad (B)$$

Theorem *B* can be proved by induction on *s*, and then simply specialized to obtain theorem *A* and similar theorems with functions other than *square_seq*. As an aside, notice that the type $\mathbb{Z}$ in theorem *A* could also be generalized (to the generic type X say), since the type of the elements in the sequence is also irrelevant. This would mean defining *map* to be generic in X and replacing every occurrence of $\mathbb{Z}$ in theorem *B* by X. Δ

The above example showed the use of a higher order function to generalize a theorem concerning sequences. Sequences are very similar to lists, and lists can be specified in Z using the free type mechanism. This suggests that the technique of using higher order functions can also be applied to theorems involving free types. This is illustrated in the next example.

Example 5

Consider the free type

$$LIST ::= nil \mid join \langle\langle \mathbb{Z} \times LIST \rangle\rangle$$

which consists of lists of integers. Generalizing theorems involving *LIST* is similar to the problem of generalizing theorems involving sequences in example 4. For example, if _ ^_ is concatenation of *LIST*s and *square_list* is a function which squares every element of a *LIST*, then theorems such as

$$\begin{aligned}&\vdash \forall\ l, m : LIST \bullet\\ &\qquad square\_list(l \mathbin{\char`\^} m) = (square\_list\ l) \mathbin{\char`\^} (square\_list\ m)\end{aligned}$$

can be generalized in a similar way as in example 4. The generalization would involve a higher order function, similar to *map*, but for *LIST*s rather than sequences. Δ

2.2.1 Strengths and Weaknesses

It has been shown that theorems involving higher order functions are very useful. They can be stored in a library and repeatedly used to obtain other theorems. This cuts down the proof effort. But where do these higher order functions come from? It is significantly more difficult to find a higher order function, than to replace a subterm with a variable. But some help can be offered. For example, if the theorem in example 4 is proved *without* using higher order functions, then it can be seen that no property of multiplication is used (the reader might like to try this). This means that the operation on the individual elements of the sequences is irrelevant. Therefore, the higher order function *map* needs only to consider an arbitrary operation on the elements. As far as theorems involving free types are concerned, a *single* higher order function can be automatically generated from the *primitive recursion theorem (PRT)* for the free type [Smith a]. This function can be used to express *any* primitive recursive function over the free type. The work described here is an extension to [Smith a]. The next example shows how the PRT for the free type *LIST* in example 5 can be used to obtain this single higher order function. It is then used to define two functions over *LIST*.

Example 6

The PRT for the free type *LIST* in example 5 is (from [Smith a])

$$\begin{aligned}&\vdash \forall\ x : X;\ f : (X \times \mathbb{Z} \times LIST) \to X \bullet\\ &\qquad \exists_1\ h : LIST \to X \bullet\\ &\qquad\qquad h\ nil = x\\ &\qquad\qquad \forall\ i : \mathbb{Z};\ l : LIST \bullet h(join(i, l)) = f\ (h\ l, i, l)\end{aligned}$$

The theorem says that a primitive recursive function h, over the free type *LIST*, is uniquely defined by its base case (x) and its recursive case (f). The theorem is generic in

x, which is the target type of h. For example, if h is the function which finds the size of a list then x will be $\mathbb{N}$. The fact that each x and f gives rise to a *unique* h means there is a function (rather than a relation) H say, linking x and f to h. That is

$$H(x,\ f) = h$$

The function H is the single higher order function discussed above. The formal definition of H is derived from the PRT, by replacing h with $H(x,\ f)$. The definition is as follows

$$
\begin{array}{l}
[X] \\
\hline
H : (X \times ((X \times \mathbb{Z} \times LIST) \rightarrow X)) \rightarrow (LIST \rightarrow X) \\
\hline
\forall\ x : X;\ f : (X \times \mathbb{Z} \times LIST) \rightarrow X \bullet \\
\quad H(x,\ f)\ nil = x \\
\quad \forall\ i : \mathbb{Z};\ l : LIST \bullet H(x,\ f)\ (join(i, l)) = f\ (H(x,\ f)\ l,\ i,\ l) \\
\hline
\end{array}
$$

Using H, the functions $size$ (the number of elements in a $LIST$), and the concatenation of two lists, _^_, can be written

$$
\begin{array}{l}
f1 == \lambda\ n : \mathbb{N};\ i : \mathbb{Z};\ l : LIST \bullet n + 1 \\
f2 == \lambda\ l : LIST;\ i : \mathbb{Z};\ m : LIST \bullet join(i,\ l) \\
\\
size == H(0,\ f1) \\
\_\hat{}\_ == \lambda\ l,\ m : LIST \bullet H(m,\ f2)\ l
\end{array}
$$

Δ

For free types, given the PRT, the single higher order function described above can be automatically generated, and is therefore also of interest to a tool builder. In general, although higher order functions make theorem proving easier, they make the proof harder to understand. Quite often a more obscure specification can lead to an easier proo' This conflict between specification and proof is discussed in section 7.

3 Induction

Proofs by induction occur frequently: the proof of any property defined over an infinite set (for example the natural numbers) will almost inevitably involve induction. Proof by induction is the basis of the *Oyster-Clam* system [Bundy a], which carries out *program synthesis*. Program synthesis is where a program is extracted from the proof of a theorem. The theorem to be proved is

$$\vdash \forall\ inputs \bullet \exists\ output \bullet spec(inputs,\ output) \quad (A)$$

where $spec(inputs,\ output)$ is a specification of the program. Theorem A says that the program is required to produce an output satisfying the specification for every input. What is required for the proof is the construction of an existential witness for theorem A which will be the program $prog(inputs)$ satisfying the theorem

$$\vdash \forall\ inputs \bullet spec(inputs,\ prog(inputs))$$

Given that theorem A starts with a universal quantifier, a proof by induction will usually be appropriate. Each step of the proof corresponds to the introduction of a program construct. For example, an inductive proof step corresponds to the introduction of a recursive procedure. Proving A can involve complex induction schemata, corresponding to the program structure.

For any proof by induction, a choice has to be made of the induction variable and the induction schema; that is, what will constitute the base and step cases. For example, proving properties of the even numbers will usually involve stepping the induction variable by two rather than one. The induction schema together with the induction variable will be called the *induction strategy*. This section presents a technique for choosing the induction strategy. When the induction strategy has been chosen, the proof must be completed, so this section also presents a technique for completing the proof. The techniques presented here are of interest to a person carrying out a pen and paper proof, because they help to find the right induction strategy, and to complete the proof. They are of interest to a tool builder as both techniques can be automated.

3.1 Choosing the Induction Strategy

The technique presented here for choosing the induction strategy comes from the Boyer-Moore theorem prover [Boyer]. Firstly, the technique is explained when there is only one recursive function and one possible induction variable. Theorems involving more than one recursive function and more than one possible induction variable are discussed later.

For theorems involving only one recursive function and one possible induction variable, then obviously that variable must be chosen. The induction schema chosen should be that one which mirrors the form of recursion used to define the function. For example, the induction schema will have the same number of base and step cases as the form of recursion. The next two examples illustrate the technique.

Example 7

Consider the following recursive function

$$\begin{array}{|l}
\_+\_ : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N} \\
\hline
\forall\, m, n : \mathbb{N} \bullet \\
\quad 0 + m = m \qquad\qquad\qquad (1) \\
\quad s(n) + m = s(n + m) \qquad (2)
\end{array}$$

The theorem

$$\vdash \forall\, n : \mathbb{N} \bullet n + 0 = 0$$

can be proved using the one step induction schema

$$\begin{array}{ll}
\vdash P(0) \wedge & (S1) \\
\quad \forall\, n : \mathbb{N} \bullet P(n) \Rightarrow P(s\ n) & \\
\quad \Rightarrow & \\
\quad \forall\, n : \mathbb{N} \bullet P(n) &
\end{array}$$

The base case in $S1$, namely $P(0)$, corresponds to the base case of + (axiom 1). Similarly the step case in $S1$, namely

$$\forall\, n : \mathbb{N} \bullet P(n) \Rightarrow P(s\ n)$$

corresponds to the form of recursion in the step case of + (axiom 2). Δ

Example 8

Consider the following recursive definition of the set of even natural numbers.

$$
\begin{array}{|l r}
\hline
\mathit{even} : \mathbb{P}\,\mathbb{N} & \\
\hline
\forall\, n : \mathbb{N} \bullet & \\
\quad 0 \in \mathit{even} & (1) \\
\quad s(0) \notin \mathit{even} & (2) \\
\quad s(s\ n) \in \mathit{even} \Leftrightarrow n \in \mathit{even} & (3) \\
\hline
\end{array}
$$

The theorem

$$\vdash \forall\, n : \mathbb{N} \bullet n \in \mathit{even} \Rightarrow s(n) \notin \mathit{even}$$

can be proved using the two step induction schema

$$
\begin{array}{l r}
\vdash P(0) \land & (S2) \\
\quad P(s\ 0) \land & \\
\quad \forall\, n : \mathbb{N} \bullet P(n) \Rightarrow P(s(s\ n)) & \\
\quad \Rightarrow & \\
\quad \forall\, n : \mathbb{N} \bullet P(r) &
\end{array}
$$

The two base cases in $S2$, namely $P(0)$ and $P(s\ 0)$, correspond to the two base cases in the definition of $even$ (axioms 1 and 2). The step case in $S2$, namely

$$\forall\, n : \mathbb{N} \bullet P(n) \Rightarrow P(s(s\ n))$$

corresponds to the form of recursion in axiom 3 of $even$. Δ

It is worth saying at this point that induction schemata $S1$ and $S2$ are logically equivalent. It is just that one induction schema is more appropriate for a theorem than another. Now suppose that a theorem involved more than one recursive function. Also suppose the theorem contained more than one possible induction variable. (In examples 7 and 8 there was only one possible induction variable, namely n.) Which induction schema, and which induction variable should be chosen? The technique for choosing the induction strategy is as follows. The explanation is based on that given in [Bundy b]. During the explanation, the phrase *recursion term* is used. This is a term such as $s(n)$ in example 7 and $s(s\ n)$ in example 8. The recursion term appears in both the definition of a recursive function, and in the corresponding induction schema.

Firstly, the form of the theorem is analysed to produce a number of *raw induction suggestions*. These suggestions are then combined to produce a single induction strategy which will be used for the proof. The raw induction suggestions are generated as follows. The theorem is scanned to locate its recursive functions. Each occurrence of a recursive function f, with a variable x in its recursive argument position, produces a raw induction suggestion (the recursive argument position for the function $+$ in example 7 is its left hand argument). The raw induction suggestion consists of the induction schema that mirrors the form of recursion used to define f, and induction variable x.

These suggestions are then combined as follows. It may be possible for one suggestion to *subsume* another. This will be the case if the two suggestions consist of the same induction variable, and the recursion term of one induction schema consists of repeated nestings of the recursion term of the other schema. For example, the recursion term $s(s\ n)$ in example 8 subsumes the recursion term $s(n)$ in example 7. If it is not possible for one suggestion to subsume another, then it may be possible to produce a *new* suggestion that subsumes both. This is achieved by *merging* the two suggestions. For example, a two step schema and a three step schema can be merged to give a six step schema. After subsumption and merging has been carried out there will be one suggestion for each induction variable. It then remains to choose one of these suggestions as the induction strategy for the theorem. This final step is achieved by considering the terms that would occur in the induction conclusion, for each suggestion. Some suggestions would result in terms that could not be rewritten using the definitions of the recursive functions. Such suggestions are *flawed* and are removed. If more than one suggestion remains, the winner is the one that subsumes the largest number of raw suggestions. The next example illustrates the technique.

Example 9

Consider the theorem

$$\vdash \forall\ n, m : \mathbb{N} \bullet n \in even \wedge m \in even \Rightarrow (n + m) \in even$$

where $+$ and $even$ are defined in examples 7 and 8. This theorem contains two recursive definitions, $even$ and $+$, and two possible induction variables, n and m. The raw induction suggestions are

$$\begin{array}{ll} (S2,\ n) & (A) \\ (S2,\ m) & (B) \\ (S1,\ n) & (C) \end{array}$$

where $S1$ and $S2$ are the one step and two step induction schemata appearing in examples 7 and 8. Suggestion A is generated by the first occurrence of $even$ in the theorem, with the variable n in its recursive argument position. Suggestion B is generated by the second occurrence of $even$, with the variable m in its recursive argument position. Suggestion C is generated by the occurrence of $+$, with the variable n in its recursive argument position.

These three suggestions are combined to form the induction strategy as follows. Suggestion A subsumes suggestion C, because they both contain the induction variable n, the recursion term in $S2$ is $s(s\ n)$, and the recursion term in $S1$ is $s(n)$. This leaves suggestions A and B. Suggestion B is flawed because it would produce the term $n + s(s\ m)$ in the induction conclusion, and this can not be rewritten using the definition of $+$. The induction strategy is therefore A; two step induction on n. Δ

3.1.1 Strengths and Weaknesses

Using the above technique, an induction strategy can be automatically generated. The technique has been automated in the Boyer-Moore theorem prover [Boyer], and in the Oyster-Clam system [Bundy b]. But the generated induction strategy might not always be appropriate to prove the theorem. Once again, the technique is still safe, it just means that the user will not be able to prove the theorem using that induction strategy. The next example illustrates this.

Example 10

Consider the theorem

$$\vdash \forall\ m, n : \mathbb{N}_1 \bullet m^2 \neq 2*n^2$$

Suppose the two functions that appear in this theorem (multiplication and exponentiation) are defined recursively (which is usually the case). The above technique would produce an induction schema based on these two recursive functions. But the appropriate induction schema for this theorem is

$$\begin{array}{l} \vdash \forall\ n : \mathbb{N} \bullet (\forall\ n' : \mathbb{N} \mid n' < n \bullet P(n')) \Rightarrow P(n) \\ \Rightarrow \\ \forall\ n : \mathbb{N} \bullet P(n) \end{array}$$

This is a case of *Noetherian* induction. This induction schema says that $P(n)$ must be proved under the assumption that P holds for all values less than n. That is, the induction conclusion $P(n)$ must be proved from the induction hypothesis

$$\forall\ n' : \mathbb{N} \mid n' < n \bullet P(n')$$

Basically, this means that for the above theorem, $m^2 \neq 2*n^2$ must be proved under the assumption that $m^2 \neq 2*n^2$ is true for all smaller values of m and n. This is achieved by assuming $m^2 = 2*n^2$ (the negation of the induction conclusion) and obtaining a contradiction. The contradiction arises, because in assuming $m^2 = 2*n^2$, it can be proved that $m'^2 = 2*n'^2$, where m' and n' are smaller than m and n. This contradicts the induction hypothesis. Δ

3.2 Completing the Proof

Once the induction strategy has been chosen, the proof must be completed. In [Bundy a] a technique is described for manipulating the induction conclusion so that the induction hypothesis can be used. This technique is called *rippling-out*. The difference between an induction hypothesis and conclusion is in the recursion term (recursion terms were described in section 3.1). For example, the induction strategy finally chosen in example 9, used the recursion term $s(s(n))$. The induction conclusion contains $s(s(n))$ wherever the induction hypothesis contains just n. The expression $s(s(\ldots))$ (the recursion term without the n) is an example of a *wave front*. To complete the proof, the wave front must be moved outwards from its deeply nested positions (or rippled, like a wave on a pond), to reveal a copy of the induction hypothesis. The next example illustrates rippling-out.

Example 11

Consider the function + as described in example 7, namely

$$\boxed{\begin{array}{ll} \_+\_ : (\mathbb{N} \times \mathbb{N}) \rightarrow \mathbb{N} & \\ \hline \forall\ m, n : \mathbb{N} \bullet & \\ \quad 0 + m = m & (1) \\ \quad s(n) + m = s(n + m) & (2) \end{array}}$$

Proving the associativity of +, namely

$$\vdash \forall\ x, y, z : \mathbb{N} \bullet (x + y) + z = x + (y + z)$$

by induction on x gives an induction hypothesis

$$(x + y) + z = x + (y + z)$$

and an induction conclusion

$$(s(x) + y) + z = s(x) + (y + z)$$

Using repeated applications of axiom 2, the wave front $s(\ldots)$ can be rippled outwards to give

$$s((x + y) + z) = s(x + (y + z)) \quad (A)$$

Since s is an injective function, it obeys the law

$$(s(m) = s(n)) = (m = n)$$

for any natural numbers m and n. Using this law, expression A above can be rewritten as

$$(x + y) + z = x + (y + z)$$

The wave front $s(\ldots)$ has now completely rippled-out, revealing a copy of the induction hypothesis. The proof is therefore complete. Δ

3.2.1 Strengths and Weaknesses

The technique of rippling-out is easy to apply. The user simply has to apply axioms so that wave fronts in the induction conclusion move outwards. The technique has been automated in the Oyster-Clam system [Bundy a]. The technique is used by Oyster-Clam as a central strategy for proof. In this system, not only does the technique complete an induction proof, it also plays a part in choosing the induction strategy. This is because the technique for choosing the induction strategy (section 3.1) has been automated in Oyster-Clam so as to look ahead to ensure subsequent rippling can proceed. But the technique of rippling-out is not always appropriate. For example, sometimes a wave front needs to move *sideways* instead of outwards. That is, the wavefront remains at the same level of nesting. The next example illustrates this.

Example 12

Consider the function $qrev$ in example 2. One of the axioms of $qrev$ is

$$qrev(\langle x\rangle^\frown s,\ t) = qrev(s,\ \langle x\rangle^\frown t)$$

Using this axiom in a proof will move the wavefront, $\langle x\rangle^\frown$, sideways. The wavefront is at the same level of nesting on both sides of the above axiom. Δ

Extensions to rippling-out have been made in [Bundy c]. They include rippling sideways as described above, but are beyond the scope of this report. The extensions have been automated in Oyster-Clam.

4 Window Inference

Window inference is a style of reasoning which enables an expression to be transformed by restricting attention to a subexpression. This is called opening a *window* on the subexpression. Window inferencing transforms an expression without affecting the rest of the expression, but allows contextual information to be used while transforming the subexpression. The contextual information is derived from the original expression minus the subexpression. This section formalizes window inference so that it can be automated and is therefore of interest to a tool builder. The formalization presented here is from [Grundy].

4.1 Formalizing Window Inference

It is useful to see an example of window inference before attempting to formalize it. The next example illustrates the use of window inference to simplify an expression.

Example 13

Suppose the expression

$$\langle head\ s\rangle ^\frown (tail\ s) = t \wedge s \neq \langle\rangle \quad (A)$$

where s and t are sequences, is to be simplified. Consider the following law from [Spivey]

$$\vdash s \neq \langle\rangle \Rightarrow \langle head\ s\rangle \frown (tail\ s) = s \quad (B)$$

How can this law be used to simplify expression A? Window inference allows a window to be opened (shown as a box below) on a subexpression of A

$$\boxed{\langle head\ s\rangle \frown (tail\ s) = t} \wedge s \neq \langle\rangle$$

The contextual information, $s \neq \langle\rangle$ (the remainder of A) can then also be used to simplify the window expression. This contextual information, together with theorem B, enables the following fact to be deduced (by modus ponens)

$$\langle head\ s\rangle \frown (tail\ s) = s$$

Rewriting with this new fact allows the window expression to be simplified, so that the original expression becomes

$$s = t \wedge s \neq \langle\rangle$$

Δ

Window inference can be formalized by using *window rules*. A window rule is a particular type of inference rule. For example, the window inference carried out in the last example can be formalized with the window rule

$$\frac{Q,\ \Gamma \vdash P \Leftrightarrow P'}{\Gamma \vdash (P \wedge Q) \Leftrightarrow (P' \wedge Q)}$$

In this rule the complete expression is $P \wedge Q$, the window expression is P, and the contextual information is Q. The rule states that if the window expression can be simplified to P' (using the contextual information) then the complete expression can be simplified to $P' \wedge Q$. The symbol Γ denotes a list (possibly empty) of other facts that can be used when simplifying. In the last example Γ consisted of theorem B. The general form of a window rule is

$$\frac{\gamma,\ \Gamma \vdash e\ r\ e'}{\Gamma \vdash E[e]\ R\ E[e']}$$

The complete expression is $E[e]$, the window expression is e, and the contextual information is γ. The transformed window expression is e' which means that the complete expression is transformed to $E[e']$. The symbol Γ denotes a list (possibly empty) of additional facts that may be used during the transformation. The relationship between the original and transformed window expression is r. Similarly, the complete expressions before and after transformation are related by R. In general r and R are different as illustrated in the next example.

Example 14

Consider the formal refinement of a specification to code (see for example [Morgan]). When strengthening the postcondition it is natural to do so under the assumption that the precondition holds. This can be formalized using the window rule

$$\frac{pre,\ \Gamma \vdash post \Leftarrow post'}{\Gamma \vdash w : [pre,\ post] \sqsubseteq w : [pre,\ post']}$$

The symbol $\sqsubseteq$ is the refinement relation and w is the list of variables whose values may change. In this example r is "reverse" implication and R is refinement. Therefore when carrying out a formal refinement of the statement

$$w : [pre,\ post]$$

a window may be opened on the subexpression $post$ as shown

$$w : [pre,\ \boxed{post}\ ]$$

while assuming the contextual information pre. Δ

4.2 Opening a Window Within a Window

A window can be opened within a window and so on, creating a *window stack*. To formalize this consider the pair of relations $(r,\ R)$ appearing in a general window rule. A window may be opened inside another provided that r for the outer window is the same as R for the inner. The next example illustrates this.

Example 15

Consider a backwards (subgoaling) style of proof. A goal of the form

$$x \in \{\ D \mid P \bullet u\ \}$$

can often be solved by strengthening the predicate P to P'. This can be achieved in a series of window inference steps as follows. First of all a window is opened on the set using the window rule

$$\frac{\Gamma \vdash S \supseteq S'}{\Gamma \vdash x \in S \Leftarrow x \in S'}$$

This rule states that the set must be transformed into a new set with less elements. This is achieved by opening a window on the subexpression P using the window rule

$$\frac{D,\ \Gamma \vdash P \Leftarrow P'}{\Gamma \vdash \{\ D \mid P \bullet u\ \} \supseteq \{\ D \mid P' \bullet u\ \}}$$

and strengthening the predicate P (to give a new predicate P'). Once this has been achieved (by opening another window perhaps, or using a theorem), both windows can be closed to give the new subgoal

$$x \in \{\ D \mid P' \bullet u\ \}$$

Δ

4.3 Strengths and Weaknesses

Window inference is a powerful technique as it allows the user to concentrate on a subexpression, and assume contextual information while doing so. There are many areas where window inference is useful, for example the simplification of expressions, refinement, and goal directed theorem proving. A disadvantage is that the window rules which formalize the technique are complicated. If window inference is automated, then although window rules are complicated they will be hidden from the user. The tool will allow a window to be opened up provided that it has a window rule to justify it. Window inference has been automated in the HOL theorem prover [Grundy].

5 Consistency of Z Specifications

It is important that a Z specification is consistent. An inconsistent specification may lead to false conclusions in reasoning thus destroying the point of having a specification. For example, a specification of a natural number x using the axiom $x = x + 1$ is syntactically correct and well typed, but no such x exists. In general, for each object specified, a theorem should be proved stating that the object exists.

One way in which inconsistencies can arise is when using a Z free type. [Smith a] shows that a user defined recursive free type may not exist, or even if it does, a recursive function defined over it may not. [Smith a] presents techniques for checking the consistency of recursive free types, and recursive functions defined over them. This section extends the technique so that a wider class of functions can be checked. The extension is of interest to a person carrying out a pen and paper proof so as to avoid false conclusions when reasoning. It is of interest to a tool builder as the extension can be automated.

5.1 Functions Defined Over Free Types

[Smith a] presents a technique to prove the existence of a recursive function defined over a recursive free type. But the technique can only be used for a function defined by *primitive* recursion. The technique is now extended to cover some non-primitive recursive functions. Basically the idea is to rewrite the non-primitive definition in terms of a primitive one, after which the technique in [Smith a] may be used. The reader needs only to appreciate how a non-primitive function can be rewritten in terms of a primitive one. The extension only covers non-primitive functions defined over a free type of the form

$$T ::= a1 \mid \ldots \mid am \mid c1 \langle\langle T \rangle\rangle \mid \ldots \mid cn \langle\langle T \rangle\rangle$$

(the basic technique in [Smith a] covers all free types). A function f defined by primitive recursion over the above free type T has a recursive case of the form

$$f(ci\ t) = A(f\ t)$$

where $A(f\ t)$ is some expression involving $f\ t$. A non-primitive function g might have a step case of the form

$$g(c1(c2\ t)) = B(\ g(c1\ t),\ g(c2\ t),\ g\ t\ ) \qquad (1)$$

In general, if there are n constructors ci on the left hand side of 1 then the right hand side will contain applications of g containing zero, one, two, ..., $n-1$ constructors. Such a function g can be written in terms of a primitive recursive function $g1$. The idea is that $g1\ t$ delivers the tuple

```
(g t,
 g(c1 t), ... , g(cn t),
 applications of g with two constructors,
 .
 .
 .
 applications of g with n-1 constuctors
)
```

(this tuple will contain $1 + n + n^2 + ... + n^{n-1}$ elements). It is then the case that *g1(ci t)* can be written in terms of *g1 t* (note this is primitive recursion). Once *g1* has been constructed the original function *g* can be written *g t = FST(g1 t)*, where *FST* projects the first element from the tuple. The next two examples illustrate the technique.

Example 16

The natural numbers can be considered as the free type

```
ℕ ::= 0 | s ⟪ ℕ ⟫
```

Consider the function

```
┌──────────────────────────────────────────
│ fib : ℕ → ℕ
├──────────────────────────────────────────
│ ∀ n : ℕ •
│     fib 0 = 0                                (1)
│     fib 1 = 1                                (2)
│     fib(n+2) = fib(n+1) + fib(n)             (3)
└──────────────────────────────────────────
```

which generates the Fibonacci numbers. The function *fib* is defined by non-primitive recursion. A primitive recursive function *fib1* is now constructed. The idea behind the construction is that

```
fib1 n = (fib n, fib(n+1))    (A)
```

Using this idea

```
  fib1 0
= (fib 0, fib 1)      (using A)
= (0, 1)              (using axioms 1 and 2 of fib)
```

and

```
  fib1(n+1)
= (fib(n+1), fib(n+2))                        (using A)
= (fib(n+1), fib(n+1) + fib(n))               (using axiom 3 of fib)
= (λ x, y : ℕ • (y, y+x)) (fib n, fib(n+1))   (β-abstraction)
= (λ x, y : ℕ • (y, y+x)) (fib1 n)            (using A)
```

The formal definition of *fib1* is therefore

```
h == λ x, y : ℕ • (y, y + x)
```

```
┌──────────────────────────────
│ fib1 : ℕ → (ℕ × ℕ)
├──────────────────────────────
│ ∀ n : ℕ •
│     fib1 0 = (0, 1)
│     fib1(n+1) = h(fib1 n)
└──────────────────────────────
```

Notice that this is a primitive recursive function since `fib1(n+1)` is defined in terms of `fib1(n)`. The original function `fib` can then be written

```
fib n = fst(fib1 n)   (B)
```

The above approach of using the function `fib1` is also used in producing the *fast Fibonacci* function in functional programming, described in [Bird]. The definition B, considered as a functional program, is more efficient than the original definition, taking less rewrites to evaluate the *n*th Fibonacci number. Δ

Example 17

Consider the free type

```
T ::= a | b « T » | c « T »
```

and the non-primitive recursive function

```
g : T → ℕ
─────────────────────────────
∀ t : T •
    g a = 0
    g(b a) = 1
    g(c a) = 2
    g(b(b t)) = g(c t) + (g t)
    g(b(c t)) = g t
    g(c(c t)) = g(c t) + 3
    g(c(b t)) = 2
```

A primitive recursive function `g1` is constructed. The idea behind the construction is that

```
g1 t = (g t, g(b t), g(c t))
```

Using a similar method as in the last example, the formal definition of `g1` is

```
h1 == λ x, y, z : ℕ • (y, z + x, 2)
h2 == λ x, y, z : ℕ • (z, x, z + 3)
```

```
g1 : T → (ℕ × ℕ × ℕ)
─────────────────────────
∀ t : T •
    g1 a = (0, 1, 2)
    g1(b t) = h1(g1 t)
    g1(c t) = h2(g1 t)
```

The original function `g` can then be written

```
g t = fst3(g1 t)
```

where

```
fst3 == λ x, y, z : ℕ • x
```

Δ

5.2 Strengths and Weaknesses

The technique described helps in checking the consistency of a Z specification. The technique of rewriting a non-primitive recursive function in terms of a primitive one can be automated. A disadvantage of the technique is that the definition of the non-primitive recursive function must be rewritten in an obscure way. This obscure specification helps to prove the existence of the function. This is another example of how a more obscure specification can lead to an easier proof. This conflict between specification and proof is discussed in section 7.

6 Reasoning with Schemas

In Z, it is useful to reason at the schema level, without getting lost in a mass of low level predicates. A schema can appear as a set, a predicate, in a schema expression or as an inclusion. In its last role it provides its main expressive power, but also the greatest difficulty for reasoning as it is not obvious when to stop expanding schema inclusions.

This topic will not be dealt with here except to note that the main new feature introduced by a schema is its signature, so proof rules need to be concerned with the effect of binding and variable introduction. In the case of schemas as predicates and schema expressions it is reasonable to assume that typechecking has discharged the scope and type consistency obligations, in which case schemas can be handled as predicates and simple laws given for the schema operators. The propositional schema operators ($\wedge$, $\vee$ etc) obey all the usual laws of propositional logic. If S, T and U are schemas then

$$S \wedge (T \vee U) = (S \wedge T) \vee (S \wedge U)$$

Similar laws can be given for the other schema operators, and as an illustration of the laws useful for reasoning at the schema level, the laws appropriate for preconditions will be given.

6.1 Laws For Calculating Preconditions

When describing an operation in Z, it is useful to know when the specified operation can be used. In particular, for consistency, it is important to check that the domain of applicability is not empty. When using a schema to describe an operation, the applicability of the operation is described by its *precondition*. This section contains useful laws for calculating the precondition. The laws presented here have been rigorously proven by hand although they should be formally proved using the semantics of Z [Brien]. The work extends that in [Gilmore] to consider all the schema operators in [Spivey] and [McMorran].

Some of these rules have side conditions which are expressed in square brackets directly after the rule. The following notation is used to express these side conditions. If S is a schema, let bS denote the set of "before" components of S (the undashed variables and the inputs), and aS denote the set of "after" components of S (the dashed variables and the outputs). Also let $!S$ denote just the outputs. The next example illustrates this notation.

Example 18

If S is the schema

```
┌ S ───────────────
│ x, x′, z! : ℤ
│ y, y′, w? : ℕ
├─────────────
│ ...
└────────────────
```

then

$$bS = \{x,\ y,\ w?\} \qquad aS = \{x',\ y',\ z!\} \qquad !S = \{z!\}$$

Δ

Also, the rules are general in the sense that there is no necessity for a schema to be of the form

```
┌ S ──────────
│ ΔState
├─────────
│ ...
└────────────
```

that is, where every undashed variable has a dashed counterpart. The schema could have an undashed variable with no dashed counterpart, or vice versa.The rules are listed below under the particular schema operation involved.

Disjunction

$$pre(S \vee T) = pre(S) \vee pre(T)$$

Conjunction

$$pre(S \wedge T) = pre(S) \wedge pre(T) \qquad [aS \cap aT = \{\}]$$

Implication

$$pre(S \Rightarrow T) = pre(\neg S) \vee pre(T)$$

Equivalence

$$pre(S \Leftrightarrow T) = pre(S \Rightarrow T) \wedge pre(T \Rightarrow S) \qquad [aS \cap aT = \{\}]$$

This is a very interesting law. At first sight, this law appears to follow easily from the law for conjunction above, since

$$S \Leftrightarrow T \ = \ S \Rightarrow T \wedge T \Rightarrow S$$

But in order to use that law its side condition must be satisfied for the schemas $S \Rightarrow T$ and $T \Rightarrow S$. But these two schemas have the same signature (formed from merging the signatures of S and T). Thus the side condition is not satisfied, and so the law can not be used. The above law for schema equivalence must be derived by other means. It can be used with the law for implication to obtain an expression for $pre(S \Leftrightarrow T)$ in terms of the preconditions of S, T, $\neg S$ and $\neg T$.

Negation

There is no useful law for $pre(\neg S)$. At first glance, one might have expected $pre(\neg S)$ to be equal to $\neg(pre\ S)$ but this is not the case. A counterexample follows.

Example 19

Let S be the schema

```
┌ S ─────────
│ x, x' : ℤ
├────────────
│ x = x'
└────────────
```

Both $pre\ S$ and $pre(\neg S)$ are the schema

```
┌────────────
│ x : ℤ
├────────────
│ true
└────────────
```

Thus $\neg(pre\ S)$ is the schema

```
┌────────────
│ x : ℤ
├────────────
│ false
└────────────
```

which is very different from $pre(\neg S)$. Δ

The reason why $pre(\neg S)$ is not equal to $\neg(pre\ S)$ is as follows. A precondition describes the *applicability* of an operation rather than describing the operation itself. Thus it is possible for two schemas to have very different properties, but the same precondition (as seen for the two schemas S and $\neg S$ in the above example).

Overriding

$$pre(S \oplus T) = pre(S) \vee pre(T)$$

This law is similar to the law for disjunction because $S \oplus T$ is like $S \vee T$ but with the *priority* given to T. This priority does not affect the precondition.

Composition

$$pre(S ⨾ T) = pre(S ⨾ pre(T)) \qquad [!S \cap !T = \{\}]$$

Recall that for $S ⨾ T$ to be defined the set of dashed variables of S must equal the set of undashed variables of T.

Piping

$$pre(S \gg T) = pre(S \gg pre(T)) \qquad [(aS - pipe) \cap aT = \{\}]$$

where $pipe$ is the set of outputs of S that are piped into T. This law is similar to that for composition, but with ⨾ replaced by ». This is because the operation of piping is similar to composition, but with inputs and outputs forming the interface between S and T, rather than dashed and undashed variables.

Projection

$$pre(S \upharpoonright T) = (pre(S) \wedge pre(T)) \setminus (bS - bT) \qquad [aS \cap aT = \{\}]$$

where $bS - bT$ is the list of variables that appear in bS but not in bT.

Restriction

$$pre\ [S \mid P] = [pre(S) \mid P] \qquad [P \text{ is a predicate over } bS]$$

Quantification

$$pre(\exists D \mid P \bullet S) = (pre\ [S \mid P]) \setminus (D - aS)$$

where $(D - aS)$ is the list of variables that appear in the declaration D but not in aS. There are no useful rules for $pre(\exists_1 D \mid P \bullet S)$ and $pre(\forall D \mid P \bullet S)$.

Hiding

$$pre(S \setminus (v)) = (pre\ S) \setminus (v - aS)$$

where v is a list of variables that appear in S. The expression $(v - aS)$ is the list of variables that appear in v but not in aS.

The next example illustrates the use of the above rules.

Example 20

Let $Sub10$ and $Sub4$ be schemas describing the operations of subtracting 10 and 4 respectively.

```
┌─ Sub10 ──────────┐      ┌─ Sub4 ──────────┐
│ a?, b! : ℕ       │      │ b?, c! : ℕ      │
├──────────────────┤      ├─────────────────┤
│ b! = a? - 10     │      │ c! = b? - 4     │
└──────────────────┘      └─────────────────┘
```

The laws can be used to calculate the precondition of the schema $pre(Sub10 \gg Sub4)$. Using the law for piping, then

$$pre(Sub10 \gg Sub4) = pre(Sub10 \gg pre(Sub4)) \qquad (A)$$

Now $pre(Sub4)$ is the schema

```
┌─────────────────────────────┐
│ b? : ℕ                      │
├─────────────────────────────┤
│ ∃ c! : ℕ • c! = b? - 4      │
└─────────────────────────────┘
```

which simplifies to

```
┌──────────┐
│ b? : ℕ   │
├──────────┤
│ b? ≥ 4   │
└──────────┘
```

The schema $Sub10 \gg pre(Sub4)$ is therefore

$$a? : \mathbb{N}$$

$$\exists\ b : \mathbb{N} \bullet b = a? - 10 \land b \geq 4$$

which simplifies to

$$a? : \mathbb{N}$$

$$a? \geq 14$$

(B)

Using A and the fact that schema B has no after components, $pre(Sub10 \gg Sub4)$ is equal to schema B. The calculated precondition is as expected, since $Sub10 \gg Sub4$ represents the operation of subtracting 14. Δ

6.2 Strengths and Weaknesses

The rules allow reasoning at the schema level. Also if an operation has been specified in terms of basic operations, using many schema operators, then the rules can be used to calculate its precondition from the preconditions of the basic operations. This means that the preconditions of the basic operations can be reused, thus avoiding duplication of effort. The rules could be used by a tool builder to populate a library, or be used to produce an automatic precondition calculator.

7 The Conflict Between Specification and Proof

It is sometimes easier to prove a theorem starting from one specification than from another. Although it can be easier to start from one specification, this can be at the expense of clarity. If the specification is hard to understand, then it is unclear exactly *what* has been specified. It is unclear if the specification captures the author's intentions. This in turn makes it unclear exactly what has been proved. The whole point in carrying out proof is to increase confidence in the system being developed. If it is unclear exactly what has been proved then the proof is pointless. The next example illustrates this.

Example 21

Consider the *highest common factor (hcf)* of two non-zero natural numbers. Recall that the hcf of two numbers is the *largest* number which divides both. For example the hcf of 6 and 9 is 3. Consider the following specification of hcf (which is basically Euclid's algorithm for calculating the hcf).

$$hcf : (\mathbb{N}_1 \times \mathbb{N}_1) \rightarrow \mathbb{N}_1$$

$$\begin{aligned} &\forall\ x, y : \mathbb{N}_1 \bullet \\ &\quad hcf(x,x) = x \\ &\quad hcf(x + y,\ y) = hcf(x,y) \\ &\quad hcf(x,\ x + y) = hcf(x,y) \end{aligned}$$

With this specification theorems involving hcf are easy to prove. The reason is that the specification lends itself to proof by induction. The particular induction schema required is generated by section 3 (induction), and is

$$\begin{array}{l}
\vdash \forall\ x : \mathbb{N}_1 \bullet P(x,x) \land \\
\quad \forall\ x,\ y : \mathbb{N}_1 \bullet P(x,y) \Rightarrow P(x + y,\ y) \land \\
\quad \forall\ x,\ y : \mathbb{N}_1 \bullet P(x,y) \Rightarrow P(x,\ x + y) \land \\
\quad \Rightarrow \\
\quad \forall\ x,\ y : \mathbb{N}_1 \bullet P(x,y)
\end{array}$$

The problem is that the above specification and corresponding induction schema are hard to understand. It is not obvious that the specification captures the meaning of hcf. Now consider the following alternative specification of hcf.

$$\begin{array}{|l}
hcf : (\mathbb{N}_1 \times \mathbb{N}_1) \rightarrow \mathbb{N}_1 \\
\hline
\forall\ x,\ y,\ z : \mathbb{N}_1 \bullet \\
\quad hcf(x,y)\ divides\ x \\
\quad hcf(x,y)\ divides\ y \\
\quad z\ divides\ x \land z\ divides\ y \Rightarrow z\ divides\ hcf(x,y)
\end{array}$$

where $m\ divides\ n$ if and only if n/m is a natural number. Proving theorems is more difficult using this specification, but the specification is easier to understand. For example, the first two axioms state that the hcf of two numbers divides both numbers, while the third axiom states that the hcf is the largest such number. It is much easier to see that this second specification captures the meaning of hcf. Δ

As mentioned earlier, if a specification is hard to understand then it is unclear whether it captures the author's intentions. There is a real danger that it specifies something very different. It is possible for two specifications to be quite similar, but specify very different objects. Thus if a mistake has been made in an unclear specification it will be hard to notice, and something very different will be specified. This in turn means that a different theorem to the intended theorem is being proved. The next example illustrates this point.

Example 22

Suppose a mistake has been made in the unclear specification of hcf in example 21, giving

$$\begin{array}{|l}
hcf : (\mathbb{N}_1 \times \mathbb{N}_1) \rightarrow \mathbb{N}_1 \\
\hline
\forall\ x,\ y : \mathbb{N}_1 \bullet \\
\quad hcf(x,x) = x \\
\quad hcf(x + y,\ y) = hcf(x,y) \\
\quad hcf(x,\ x + y) = x
\end{array}$$

(the right hand side of the third axiom is different). This specification does not specify highest common factor at all, it specifies a modulo function. For any two numbers x and y the above function repeatedly subtracts y from x until it is in the range $1..y$. Such a mistake is more difficult to find in an unclear specification. Thus theorems proved using the above specification are not theorems about highest common factor at all, they are theorems about a modulo function. Δ

Other examples of this conflict between specification and proof have appeared in earlier sections of this paper. In section 3 (induction), higher order functions were used to make theorem proving easier. But these functions made the specification and proof harder to understand. In section 5 (consistency of Z specifications), some recursive functions were rewritten to make it easier to prove their existence. The new versions were harder to understand.

[Gravell] and [Macdonald] present techniques for writing a specification so that it is easier to understand. For example [Gravell] uses the phrase *syntactic gap* to mean the difference between the English and the mathematics in a specification. The idea is to minimise the syntactic gap. There is of course an assumption here that a person's English is intuitive and easy to understand.

There appears to be a conflict between specification and proof. But perhaps this conflict can be turned to advantage. Why not have both the clear specification for understanding, and the unclear specification for proof? The equivalence of these two specifications must be proved: firstly to ensure that properties of the "proof" specification are indeed properties of the "clear" specification; secondly to ensure that all properties of the "clear" specification can be derived from the "proof" specification. In example 21, the equivalence between the two specifications can be proved as follows. The induction schema given in example 21 can be used to show that the "proof" specification implies the "clear" specification. To show the converse, the lemma

$$\vdash \forall\ x, y : \mathbb{N}_1 \bullet (x \text{ divides } y \land y \text{ divides } x) \Rightarrow (x = y)$$

can be used to link the relation divides in the "clear" specification to equality in the "proof" specification. In general, perhaps equivalence could be proved by a trusted transformation approach. This approach would be similar to the technique of program transformations.

8 Conclusions

This paper has presented a number of useful techniques for proving theorems in Z. Some of the techniques give the user an advantage when proving theorems. For example, generalization can make a theorem easier to prove, as well as making it more useful. Also the technique for choosing the right induction schema and induction variable, and the technique for finishing the proof, help to guide the user. One of the techniques for generalizing a theorem is to use higher order functions. The identification of higher order functions and theorems involving them presents an opportunity for further work in this area.

The paper has discussed the conflict between specification and proof. Sometimes a less intuitive specification can be better for proof, but harder to understand. This conflict can perhaps be turned to advantage by having both specifications. One specification would be for understanding, the other for proof. The equivalence of these two specifications must be proved: firstly to ensure that properties of the "proof" specification are indeed properties of the "clear" specification; secondly to ensure that all properties of the "clear" specification can be derived from the "proof" specification. Proving the equivalence between these two specifications is an area for further work. Perhaps equivalence could be proved by a trusted transformation approach. This approach would be similar to the technique of program transformations.

The paper has also discussed the important topic of consistency of Z specifications. A technique is presented that can check the consistency of certain non-primitive recursive functions defined over a recursive free type. This technique should be extended to cover a wider class of such functions and free types. This is an area where further work is needed. The laws for calculating preconditions are also useful when checking consistency. They can be used to check that a specified operation is actually possible.

The paper has identified some useful theorems for a theorem library. Examples of such theorems are those generated by generalization. These theorems could be reused for many different problems. Also, the laws for calculating preconditions would be a useful addition to such a library.

Some of the techniques could be automated. The resulting proof tool would be tailored to the user, rather than the reverse which is currently the case. For example, if window inference is automated this will give users the chance to do on a machine what they do on paper. Using the techniques for generalization, a tool could automatically generalize a theorem by replacing every occurrence of a subterm by a variable. If the tool was successful in proving the generalized theorem it could automatically replace the variable with the original subterm, thus proving the original theorem. The technique for choosing the right induction schema and induction variable, and the technique for finishing the proof, would mean a tool could automatically attempt to prove a theorem by induction. The laws for calculating preconditions could be used to produce an automatic precondition calculator.

References

[Bird] R. Bird and P. Wadler, "Introduction to Functional Programming", Prentice Hall, 1988.

[Boyer] R. S. Boyer and J. S. Moore, "A Computational Logic", Academic Press, 1979.

[Brien] S. M. Brien, P. J. Lupton, J. E. Nicholls and I. H. Sorensen, "Z Base Standard (Version 0.4)", ZIP Project, Programming Research Group, Oxford University, December 1991.

[Brumfitt] J. Brumfitt, "Metamorph - A Formal Methods Toolkit for Designing Digital Hardware", Logica Cambridge Limited, 1991.

[Bundy a] A. Bundy, A. Smaill and J. Hesketh, "Turning Eureka Steps into Calculations in Automatic Program Synthesis", Department of Artificial Intelligence, University of Edinburgh, Research paper number 448, 1990.

[Bundy b] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill and A. Stevens, "A Rational Reconstruction and Extension of Recursion Analysis", Department of Artificial Intelligence, University of Edinburgh, Research paper number 419, 1989.

[Bundy c] A. Bundy, F. van Harmelen and A. Smaill, "Extensions to the Rippling-Out Tactic for Guiding Inductive Proofs", Department of Artificial Intelligence, University of Edinburgh, Research paper number 459, 1990.

[Gilmore] S. Gilmore, "Correctness-Oriented Approaches to Software Development", University of Edinburgh (Department of Computer Science), Report no: CST-76-91(Thesis), April 1991.

[Gravell] A. Gravell, "What is a Good Formal Specification?", Proceedings of the Z User Workshop 1990, Springer-Verlag.

[Grundy] J. Grundy, "Window Inference in the HOL System", University of Cambridge Computer Laboratory, 1991.

[Hayes] I. Hayes (ed.), "Specification Case Studies", Prentice-Hall, 1987.

[King] S. King, I. H. Sorensen and J. Woodcock, "Z: Grammar and Concrete and Abstract Syntaxes (Version 2.0)", Programming Research Group, Oxford University, Technical Monograph PRG-68, July 1988.

[Macdonald] R. Macdonald, "Z Usage and Abusage", RSRE Report 91003, February 1991.

[McMorran] M.A. McMorran and J.E. Nicholls, "Z User Manual", IBM Hursley Park, Technical report no: TR12.274, July 1989.

[Morgan] C. Morgan, "Programming from Specifications", Prentice Hall, 1990.

[Smith a] A. Smith, "On Recursive Free Types in Z", RSRE Report 91028, August 1991.

[Smith b] A. Smith, "Which Theorem Prover? (A Survey of Four Theorem Provers)", RSRE memorandum 4430, October 1990.

[Spivey] J. M. Spivey, "The Z Notation", Prentice Hall, 1989.

[Woodcock] J. C. P. Woodcock and S. M. Brien, "A Logic for Z", Proceedings of the Z User Workshop 1991, Springer-Verlag.

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) ....

Overall security classification of sheet ..........UNCLASSIFIED..........

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification, eg (R), (C) or (S).

Originators Reference/Report No. REPORT 92006	Month MARCH	Year 1992

Originators Name and Location

DRA, ST ANDREWS ROAD
MALVERN, WORCS WR14 3PS

Monitoring Agency Name and Location

Title

PROOF OF THEOREMS IN Z

Report Security Classification UNCLASSIFIED	Title Classification (U, R, C or S) U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period
Project Number	Other References

Authors SMITH, A	Pagination and Ref 27

Abstract

This paper is concerned with user aspects of proof in the specification language Z. A number of techniques for proving theorems in Z are presented. Some of the techniques are not new, being drawn from the area of general mathematical theorem proving, but are brought together in one place and applied specifically to Z. The techniques have been written so that they can be understood and applied by a person carrying out a pen and paper proof. Some of the techniques may be automated, and would result in a proof tool tailored to the needs of the user; currently a user has to tailor a proof to fit a tool.

Abstract Classification (U, R, C or S)

U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)

UNLIMITED