ProofPower

Z TUTORIAL

PPTex-2.9.1w2.rda.110727

Information on the current status of ProofPower is available on the World-Wide Web, at URL:

`http://www.lemma-one.demon.co.uk/ProofPower/index.html`

This document is published by:

Lemma 1 Ltd.
2nd Floor
31A Chain Street
Reading
Berkshire
UK
RG1 2HX
e-mail: `pp@lemma-one.com`

PPTex-2.9.1w2.rda.110727 13:54:16 27/07/2011

CONTENTS

ABOUT THIS PUBLICATION

0.1 Purpose

This document is a tutorial on the use of ProofPower for formal reasoning about specifications in the Z language.

The objectives of this tutorial are:

- to describe the basic principles and concepts underlying support for Z in ProofPower
- to enable the student to write simple specifications and undertake elementary proofs in Z using ProofPower
- to enable the student to make effective use of the reference documentation

0.2 Readership

This document is intended to be among the first to be read by new users of ProofPower wishing to use the specification language Z.

0.3 Related Publications

A bibliography is given at the end of this document. Publications relating specifically to ProofPower include:

1. ProofPower Tutorial *[6]*;
2. ProofPower HOL Tutorial Notes *[8]*;
3. ProofPower Description Manual *[7]*;
4. ProofPower Reference Manual *[9]*;
5. ProofPower Document Preparation *[5]*.

0.4 Area Covered

This tutorial is an introductory course on proof in Z using ProofPower which explains how Proof-Power may be used for checking specifications and conducting proofs in Z. After working through this tutorial, the reader should be capable of using ProofPower with Z for simple tasks, and should be able to make effective use of the ProofPower documentation where necessary for approaching more difficult problems.

This tutorial supplements the ProofPower *Tutorial* [6] and the ProofPower *HOL Tutorial Notes* [8] with material relating to the Z language. The tutorial should enable users of ProofPower to become familiar with the following subjects:

1. The dialect of Z supported by the ProofPower system (which we call ProofPower-Z) and its manipulation via the metalanguage.
2. Forward proof and derived rules of inference for Z.
3. Goal directed proof, and tactics and tacticals for Z.

0.5 Prerequisites

The tutorial is not intended as an introduction to formal methods or to the language Z.

We assume a working knowledge of:

- Z as a specification language.
- Use of ProofPower as used for specification and proof in HOL.

  Ideally a reader who has not attended a course should read ProofPower *Tutorial* [6] and work through the exercises in ProofPower *HOL Tutorial Notes* [8] before beginning this tutorial.

0.6 How To Use This Tutorial

It is intended that this document will allow ProofPower users who have not attended the ProofPower Z course to work through the course material independently.

The best way to learn about ProofPower is by doing things with it.

The two kinds of things which you can do while working through these tutorial notes are:

- Do the set exercises.

  To make it easier to do the exercises the installation procedure for ProofPower results in the establishment of a ProofPower database called '`example_zed`', which contains the results of executing all of this tutorial document except the material in Chapter 8 where the solutions to the exercises may be found. To do the exercises the reader should attempt to set up his own version of the solutions document ('`usr011S.doc`') by working interactively in a ProofPower session using a copy of database '`example_zed`'.

  This is best done using a writeable copy of the database so that you can save the database after completing some of the exercises and then resume from that point later. This can be done as follows:

  ```
  cp $PPHOME/db/example_zed.polydb .
  chmod +w example_zed.polydb
  ```

Here, $*PPHOME* is an environment variable which should be set up to be the pathname of the directory in which ProofPower has been installed.

If you wish to use the X interface for ProofPower, `xpp`, you can now start your ProofPower session by starting X if necessary and then giving the UNIX command:

```
xpp -d example_zed
```

- Replay the illustrative material.

  This is best done using the source of the tutorial OHP transparencies, `usr023_slides.doc`. It can be done running on database `example_zed`, though you will find that some of the material will be rejected because definitions have already been made in this database. Alternatively you can work from a clean database (but then you may find problems if you miss out any of the material). E.g., to work on the existing database using `xpp`, you might use the command:

```
xpp -f $PPHOME/doc/usr023_slides.doc -d example_zed
```

Source documents are supplied for the exercises (Chapter 7 in file `usr011X.doc`) and solutions (Chapter 8 in file `usr011S.doc`).

It is best to build up your own document containing your solutions to the exercises and any experiments you might wish to undertake. ProofPower does not keep a record of what you type into it, and so if you want to do it again you will need to keep a copy of your script.

0.7 Acknowledgements

ICL gratefully acknowledges its debt to the many researchers (both academic and industrial) who have provided intellectual capital on which ICL has drawn in the development of ProofPower.

We are particularly indebted to Mike Gordon of The University of Cambridge, for his leading role in some of the research on which the development of ProofPower has built, and for his positive attitude towards industrial exploitation of his work.

The ProofPower system is a proof tool for Higher Order Logic which builds upon ideas arising from research carried out at the Universities of Cambridge and Edinburgh, and elsewhere. In particular the logic supported by the system is (at an abstract level) identical to that implemented in the Cambridge HOL system [1], and the paradigm adopted for implementation of proof support for the language follows that adopted by Cambridge HOL, originating with the LCF system developed at Edinburgh [2]. The functional language 'Standard ML' used both for the implementation and as an interactive metalanguage for proof development, originates in work at Edinburgh, and has been developed to its present state by an international group of academic and industrial researchers. The implementation of Standard ML on which ProofPower is based was itself originally implemented by David Matthews at the University of Cambridge, and is now commercially marketed by Abstract Hardware Limited.

The ProofPower system also supports specification and proof in the Z language, developed at the University of Oxford. We are therefore also indebted to the research at Oxford (and elsewhere) which has contributed to the development of the Z language.

INTRODUCTION TO ProofPower-Z

1.1 Using ProofPower for Z

1.1.1 Setting Up

A ProofPower system issued with the ProofPower-Z option will be provided with more than one database, not all of which will contain the Z support facilities. These include a database called 'zed' (full name 'sun4zed.db') will be available supporting the ProofPower-Z option, and a database called 'example_zed' which contains the results of executing the formal material in Chapters 1-7 of this tutorial. This includes Chapter 7 containing the exercises but not Chapter 8 which contains the solutions to the exercises.

For undertaking application work with ProofPower it is first necessary to set up a new database as a child of the issued database 'pp_zed'.

This is done using 'pp_make_database' as follows:

```
pp_make_database -p installdir/sun4bin/pp_zed zed
```

Where *installdir* is the pathname of the directory in which ProofPower has been installed and *zed* is the name to be given to the new database, for which the user may substitute a name of his choice.

ProofPower can then be invoked by the UNIX command:

```
pp -d databasename
```

Where *databasename* is the name given by the user to the database created in the previous step (*zed* if the database was created with the command as shown above).

For the purpose of undertaking the exercises in this tutorial a special database (called 'example_zed') is constructed during the ProofPower installation procedure containing material which has been pre-loaded from this document. The user should make his own copy of this database with write access permission and use this database for doing the tutorial exercises.

Apart from selecting or setting up a database including the Z support facilities, entering and leaving ProofPower for work in Z is the same as for work in HOL.

1.1.2 Formal Material in this Document

In this document the behaviour of ProofPower is frequently illustrated by showing how the system responds to various inputs. In these illustrations a line input to ProofPower is shown by a vertical line on the left, with 'SML' at the head of the line, thus:

```
SML
│
```

'SML' is an acronym for *standard ML*, the 'meta-language' in which the user of ProofPower communicates with the system.

The output from ProofPower usually displayed on the console in response to such an input will be marked by a vertical line on the left, thus:

ProofPower output
```
|
```

Sometimes we omit parts of the output, and supply '...' to mark the point of such an omission.

1.1.3 Setting the Context

A ProofPower database supporting Z will also support HOL. Though the system does support mixed language working, i.e. working at the same time with both of these languages, it is usual to work normally with a single language. To do this smoothly the context needs to be set up correctly for that language.

The main aspect of context which is relevant is the current *theory*. Associated with each theory is a language code, and the language code of the current theory influences the behaviour of the system.

The best place in the theory hierarchy to do work in Z is in a descendant of the theories which provide the Z ToolKit. The theory *z_library* should therefore be a parent of any theories which the user creates for work in Z.

SML
```
open_theory "z_library";
new_theory "usr011";
```

Theories inherit by default the language of their parent, so any theory created while *z_library* is the current theory will have Z as its language.

A second important aspect of context is the *proof context*. Many of the facilities provided by ProofPower work with either HOL or Z by access to information in the current proof context which may be set by a call to *set_pc*. A proof context should therefore be chosen which supports Z. Suitable candidates are *z_language*, which incorporates a knowledge of the Z language but not of the Z ToolKit and *z_library*, which includes knowledge of both.

SML
```
set_pc "z_library";
```

A further element of context of which the user should be aware is the current subgoal, when the subgoal package has been invoked. The subgoal package keeps a record, known as the current 'typing context', of the types of the variables which occur freely in the current subgoal. This context is referred to when type-checking terms entered through the HOL or Z parsers. If a subgoal relating to an incomplete proof is left on the goal stack, then this may cause terms entered into the system to fail to type-check if they may use of variables with types which disagree with the usage of the same variables in the subgoal.

To clear the goal stack the command:

SML
```
repeat drop_main_goal;
```

may be used.

The command *print_status* will display information about the current context, e.g.:

SML
```
print_status();
```

ProofPower output
```
Current theory name: usr011;
Current proof context name(s): [z_library];
The subgoal package is not in use;
There is no current goal.
val it = () : unit
```

THE Z LANGUAGE IN ProofPower

2.1 Introduction

ProofPower-Z is a dialect of Z which falls somewhere between that defined in the ZRM [3][4] and that which will ultimately be defined as an ISO standard. The dialect is based on a proposal made to the Z standard review committee in March 1992, some elements of which have since been incorporated into the draft standard [10].

In due course we hope to bring ProofPower-Z closely in line with the ISO standard, and in particular we would like to provide an option which will fully check specifications against that standard. For the purpose of conducting proofs however, we anticipate that extensions will continue to be available.

2.1.1 The Structure of this Tutorial

In this and following chapters the ProofPower-Z language and its proof support is described systematically but informally.

The description is organised around the abstract syntax for ProofPower-Z. One section or subsection is devoted to each constructor in the abstract syntax. Each of these subsections covers the abstract and concrete syntax, the semantics and proof support.

The emphasis is on illustration and example rather than on formal description.

This introductory section also addresses some topics orthogonal to the following sections.

2.1.1.1 Paragraphs

Usage of the term “Paragraph” follows the literature on Z. Paragraphs form the top level constituents of Z specifications, and correspond to declarations, definitions, or constant specifications in HOL. These effects in HOL are often not available as object language syntactic contructs, but are effected by the use of procedure calls in the metalanguage ProofPower-ML.

The effect of processing a paragraph in Z therefore corresponds most closely to that of executing a metalanguage procedure in HOL. Paragraphs do not have values, but are evaluated for their side effects, which are recorded in the ProofPower theory hierarchy.

Paragraphs in Z are formed using various other syntactic categories, including *predicates*, *expressions*, *schema-expressions*, and *declarations*. In HOL the syntactic categories having similar roles are *types* and *terms*.

2.1.1.2 Z Terms

In mapping Z into HOL all of the syntactic categories for Z except paragraphs are mapped into HOL terms. It is therefore convenient at times to talk of *ZTerms*, even though ‘term’ is not used in the literature on Z, meaning a HOL term in the image of the Z to HOL mapping.

Furthermore, when dealing with Z in ProofPower at an abstract or computational level, it is convenient to regard the syntactic categories *predicate*, *expression*, and *schemaexpression* as being combined into the single category *term*. This is reflected by the provision of a ProofPower-ML datatype called *Z_TERM* which reveals the abstract structure of the Z language in these terms.

Two functions are supplied with ProofPower which enable *TERM*s to be constructed from their components or broken up into their components via the datatype *Z_TERM*:

SML
```
mk_z_term : Z_TERM -> TERM;
dest_z_term : TERM -> Z_TERM;
```

e.g., to contruct the Z predicate ⌜Z true⌝:

SML
```
val term_true = mk_z_term ZTrue;
```

ProofPower output
```
val term_true = ⌜Z true⌝ : TERM
```

To construct an implication:

SML
```
val term_imp = mk_z_term (Z⇒ (term_true, term_true));
```

ProofPower output
```
val term_imp = ⌜Z true ⇒ true⌝ : TERM
```

To display the kind of Z construct represented by a HOL term, and the constituents of the construct:

SML
```
dest_z_term term_imp;
```

ProofPower output
```
val it = Z⇒ (⌜Z true⌝, ⌜Z true⌝) : Z_TERM
```

SML
```
dest_z_term term_true;
```

ProofPower output
```
val it = ZTrue : Z_TERM
```

To bind the values of the constituents of a Z TERM to ML names use a pattern matching ML binding with *dest_z_term*:

SML
```
val (Z⇒ (ante, concl)) = dest_z_term term_imp;
```

ProofPower output
```
val ante = ⌜Z true⌝ : TERM    val concl = ⌜Z true⌝ : TERM
```

The datatype *Z_TERM* is described in detail below, and its structure is used as the basis for the explanation of the Z language support in ProofPower.

2.1.1.3 Z Types

Z is a typed language, and in ProofPower, the types of Z are mapped into types in HOL. Types in Z are not provided with a concrete syntax, but a Z type may be described using a Z expression which denotes a set co-extensive with the type. The function:

SML
```
|z_type_of : TERM -> TERM;
```

when given the HOL term representing a Z expression, returns a HOL term representing a Z expression which denotes the set of all elements of the type of the Z expression.

e.g.:

SML
```
|z_type_of ⓩ{x,y:ℕ | x > y}⌝;
```

ProofPower output
```
|val it = ⓩℤ ↔ ℤ⌝ : TERM
```

z_type_of also works on Z schema expressions and predicates:

SML
```
|z_type_of ⓩ[x,y:ℕ | x > y]⌝;
```

ProofPower output
```
|val it = ⓩℙ [x, y : ℤ]⌝ : TERM
```

SML
```
|z_type_of ⓩx > y⌝;
```

ProofPower output
```
|val it = ⓩ𝔹⌝ : TERM
```

2.1.1.4 Quotation

As illustrated above, for many of the syntactic categories in Z, and for terms in HOL, quotation facilities are provided in ProofPower-ML which permit phrases to be written in the concrete syntax of Z or HOL. Such quotations, whether in HOL or in Z, evaluate (after syntax checking and type inference) to yield HOL terms which form the internal representation of the construct. When such terms are displayed a pretty printer is automatically invoked, which will use an appropriate concrete syntax for displaying the term. The pretty printer is able to determine which concrete syntax is more appropriate for displaying a term, that of Z or that of HOL.

Both in quotations and in the formatting of terms for display, mixed languages are supported. A term may have subterms in distinct languages.

The three languages of concern for this tutorial are ProofPower-ML, ProofPower-HOL, and ProofPower-Z. They are quoted using the quotation characters ' ⓜ', ' ⌜' and ' ⓩ' respectively. Quotations are terminated by the character ' ⌝', irrespective of language, and may be nested. Quotations in ML are sometimes known as "anti-quotations". ML quotations are compiled and evaluated to yield a value of type TERM. In addition HOL TYPEs may be quoted using '⌜:', and may be entered as ML expressions of type TYPE using the character sequence:

```
|        ⌜↘SML:↕
```

which is printed as ' ⌜$_{SML:}$'.

2.1.1.5 Type Inference and Casts

For the purposes of conducting proofs in Z using ProofPower it is often necessary to enter into the system fragments of Z in which free variables occur. These are sometimes necessary, for example, when providing a witness for an existence proof.

When type inference takes place on a term entered with free variables the following rules apply:

- Variables with names corresponding to the names of previously declared global variables are treated as occurrences of those global variables. They are required to have types which are instances of the type of the global variable. Z global variables are represented in HOL as constants.
- The subgoal package used for goal oriented or backward proof maintains a type inference context in which the types of all the free variables in the current subgoal are held (unless flag *subgoal_package_ti_context* is set false). Free variables in terms entered by Z quotation will be assigned the type given in the type inference context if there is one, forcing them to match free variables in the current subgoal.
- If a free variable is not a global variable and does not appear in the type inference context, then its type will be inferred from its context in the quotation if possible. Otherwise a new type variable will be introduced and used for the type of the variable.
- Type variables introduced during type inference cannot be constrained to range over tuple types or binding types in the present implementation, and therefore in places where a value of binding or tuple type is required (e.g., before '.') the type inferrer will report an error unless the type of the tuple or binding is apparent.

  In the following 'x' and 'y' are assigned type variables without demur:

  SML
  ```
  ⓩ(x,y).2⌝;
  ```

  ProofPower output
  ```
  val it = ⓩ(x, y).2⌝ : TERM
  ```

  Whereas in this example, 't' must be assigned a tuple type, a type variable will not suffice:

  SML
  ```
  ⓩt.2⌝;
  ```

  ProofPower output
  ```
  Type error in ⓩt . 2⌝
  In a term of the form ⓩT.number⌝, T must be a tuple
  The following sub−term is not a tuple
     ⓩt:'a⌝
  Exception− Fail * Type error [Z−Parser.62000] * raised
  ```

- To overcome the above problem the user may supply, when necessary, additional information in the form of type casts.

  For the purpose of applying type casts the infix operator '_$\overset{\oplus}{\oplus}$_' may be used to give guidance to the type inferrer on the type of a construct.

The constant '$\_\overset{\oplus}{\oplus}\_$' is defined as follows:

```
Z
╒[X]══════════════════════════════════════
│    _⊕⊕_ : (X × ℙ X) → X
├──────────────────────
│    ∀ x:X; y: ℙ X• x ⊕⊕ y = x
└──────────────────────────────────────────
```

When used as an infix operator '$\overset{\oplus}{\oplus}$' forces its left hand operand to be an expression, whose type is that of the elements of the expression which is the right hand operand. However, after type inference is complete, the term is constructed as if the constant and its right hand operand had not been present. If it is required to generate a term which does include this constant then its fixity status may be locally suspended by writing '$(\_\overset{\oplus}{\oplus}\_)$' as a prefix operator. In this case the constant will be treated normally.

2.1.1.6 Elided Actual Generic Parameters

Generic global variables in Z are set-generic rather than polymorphic. This means that when instantiated for use, they are instantiated with values which are sets rather than types.

When actual generic parameters are elided, the type of the parameter can usually be established by type inference, but this leaves open the choice of a particular set of that type for the actual parameter.

The choice made by the system depends upon whether the occurrence of the generic variable is in the paragraph defining the variable or in a later paragraph or term. In the defining paragraph, generic parameters must be omitted, and are supplied by the system as identical with the formal parameters.

This may be seen by viewing the generic predicate subsequently extracted from such a paragraph:

```
SML
│z_get_specⓩ⋂⌝;
```

```
ProofPower output
│val it = ⊢ [X]({⋃[X], ⋂[X]} ⊆ ℙ ℙ X → ℙ X
│    ∧ (∀ A : ℙ ℙ X
│      • ⋃[X] A = {x : X | ∃ S : A • x ∈ S}
│        ∧ ⋂[X] A = {x : X | ∀ S : A • x ∈ S})) : THM
```

In this generic predicate all occurrences of '$\bigcap$' are explicitly supplied with the formal parameter which was taken as implicit in the original declaration.

When actual generic parameters to global generic variables are omitted in contexts other than the defining paragraph the set supplied is the largest set of the inferred type, i.e. the set co-extensive with that type.

2.1.1.7 $\mathbb{U}$

Because of the above behaviour of the system in inferring types and actual generic parameters, a special global variable, which we have called '$\mathbb{U}$', turns out to be very useful.

'$\mathbb{U}$' may be though of as if defined by the abbreviation definition:

Z
$$\mathbb{U}[X] \mathrel{\widehat{=}} X$$

If '$\mathbb{U}$' is used in some specification or expression without supplying an actual generic parameter, the type inferrer will infer an appropriate type, and will then use for the actual parameter the set of all elements of that type.

This is used for two main purposes. It is frequently used by the proof facilities where quantifiers are introduced automatically. The main merit here is brevity and efficiency. The second purpose is for expressing theorems which may be used for unconditional rewriting, since formulae universally quantified over '$\mathbb{U}$' can readily be specialised for any type-correct rewrite, whereas quantification over other expressions gives rise to proof obligations which must be discharged before specialisation for rewriting can take place.

2.1.2 Syntactic Categories

There is only one form of quotation available for all syntactic categories in Z, encompassing predicates, expressions, and schema expressions.

Ambiguities therefore arise. Two mechanisms are available to force interpretations other than the default interpretation taken by the parser.

The first is the use of casts of the form $value\overset{\oplus}{\oplus}set$, described above. The left hand operand of a cast must be an expression or schema expression rather than a predicate, and use of a cast (even if the right hand operand is simply '$\mathbb{U}$'), will therefore force interpretation of the left hand operand as an expression or schema expression (if this is possible).

The second feature is the operator *Π*.

Π will accept as an operand only a predicate, and acts as an identity function on predicates. It may therefore be used to force interpretation of an expression as a predicate.

While:

SML
```
val schexp = ⓩ[x:ℕ]⌝;
```

is interpreted as a schema expression:

SML
```
z_type_of schexp;
```

ProofPower output
```
val it = ⓩℙ [x : ℤ]⌝ : TERM
```

SML
```
val predicate = ⓩΠ[x:ℕ]⌝;
```

is interpreted as a schema-as-predicate:

SML
```
z_type_of predicate;
```

ProofPower output
```
val it = ⓩ𝔹⌝ : TERM
```

When occurring at the outer level in a quotation, or in other contexts where a predicate is expected, logical connectives are treated as propositional connectives rather than schema calculus operators.

2.1.3 Z Terms

The definition of the ProofPower-ML datatype *Z_TERM* is used as a guide to the syntactic structure of ProofPower-Z in the following description of support for ProofPower-Z.

This datatype is *not* the type used for representing Z in HOL. Ordinary HOL terms are used for this purpose. However, HOL terms are an abstract datatype, and for some purposes it is more convenient to have a concrete datatype. A *Z_TERM* is a hybrid representation in which the top level structure has been made more visible, through the structure of the datatype, but where the consituents are still HOL terms.

The definition of *Z_TERM* begins:

Z_TERM

```
datatype Z_TERM =
```

and then continues with a definition of each of the constructors which may be used to make a *Z_TERM* (these may be found at the beginning of subsequent sections of this document).

A constructor definition consists of the name of a constructor function, all of which (for this datatype) begin with "Z". In most cases the name of the constructor is followed by a clause of the form: "of type" where "type" is an ML type. This indicates that the constructor has a parameter and gives the type of the parameter. Where there are in effect several parameters they must be supplied as a tuple.

For example, a logical conjunction of two predicates is represented as a *Z_TERM* by applying the constructor $Z\wedge$ to a pair of arguments which are HOL terms representing the two conjuncts, e.g:

SML

```
val conj = Z∧ (ⓩtrue⌝, ⓩtrue⌝);
```

ProofPower output

```
val conj = Z∧ (ⓩtrue⌝, ⓩtrue⌝) : Z_TERM
```

The structure of the datatype *Z_TERM* therefore presents a view of the abstract syntax of Z. Under this view all the syntactic categories are collapsed into one, though the formation of terms is subject to well-typing conditions. To assist the reader each constructor specification is preceded by an example of the concrete syntax of the construct, and parameter specifications are annotated with the syntactic category which would normally be expected.

Associated with these syntactic descriptions in the following are descriptions of the facilities for reasoning about the construct obtained, and more general descriptions of effective proof facilities and methods covering each area of the language.

2.1.4 Definitions for Examples

Wherever sensible the examples below are drawn simply from the language, using variables (sometimes propositional variables). Sometimes global variables introduced in the Z ToolKit are used.

For the schema calculus examples however, free variables will not suffice, and the ToolKit offers no help.

The following schema definitions are therefore provided for use in the examples:

SML
```
open_theory "usr011";
set_pc "z_library";
set_flag("z_type_check_only", false);
```

Z
```
[NAME,DATE]
```

Z
```
┌─File──────────────────────────────
│ people : ℙ NAME;
│ age : NAME ⇸ DATE
├──────────────────
│ dom age = people
└───────────────────────────────────
```

Z
```
┌─File2─────────────────────────────
│ people : ℙ NAME;
│ height : NAME ⇸ ℤ
├──────────────────
│ dom height = people
└───────────────────────────────────
```

Z
```
┌─File3─────────────────────────────
│ people : ℙ NAME
└───────────────────────────────────
```

Z
```
┌─FileOp────────────────────────────
│ File;File′; i?:ℕ
└───────────────────────────────────
```

2.2 Variables

2.2.1 Syntax

Z_TERM
```
│     (* local variable ⓩ x⌝ *)
│
│ |   ZLVar      of string          (* variable name *)
│                * TYPE             (* HOL type of variable *)
│                * TERM list        (* generic parameters *)
│
```

```
|       (* global variable ⓩ 𝕌[DATE]⌝ *)
|
| |     ZGVar       of string          (* variable name *)
|                   * TYPE             (* HOL type of variable *)
|                   * TERM list        (* generic parameters *)
```

In the literature variables in Z are either 'local variables' or 'global variables'. This distinction corresponds fairly closely to the distinction in most logical systems between 'variables' and 'constants'. In ProofPower, Z global variables are represented by HOL constants, and only local variables are represented by HOL variables.

2.2.2 Proof Support

Local variables are mainly used either in constructs in which they are bound (e.g. in quantification), in which case the details of proof support may be found in the section for the relevant binding construct. When free occurrences of local variables appear it is usually because of the skolemisation of existential quantifiers, or the elimination of universals. In either case they then behave similarly to global variables about which only the facts in the assumptions of the current subgoal are known.

Global variables are most often dealt with in proof by rewriting the expressions containing them with the defining axiom for the variable, or by the use of theorems established about them by similar means. Details of how this is achieved may be found in the chapter on Z paragraphs.

Integer and string literals are special cases of global variables whose definitions are built into the system.

2.3 Literals

2.3.1 Syntax

Z_TERM

```
|       (* positive integer literal ⓩ 34⌝ *)
|
| |     ZInt        of string
|
|       (* string literal ⓩ "characters"⌝ *)
|
| |     ZString     of string
```

Integer and string variables in ProofPower-Z are treated in a manner analogous to their treatment in HOL. They are treated as if they were global variables (HOL constants) whose characterisation is built into the system.

2.3.2 Proof Support

Proof support for numeric literals is provided primarily through the proof context for the theory *z_numbers*, which is incorporated in the context *z_library*. These include conversions for evaluating arithmetic expressions formed from literals.

SML
```
PC_C1 "z_library" rewrite_conv [] ⓩ543*20⌝;
```

ProofPower output
```
val it = ⊢ 543 * 20 = 10860 : THM
```

Proof support for strings requires a modest amount of mixed language working. The conversion *z_string_conv* converts a string literal into a sequence of character literals, however since the Z language contains at present no character literals, so these are displayed as HOL character literals.

SML
```
z_string_conv ⓩ"string"⌝;
```

ProofPower output
```
val it = ⊢ "string" = ⟨⌜`s`⌝, ⌜`t`⌝, ⌜`r`⌝, ⌜`i`⌝, ⌜`n`⌝, ⌜`g`⌝⟩ : THM
```

Combined with the facilities for reasoning about sequences, and *char_eq_conv* which decides equations over HOL character literals, this enables equations and inequalities concerning string literals to be solved.

2.4 Declarations

2.4.1 Syntax

Z_TERM
```
     (* declaration, e.g. ⓜdec_of ⓩ[x,y:ℤ]⌝⌝ *)

|    ZDec          of TERM list        (* variables *)
                   * TERM              (* expression *)


     (* schema reference, e.g. ⓜdec_of ⓩ[File!]⌝⌝ *)

|    ZSchemaDec    of TERM             (* schema expression *)
                   * string            (* decoration *)


     (* declaration list, e.g. ⓜdecl_of ⓩ[x,y:ℤ; File!]⌝⌝ *)

|    ZDecl         of TERM list        (* declarations *)
```

Declarations are constituents of most variable binding constructs, including the quantifiers which appear in the predicate calculus. It is rarely necessary to enter a bare declaration using the Z parser, and declarations are not accepted by the parser as top level constituents of a Z quotation. These remarks apply both to declaration lists, (which are using in the variable binding constructs) and their top level constituents, declarations and schemas as declarations.

If it is necessary to enter any of these constructs using the Z parser, a horizontal schema expression should be entered containing only the required declaration. ProofPower-ML functions may then be used to extract the required declaration list or declaration from the horizontal schema. If a declaration list is required, then the function *decl_of* should be used, if a single declaration is required, then the horizontal schema expression entered should contain only the one declaration, and the ML function *dec_of* should be used to extract the declaration.

When declaration lists or declarations are displayed using the pretty printer the same format is used, involving an ML quotation.

SML
```
val dec1 = dec_of ⓩ[x,y:ℤ]⌝;
```

ProofPower output
```
val dec1 = ⓩⓜdec_ofⓩ[x, y : ℤ]⌝⌝⌝ : TERM
```

The pretty printer has here introduced some extra quotation symbols, which have no effect in this context.

SML
```
val dec2 = dec_of ⓩ[File!]⌝;
```

ProofPower output
```
val dec2 = ⓩⓜdec_ofⓩ[File!]⌝⌝⌝ : TERM
```

SML
```
val decl1 = decl_of ⓩ[x,y:ℤ; File!]⌝;
```

ProofPower output
```
val decl1 = ⓩⓜdecl_ofⓩ[x, y : ℤ; File!]⌝⌝⌝ : TERM
```

2.4.2 Proof Support

For most uses of the system, proof support for transformation of declarations is built into the proof support for the construct in which the declaration appears, e.g. for the logical quantifiers. Use of the features specific to declarations is therefore unlikely to be necessary unless detailed tactical programming is being undertaken.

In that event the basic facilities concern transformation between declarations and their implicit predicates. This reflects the fact that the terms obtained by the above methods are semantically the same as the predicate implicit in the declaration list or declaration. Other aspects of the semantics of a declaration (e.g. the characteristic tuple) are present only syntactically in these terms, and are not incorporated semantically until the declaration is used in the formation of some construct in which the characteristic tuple is semantically significant (e.g. a lambda expression).

The conversion *z_dec_pred_conv* may be used to transform a declaration into its implicit predicate:

SML
```
val pred2 = z_dec_pred_conv dec1;
```

ProofPower output
```
val pred2 = ⊢ ⓜdec_ofⓩ[x, y : ℤ]⌝⌝ ⇔ {x, y} ⊆ ℤ : THM
```

SML

```
val pred3 = z_dec_pred_conv dec2;
```

ProofPower output

```
val pred3 = ⊢ ⌜dec_of⌜[File!]⌝⌝ ⇔ (File!) : THM
```

The conversion *z_decl_pred_conv* may be used to cause a declaration list to be transformed into its implicit predicate:

SML

```
val pred4 = z_decl_pred_conv decl1;
```

ProofPower output

```
val pred4 = ⊢ ⌜decl_of⌜[x, y : ℤ; File!]⌝⌝ ⇔ {x, y} ⊆ ℤ ∧ (File!) : THM
```

Conversions taking predicates into declarations are also available:

SML

```
val dec3 = z_pred_dec_conv ⌜{x, y} ⊆ ℤ⌝;
```

ProofPower output

```
val dec3 = ⊢ {x, y} ⊆ ℤ ⇔ ⌜dec_of⌜[x, y : ℤ]⌝⌝ : THM
```

SML

```
val dec3 = z_pred_dec_conv ⌜Π(File !)⌝;
```

ProofPower output

```
val dec3 = ⊢ (File!) ⇔ ⌜dec_of⌜[(File!)]⌝⌝ : THM
```

Note that here the use of *Π* was necessary in the term quotation to force *File*! to be interpreted as a predicate rather than as a schema expression.

THE Z PREDICATE CALCULUS

3.1 Propositional Connectives

Z_TERM

```
    (* ⓩ true⌝ *)

    ZTrue

    (* ⓩ false⌝ *)

|   ZFalse

    (* negation, e.g. ⓩ ¬ p⌝ *)

|   Z¬            of TERM           (* predicate *)

    (* conjunction, e.g. ⓩ p ∧ q⌝ *)

|   Z∧            of TERM * TERM    (* predicates *)

    (* disjunction, e.g. ⓩ p ∨ q⌝ *)

|   Z∨            of TERM * TERM    (* predicates *)

    (* implication, e.g. ⓩ p ⇒ q⌝ *)

|   Z⇒            of TERM * TERM    (* predicates *)

    (* bi−implication, e.g. ⓩ p ⇔ q⌝ *)

|   Z⇔            of TERM * TERM    (* predicates *)
```

3.1.1 Propositional Reasoning in Z

The Z propositional connectives are mapped directly to the corresponding connective in HOL, and propositional reasoning in ProofPower-Z behaves therefore in a manner identical to propositional reasoning in ProofPower-HOL.

The behaviour is sensitive to the current proof context, but almost all the proof contexts behave in the same way for propositional reasoning.

A suitable context for propositional reasoning in Z is “*z_language*”, and all other Z proof contexts contain the same material for propositional reasoning.

The main methods of proof are:

1. Forward proof using elementary rules.
2. Goal oriented proof by stripping.
3. Goal oriented automatic proof.

These methods are described and illustrated in each of the following subsections.

3.1.1.1 Forward proof using elementary rules

Forward propositional reasoning is rarely required in ProofPower-Z proofs. The methods of forward proof are illustrated below showing the use of the following rules.

These are all rules which behave identically for ProofPower-HOL and ProofPower-Z. In the case of Z however, it should be noted that they concern only the propositional connectives, and do not operate on the corresponding schema calculus operators.

asm_rule - make or use an assumption

Given an arbitrary boolean term ‘t’, infer ‘$t \vdash t$’, e.g.:

SML

```
│val x_eq_y = asm_rule ⓩx=y⌝;
```

ProofPower output

```
│val x_eq_y = x = y ⊢ x = y : THM
```

SML

```
│val eq_sym_thm = asm_rule ⓩx=y ⇒ y=x⌝;
```

ProofPower output

```
│val eq_sym_thm = x = y ⇒ y = x ⊢ x = y ⇒ y = x : THM
```

⇒_elim - use an implication (modus ponens)

Given a theorem of the form ‘$\Psi \vdash a \Rightarrow b$’ and one of the form ‘$\Upsilon \vdash a$’ infer ‘$\Psi,\ \Upsilon \vdash b$’, e.g:

SML

```
│val y_eq_x = ⇒_elim eq_sym_thm x_eq_y;
```

ProofPower output

```
│val y_eq_x = x = y ⇒ y = x, x = y ⊢ y = x : THM
```

⇒_intro - prove an implication

Given an arbitrary theorem ‘$\Psi \vdash a$’ possibly having $⌜_Z p1 ⌝$ as an assumption, infer the theorem ‘ $\Psi \backslash \{⌜_Z p1 ⌝\} \vdash p1 \Rightarrow a$′.

SML
```
val imp_thm = ⇒_intro ⌜Z x = y ⇒ y = x⌝ y_eq_x;
```

ProofPower output
```
val imp_thm = x = y ⊢ (x = y ⇒ y = x) ⇒ y = x : THM
```

SML
```
val imp_thm2 = ⇒_intro ⌜Z x = y⌝ imp_thm;
```

ProofPower output
```
val imp_thm2 = ⊢ x = y ⇒ (x = y ⇒ y = x) ⇒ y = x : THM
```

strip_∧_rule - split up a conjunction

Given an arbitrary theorem whose conclusion is a conjunction, e.g. ‘$\Psi \vdash a \wedge b \wedge c$’ deliver a list of theorems, one for each conjunct separately: ‘$[\Psi \vdash a, \Psi \vdash b, \Psi \vdash c]$’.

SML
```
val [thm1,thm2,thm3] = strip_∧_rule (asm_rule ⌜Z a ∧ b ∧ c⌝);
```

ProofPower output
```
val thm1 = a ∧ b ∧ c ⊢ a : THM
val thm2 = a ∧ b ∧ c ⊢ b : THM
val thm3 = a ∧ b ∧ c ⊢ c : THM
```

list_∧_intro - create a conjunction from a list of theorems

Takes ‘$[\Psi \vdash a, \Omega \vdash b, \Upsilon \vdash c]$’ to ‘$\Psi, \Omega, \Upsilon \vdash a \wedge b \wedge c$’.

SML
```
val newconj = list_∧_intro [thm1,thm2,thm3];
```

ProofPower output
```
val newconj = a ∧ b ∧ c ⊢ a ∧ b ∧ c : THM
```

3.1.1.2 Proof by stripping

In suitable proof contexts (which is almost all of them) proofs of propositional results can be completed using only “stripping” facilities.

In such cases a proof of the form:

SML
```
set_goal([],⌜Z conjecture ⌝);
a contr_tac;
```

will suffice.

e.g., first set the goal:

SML
```
set_goal([],ⓩ a ∧ b ⇒ b ∧ a⌝);
```

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

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

(* ?⊢ *)  ⓩa ∧ b ⇒ b ∧ a⌝
...
```

Then initiate proof by contradiction by applying *contr_tac*.

SML
```
a contr_tac;
```

ProofPower output
```
Tactic produced 0 subgoals:
Current and main goal achieved
val it = () : unit
```

In this proof method the conjecture is negated and “stripped” into the assumptions, which process is sometimes sufficient to discharge the result without further intervention by the user. In the case of pure propositional results this is always sufficient.

It is however instructive to undertake such proofs in a more piecemeal way, so as to get an understanding of how these stripping facilities work.

To achieve this a proof of the form:

SML
```
set_goal([],ⓩ conjecture ⌝);
a z_strip_tac;  (* repeat as often as necessary *)
```

may be used.

e.g.:

SML
```
set_goal([],ⓩ a ∧ b ⇒ b ∧ a⌝);
```

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

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

(* ?⊢ *)  ⓩa ∧ b ⇒ b ∧ a⌝
...
```

SML
```
a z_strip_tac;
```

The current goal is an implication, which is dealt with by transferring the antecedent of the implication into the assumption list. As it is added to the assumptions the antecedent, ⌜Z a∧b⌝, is completely stripped, which in this case results in it being split into two separate assumptions ⌜Z a⌝ and ⌜Z b⌝.

ProofPower output
```
Tactic produced 1 subgoal:

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

(*  2 *)  ⌜Z a⌝
(*  1 *)  ⌜Z b⌝

(* ?⊢ *)  ⌜Z b ∧ a⌝
...
```

Now the conclusion of the current goal is a conjunction, and stripping results in two subgoals, one for each conjunct.

SML
```
a z_strip_tac;
```

ProofPower output
```
Tactic produced 2 subgoals:

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

(*  2 *)  ⌜Z a⌝
(*  1 *)  ⌜Z b⌝

(* ?⊢ *)  ⌜Z a⌝


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

(*  2 *)  ⌜Z a⌝
(*  1 *)  ⌜Z b⌝

(* ?⊢ *)  ⌜Z b⌝
...
```

Each of these subgoals has its conclusion among its assumptions. If no other action is possible *z_strip_tac* will check for this condition and will discharge the subgoal if it applies.

SML
```
a z_strip_tac;
```

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

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

(*  2 *)  ⓩa⌝
(*  1 *)  ⓩb⌝

(* ?⊢ *)  ⓩa⌝
...
```

SML
```
a z_strip_tac;
```

ProofPower output
```
Tactic produced 0 subgoals:
Current and main goal achieved
val it = () : unit
```

In such a proof, the conclusion of the current goal is stripped one step at a time. Whenever an assumption is added to the list of assumptions in the current goal, this assumption is completely stripped. If the original goal was completely well formed Z, (and the proof context is appropriate for reasoning in Z) then this stripping process should result only in subgoals which are also well formed Z. If the original subgoal was true, so will be any subgoals obtained by this stripping process, so if an evidently false subgoal appears the original conjecture must have been false.

To get an idea of how the assumptions are being stripped a tactic is available which performs step-by-step stripping on new assumptions before they are taken out of the conclusion.

SML
```
set_goal([],ⓩ conjecture ⌝);
a step_strip_tac; (* repeat as often as necessary *)
```

e.g.:

SML
```
set_goal([], ⓩ (a ∧ b ∧ (¬a ∨ ¬b)) ⇒ false⌝);
```

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

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

(* ?⊢ *)  ⓩa ∧ b ∧ (¬ a ∨ ¬ b) ⇒ false⌝
...
```

If *z_strip_tac* were used at this point the antecendent of the implication in the conclusion of the goal would be completely stripped into the assumptions, which would in fact solve the goal.

step_strip_tac, however, performs as much stripping as possible while the antecedent is still in place in the conclusion, creating new assumptions only when no further stripping is possible.

In this case the leftmost conjunct of the antecendent is considered completely stripped and is added to the assumptions, while the remaining conjuncts are left for further attention.

SML

```
a step_strip_tac;
```

ProofPower output

```
Tactic produced 1 subgoal:

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

(*  1 *)  ⌜a⌝

(* ?⊢ *)  ⌜b ∧ (¬ a ∨ ¬ b) ⇒ false⌝
...
```

SML

```
a step_strip_tac;
```

The leftmost conjunct of the antecedent is again transferred to the assumptions.

ProofPower output

```
Tactic produced 1 subgoal:

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

(*  2 *)  ⌜a⌝
(*  1 *)  ⌜b⌝

(* ?⊢ *)  ⌜¬ a ∨ ¬ b ⇒ false⌝
...
```

The antecedent is now a disjunction. Stripping a disjunction into the assumptions gives rise to a case split.

SML

```
a step_strip_tac;
```

ProofPower output

```
Tactic produced 2 subgoals:

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

(*  2 *)  ⓩa⌝
(*  1 *)  ⓩb⌝

(* ?⊢ *)  ⓩ¬ b ⇒ false⌝


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

(*  2 *)  ⓩa⌝
(*  1 *)  ⓩb⌝

(* ?⊢ *)  ⓩ¬ a ⇒ false⌝
...
```

The next step would attempt to strip ⓩ¬ a⌝ into the assumptions. However, ⓩa⌝ is already in the assumptions and this results in the discharge of the subgoal.

SML

```
a step_strip_tac;
```

ProofPower output

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

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

(*  2 *)  ⓩa⌝
(*  1 *)  ⓩb⌝

(* ?⊢ *)  ⓩ¬ b ⇒ false⌝
...
```

SML

```
a step_strip_tac;
```

ProofPower output

```
Tactic produced 0 subgoals:
Current and main goal achieved
val it = () : unit
```

3.1.1.3 Automatic Proof

An automatic proof procedure is provided for each proof context which is usually capable of solving results reducible to the pure propositional calculus. Even when it fails it may have resulted in more simplification than would be obtained by other methods.

SML
```
set_goal([],⌜Z conjecture⌝);
a (prove_tac[]); (* once only *)
```

e.g.:

SML
```
set_goal([],⌜Z (a ∧ b ∧ (¬a ∨ ¬b)) ⇒ false⌝);
```

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

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

(* ?⊢ *)  ⌜Z a ∧ b ∧ (¬ a ∨ ¬ b) ⇒ false⌝
...
```

SML
```
a (prove_tac[]);
```

ProofPower output
```
Tactic produced 0 subgoals:
Current and main goal achieved
val it = () : unit
```

If subgoals are left by *prove_tac* then they will not normally be further progressed by repeated application of the tactic.

prove_tac should be used only when the current goal has no assumptions, or where it is expected that *prove_tac* can completely discharge the current goal without making use of the assumptions.

If the assumptions must be used to obtain the proof, or if there are assumptions and *prove_tac* is likely to leave some outstanding subgoals, then instead of *prove_tac*, *asm_prove_tac* should be used:

SML
```
a (asm_prove_tac[]); (* once only *)
```

e.g.:

SML
```
set_goal([⌜Z ¬a⌝,⌜Z ¬b⌝], ⌜Z ¬(a ∨ b)⌝);
```

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

(∗ ∗∗∗ Goal "" ∗∗∗ ∗)

(∗  2 ∗)  ⓩ¬ b⌝
(∗  1 ∗)  ⓩ¬ a⌝

(∗ ?⊢ ∗)  ⓩ¬ (a ∨ b)⌝
...
```

SML
```
a (asm_prove_tac[]);
```

ProofPower output
```
Tactic produced 0 subgoals:
Current and main goal achieved
val it = () : unit
```

If *prove_tac* is used in these circumstances then it may fail.

If the conjecture to be proven can be completely proved by one application of *prove_tac* then invocation of the subgoal package is unnecessary. *prove_rule* may be used to obtain the result as follows:

SML
```
prove_rule [] ⓩ (a ∧ b ∧ (¬a ∨ ¬b)) ⇒ false⌝;
```

ProofPower output
```
val it = ⊢ a ∧ b ∧ (¬ a ∨ ¬ b) ⇒ false : THM
```

3.2 Predicates

Z_TERM
```
        (∗ equation, e.g. ⓩ a = b⌝ ∗)

  |     ZEq              of TERM ∗ TERM   (∗ expressions ∗)

        (∗ membership, e.g. ⓩ a ∈ b⌝ ∗)

  |     Z∈               of TERM ∗ TERM   (∗ expressions ∗)

        (∗ schema predicate, e.g. ⓩ Π (File ′)⌝ ∗)

  |     ZSchemaPred          of TERM      (∗ schema expression ∗)
                             ∗ string     (∗ decoration ∗)
```

At bottom there are just two kinds of predicate in Z, equations and membership assertions, though there are a variety of ways in which these are presented in the concrete syntax.

Equations always appear in a direct literal way, but membership statements come in a variety of concrete forms.

Where a rel fixity paragraph has been entered the actual membership sign is omitted, and the set may be "applied" to its member as if it were a propositional function or predicate using prefix, postfix or "fancyfix" notation, according to the details in the fixity declaration.

Finally schemas as predicates are effectively abbreviations of assertions about the membership of theta terms in the schema expressions, the theta terms themselves being abbreviations of binding displays.

3.2.1 Equations

3.2.1.1 Syntax

```
Z_TERM
│        (* equation, e.g. ⓩ a = b⌝ *)
│
│ |      ZEq          of TERM * TERM   (* expressions *)
```

3.2.1.2 Proof Support

Equations are exploited in proof usually by the use of the rewriting facilities, which enable equations to be used to transform terms.

Equations are established in two main ways:

- By rewriting either or both sides of the equation until they are identical.
- By using identity criteria specific to the type of the expressions equated.

In the former case the main resource deployed will be the standard rewriting facilities.

The latter case can be further divided into cases according to the type of the expression. Types of expressions in Z may be classified according to their outer type constructor into the following categories:

1. Elements of Given sets
2. Sets
3. Tuples
4. Bindings
5. Elements of Free Types

Elements of given sets, unless further constraints are added, admit no opportunities for proving equations other than by rewriting the two sides of the equation, since nothing is known about the elements of a given set.

Sets may be proven equal using the principle of extensionality, that two sets are equal if and only if they have the same members. This principle is built in to the proof context *z_language_ext*:

SML
```
|PC_C1 "z_language_ext" rewrite_conv[] ⓩ x = y ⦂ ℙ 𝕌⌝;
```

ProofPower output
```
|val it = ⊢ x = y ⇔ (∀ x1 : 𝕌 • x1 ∈ x ⇔ x1 ∈ y) : THM
```

Note the need for the type cast here to ensure that the free variables have power set types; otherwise the principle of extensionality could not be applied. Normally the cast would not be necessary because the context would be sufficient to determine the type of the free variables.

Tuples are proven equal with an analogous principle, viz. that two tuples are equal if and only if each of their components is equal. The well typing rules will ensure that they have the same number and type of components. This principle, for use with tuple displays only, is built into the proof context *z_language*.

SML
```
|rewrite_conv[] ⓩ (x, y) = (v, w)⌝;
```

ProofPower output
```
|val it = ⊢ (x, y) = (v, w) ⇔ x = v ∧ y = w : THM
```

Bindings are similar to tuples, being labelled rather than unlabelled records. Two bindings are equal iff each of their respective components are equal. The type system ensures that the names and types of the components are the same. This principle is supported by *z_binding_eq_conv1*, which is built into the proof context "z_language" for binding displays.

SML
```
|rewrite_conv[] ⓩ(cn1 ≙ x, cn2 ≙ y) = (cn1 ≙ v, cn2 ≙ w)⌝;
```

ProofPower output
```
|val it = ⊢ (cn1 ≙ x, cn2 ≙ y) = (cn1 ≙ v, cn2 ≙ w)
|                          ⇔ x = v ∧ y = w : THM
```

SML
```
|z_binding_eq_conv1 ⓩx = y ⦂ [cn1: 𝕌; cn2: 𝕌]⌝;
```

ProofPower output
```
|val it = ⊢ x = y ⇔ x.cn1 = y.cn1 ∧ x.cn2 = y.cn2 : THM
```

Two elements of the same free type are equal under the following conditions:

1. They are formed using the same constructor (the ranges of the constructors are disjoint)
2. Where the constructor is a function the values supplied to this function are the same (the constructor functions are injections).

An equality principle may be derived from the axiom which introduces the free type. Automatic derivation of this principle is not yet supported.

3.2.2 Membership Assertions

3.2.2.1 Syntax

Z_TERM

```
|        (* membership, e.g. ⓩ a ∈ b⌝ *)
|
| |      Z∈                of TERM * TERM  (* expressions *)
```

The forms of concrete syntax are varied by the use of *rel fixity* paragraphs (see section 5.2.1).

The following examples illustrate these variations.

Without a *rel fixity* paragraph in force *prerel* in the example below is treated as a local variable denoting a function, and the quoted term as a function application.

SML

```
|dest_z_term ⓩprerel x⌝;
```

ProofPower output

```
|val it = ZApp (ⓩprerel⌝, ⓩx⌝) : Z_TERM
```

The following paragraph attaches prefix status to the name *prerel* (even though this has not been declared as a global variable).

Z

```
|rel prerel _
```

Which causes the same Z quotation to be interpreted as a set membership assertion:

SML

```
|dest_z_term ⓩprerel x⌝;
```

ProofPower output

```
|val it = Z∈ (ⓩx⌝, ⓩ(prerel _)⌝) : Z_TERM
```

Note that to parse the term consisting only of the local variable *prerel* once the fixity clause has been introduced, the name must be supplied with an underscore and enclosed in brackets.

Z

```
|rel _ postrel
```

SML

```
|dest_z_term ⓩx postrel⌝;
```

ProofPower output

```
|val it = Z∈ (ⓩx⌝, ⓩ(_ postrel)⌝) : Z_TERM
```

Z

```
|rel _ infixrel _
```

SML
```
dest_z_term ⓩx infixrel y⌝;
```

ProofPower output
```
val it = Z∈ (ⓩ(x, y)⌝, ⓩ(_ infixrel _)⌝) : Z_TERM
```

The above example shows that where there is more than one argument these are made into a tuple.

Z
```
rel rellb ... relrb
```

SML
```
dest_z_term ⓩrellb 1,2,3 relrb⌝;
```

ProofPower output
```
val it = Z∈ (ⓩ⟨1, 2, 3⟩⌝, ⓩ(rellb ... relrb)⌝) : Z_TERM
```

The use of "..." in a rel fixity clause indicates a position at which a sequence display is required with the sequence brackets omitted. This is interpreted as asserting the membership of the sequence in the set.

3.2.2.2 Proof Support

Reasoning about membership is in general specific to the construct of which membership is asserted. Throughout this tutorial constructs in the Z language which yield sets are normally characterised by identifying the conditions for membership of the resulting set. Each of these characterisations provides a method for proving a result about set membership from more elementary results (possibly also about membership, but in this case usually of simpler expressions).

A general convention is adopted for the naming of conversions concerning set membership, and they may therefore be sought in the keyword index for the Reference Manual. Such conversions have names of the form *z_∈_something_conv*, where *something* is the name of the kind of construct of which membership is asserted. In the case of constructs which form part of the Z language most of these conversion are built into the proof context "*z_language*" and so their names need not be invoked or remembered for most purposes. Where global variables are defined in the Z ToolKit which are functions yielding sets, then the relevant theory will normally contain a theorem giving a characterisation by membership of the resulting sets, and a proof context will normally be supplied which invokes this characterisation.

3.2.3 Schemas as Predicates

3.2.3.1 Syntax

An arbitrary expression denoting a set of bindings, together with an optional decoration, may be used as a predicate.

Z_TERM
```
        (* schema predicate, e.g. ⓩ Π (File ′)⌝ *)

    |   ZSchemaPred     of TERM         (* schema expression *)
                        * string        (* decoration *)
```

3.2.3.2 Proof Support

Schemas as predicates are eliminated in favour of membership statements by *z_schema_pred_conv* (which is built into rewriting in proof context *z_language*) or by rewriting with *z′schema_pred_def*:

SML
```
pure_rewrite_conv[z′schema_pred_def]ⓩ Π[x:X]⌝;
```

ProofPower output
```
val it = ⊢ [x : X] ⇔ (x ≙ x) ∈ [x : X] : THM
```

Normally the membership statement will be eliminated immediately, as follows:

SML
```
rewrite_conv[]ⓩ Π[x:X]⌝;
```

ProofPower output
```
val it = ⊢ [x : X] ⇔ x ∈ X : THM
```

3.2.4 Propositional Equational Reasoning

Special facilities are provided for solving problems which lie in the domain of the propositional calculus augmented by equality. These facilities are not specific to Z but work in Z because in this region the mapping from Z to HOL is completely transparent.

The facilities provided consist of decision procedures for problems in this domain provided through *prove_tac* and *prove_rule* in proof contexts *prop_eq* and *prop_eq_pair*.

e.g.:

SML
```
push_pc "prop_eq";
prove_rule [] ⓩa=b ∧ c=d ∧ e=f ⇒ b=e ⇒ c=a ⇒ d=f⌝;
pop_pc();
```

ProofPower output
```
...
val it = ⊢ a = b ∧ c = d ∧ e = f ⇒ b = e ⇒ c = a ⇒ d = f : ...
THM
```

PC_T1 may be used to invoke this decision procedure during a tactical proof in the following way:

SML
```
set_goal([], ⓩa=b ⇒ b=c ⇒ c=a⌝);
a (PC_T1 "prop_eq" prove_tac[]);
```

ProofPower output
```
...
Tactic produced 0 subgoals:
Current and main goal achieved
...
```

This method avoids the need to change the current proof context. (*PC_T* and *PC_T1* are general facilities for invoking a tactic in a specific proof context)

prop_eq_pair is a similar proof context which extends the domain or reasoning of *prop_eq* to include a knowledge of HOL ordered pairs, however, since HOL pairs are distinct from Z pairs (2-tuples) this does not work for Z.

3.3 Quantifiers

Z_TERM

```
      (* universal quantification, ⌜Z ∀ File | p • q⌝ *)

  |   Z∀          of TERM            (* declaration *)
                  * TERM * TERM      (* predicates *)

      (* existential quantification, ⌜Z ∃ File | p • q⌝ *)

  |   Z∃          of TERM            (* declaration *)
                  * TERM * TERM      (* predicates *)

      (* unique existential quantification, ⌜Z ∃1 File | p • q⌝ *)

  |   Z∃1         of TERM            (* declaration *)
                  * TERM * TERM      (* predicates *)
```

3.3.1 Universal Quantification

3.3.1.1 Syntax

Z_TERM

```
      (* universal quantification, ⌜Z ∀ File | p • q⌝ *)

  |   Z∀          of TERM            (* declaration *)
                  * TERM * TERM      (* predicates *)
```

3.3.1.2 Proof Support

For forward proof the rules *z_∀_elim* and *z_∀_intro* are the key facilities.

In its most simple case, where the signature of the declaration part of the universal quantifier contains only a single component name, a value for that variable is supplied:

SML

```
z_plus_order_thm;
```

ProofPower output

```
val it = ⊢ ∀ i : 𝕌
    • ∀ j, k : 𝕌
      • j + i = i + j ∧ (i + j) + k = i + j + k
        ∧ j + i + k = i + j + k : THM
```

SML
```
z_∀_elim ⓩ43⌝ z_plus_order_thm;
```

ProofPower output
```
val it = ⊢ 43 ∈ 𝕌 ∧ true
    ⇒ (∀ j, k : 𝕌
      • j + 43 = 43 + j
        ∧ (43 + j) + k = 43 + j + k
        ∧ j + 43 + k = 43 + j + k) : THM
```

Where the signature has more than one component it is necessary to supply a binding which has the same signature as the outermost quantifier. Any expression of the right type will do:

SML
```
z_plus_assoc_thm;
```

ProofPower output
```
val it = ⊢ ∀ i, j, k : 𝕌 • (i + j) + k = i + j + k : THM
```

SML
```
z_∀_elim ⓩexp ⦂ [i, j, k : ℤ]⌝ z_plus_assoc_thm;
```

ProofPower output
```
val it = ⊢ {exp.i, exp.j, exp.k} ⊆ 𝕌 ∧ true
    ⇒ (exp.i + exp.j) + exp.k = exp.i + exp.j + exp.k : THM
```

In the above case projections were used when substituting the value into the new conclusion. If an explicit binding is supplied (which is the most common case) these projections are not required:

SML
```
z_∀_elim ⓩ(i ≜ 2, j ≜ 3, k ≜ 4)⌝ z_plus_assoc_thm;
```

ProofPower output
```
val it = ⊢ {2, 3, 4} ⊆ 𝕌 ∧ true ⇒ (2 + 3) + 4 = 2 + 3 + 4 : THM
```

z_∀_intro1 is a left inverse of *z_∀_elim*, and may be used to introduce universal quantifiers.

SML
```
z_∀_elim ⓩ(i ≜ v⦂ℤ, j ≜ w⦂ℤ, k ≜ x⦂ℤ)⌝ z_plus_assoc_thm;
```

ProofPower output
```
val it = ⊢ {v, w, x} ⊆ 𝕌 ∧ true ⇒ (v + w) + x = v + w + x : THM
```

SML
```
z_∀_intro1 it;
```

ProofPower output
```
val it = ⊢ ∀ v, w, x : 𝕌 • (v + w) + x = v + w + x : THM
```

Because of the complications caused by the predicate implicit in the declarations, forward proof using these rules is much less convenient than using more powerful facilities.

For example, if the result required is $(2 + 3) + 4 = 2 + 3 + 4$, then this can most conveniently be proven using *prove_rule* in an appropriate proof context.

e.g.

SML
```
|prove_rule [z_plus_assoc_thm] ⓩ(2 + 3) + 4 = 2 + 3 + 4⌝;
```

ProofPower output
```
|val it = ⊢ (2 + 3) + 4 = 2 + 3 + 4 : THM
```

In goal oriented proof a wider range of facilities provide support for universal quantification.

Elimination of an outer universal quantifier in the conclusion of the current goal is accomplished by *z_∀_tac*.

SML
```
|set_goal([],ⓩ∀ x:X • x = x⌝);
```

ProofPower output
```
|(∗ ∗∗∗ Goal "" ∗∗∗ ∗)
|
|(∗ ?⊢ ∗) ⓩ∀ x : X • x = x⌝
|...
```

SML
```
|a z_∀_tac;
```

ProofPower output
```
|(∗ ∗∗∗ Goal "" ∗∗∗ ∗)
|
|(∗ ?⊢ ∗) ⓩx ∈ X ∧ true ⇒ x = x⌝
|...
```

Exactly the same effect for universal quantifiers is obtained by using *z_strip_tac*.

Universal assumptions may be specialised in a number of ways.

The tactics *z_spec_asm_tac* or *z_spec_nth_asm_tac* may be used in a manner analogous to *z_∀_elim* to specialise an assumption which is unversally quantified. The result of the specialisation is stripped into the assumptions, and the original assumption also remains in the assumptions.

SML
```
|set_goal([], ⓩ[X] (∀x:X • P x ⇒ Q x) ⇒ (∀x:X • P x) ⇒ (∀x:X • Q x)⌝);
|a (REPEAT z_strip_tac);
```

ProofPower output
```
Tactic produced 1 subgoal:

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

(*  3 *)  ⓩ∀ x : X • P x ⇒ Q x⌝
(*  2 *)  ⓩ∀ x : X • P x⌝
(*  1 *)  ⓩx ∈ X⌝

(* ?⊢ *)  ⓩQ x⌝
...
```

SML
```
a (z_spec_nth_asm_tac 3 ⓩx⌝);
```

ProofPower output
```
Tactic produced 1 subgoal:

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

(*  4 *)  ⓩ∀ x : X • P x ⇒ Q x⌝
(*  3 *)  ⓩ∀ x : X • P x⌝
(*  2 *)  ⓩx ∈ X⌝
(*  1 *)  ⓩ¬ P x⌝

(* ?⊢ *)  ⓩQ x⌝
...
```

The effect of stripping ⓩP x ⇒ Q x⌝ into the assumptions here has been to cause a case split into the two cases ⓩQ x⌝ and ⓩ¬ P x⌝ of which the first case was discharged automatically because the new assumption matches the conclusion of the goal.

One more specialisation completes the proof:

SML
```
a (z_spec_nth_asm_tac 3 ⓩx⌝);
```

ProofPower output
```
Tactic produced 0 subgoals:
Current and main goal achieved
val it = () : unit
```

Specialisation of assumption 3 has created a new assumption which contradicts an existing assumption, and therefore discharges the goal.

3.3.2 Existential Quantification

3.3.2.1 Syntax

Z_TERM

```
        (∗ existential quantification, ⓩ ∃ File | p • q⌝ ∗)

|       Z∃          of TERM             (∗ declaration ∗)
                    ∗ TERM ∗ TERM       (∗ predicates ∗)
```

3.3.2.2 Proof Support

Support for existential quantifiers in goal oriented proof consists in three main features.

Firstly, existentials entered into the assumptions are skolemised automatically in all Z proof contexts.

SML

```
set_goal([],ⓩ(∃ x:ℤ • x=2 ∧ x=3) ⇒ false⌝);
a z_strip_tac;
```

ProofPower output

```
...
(∗ *** Goal "" *** ∗)

(∗  3 ∗)  ⓩx ∈ ℤ⌝
(∗  2 ∗)  ⓩx = 2⌝
(∗  1 ∗)  ⓩx = 3⌝

(∗ ?⊢ ∗)  ⓩfalse⌝
...
```

Secondly when stripping a negated existential conclusion the negation is pushed over the quantifier, resulting in a universal quantifier, which will be eliminated in the next stage of stripping. The universal quantifier will be elimated on the next step of stripping so the effect is similar to skolemisation.

SML

```
set_goal([],ⓩ¬(∃ x:ℤ • x=2 ∧ x=3)⌝);
a z_strip_tac;
```

ProofPower output

```
...
(∗ *** Goal "" *** ∗)

(∗ ?⊢ ∗)  ⓩ∀ x : ℤ • ¬ (x = 2 ∧ x = 3)⌝
...
```

SML

```
a z_strip_tac;
```

ProofPower output

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

(* ?⊢ *) ⓩx ∈ ℤ ∧ true ⇒ ¬ (x = 2 ∧ x = 3)⌝
...
```

Finally, the user may attempt to prove a goal with an existential conclusion by offering a witness. This is done using *z_∃_tac*.

SML

```
set_merge_pcs ["z_library"];
set_goal([],ⓩ∃ x:ℤ • x*x = 4⌝);
a (z_∃_tac ⓩ2⌝);
```

ProofPower output

```
...
(* *** Goal "" *** *)

(* ?⊢ *) ⓩ2 ∈ ℤ ∧ true ∧ 2 * 2 = 4⌝
...
```

z_∃_tac is parameterised in a manner similar to *z_∀_elim*, accepting a binding (display or expression) in general, and other types of value where the binding would have only one component.

The proof can be completed in this proof context by rewriting.

SML

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

ProofPower output

```
Tactic produced 0 subgoals:
Current and main goal achieved
val it = () : unit
```

WARNING: *z_∃_tac* could send you down a blind alley. A true existential goal can be transformed into a false subgoal if the wrong witness is identified.

In some cases it may be more convenient effectively to offer several alternative witnesses which work in difference circumstances. This is analogous to instantiating the same universal assumption in more than one way. If this is desired then instead of using *z_∃_tac* the user should switch to proof by contradiction by using *contr_tac*. This will transfer the negated existential conclusion into the assumptions, which will appear as a universally quantified assumption after the negation has been pushed in. The assumption may then be specialised as often as necessary.

3.3.3 Unique Existential Quantification

3.3.3.1 Syntax

Z_TERM

```
        (* unique existential quantification, ⓩ ∃₁ File | p • q⌝ *)

 |      Z∃₁          of TERM             (* declaration *)
                     * TERM * TERM       (* predicates *)
```

3.3.3.2 Proof Support

Support for unique existential quantifiers in goal oriented proof consists in two main features.

Firstly, unique existentials entered into the assumptions are skolemised automatically in all Z proof contexts. A unique existential results in a universal assumption which expresses the uniqueness condition.

SML
```
set_goal([],ⓩ(∃₁ x:ℤ • x=2 ∧ x=3) ⇒ false⌝);
a z_strip_tac;
```

ProofPower output
```
...
(* *** Goal "" *** *)

(*  4 *)  ⓩx ∈ ℤ⌝
(*  3 *)  ⓩx = 2⌝
(*  2 *)  ⓩx = 3⌝
(*  1 *)  ⓩ∀ x′ : ℤ | true ∧ x′ = 2 ∧ x′ = 3 • x′ = x⌝

(* ?⊢ *)  ⓩfalse⌝
...

```

Secondly when stripping a negated unique existential conclusion the negation is pushed over the existential quantifier, resulting in a universal quantifier, which will be eliminated in the next stage of stripping. The universal quantifier will be elimated on the next step of stripping so the effect is similar to skolemisation.

SML
```
set_goal([],ⓩ¬(∃₁ x:ℤ • x=2 ∨ x=3)⌝);
a z_strip_tac;
```

ProofPower output
```
...
(* *** Goal "" *** *)

(* ?⊢ *)  ⓩ∀ x : ℤ
              | true ∧ (x = 2 ∨ x = 3)
              • ¬
              (∀ x′ : ℤ | true ∧ (x′ = 2 ∨ x′ = 3) • x′ = x)⌝
...
```

A unique existential conclusion may be handled by proof by contradiction using *contr_tac*.

3.4 Predicate Calculus Proofs

Several methods of proof may be adopted for results in the predicate calculus (or for dealing with the predicate calculus aspects of other proofs):

They are:

1. Proof by stripping.
2. Automatic proof.
3. Proof by the "two tactic method".
4. Proof using forward chaining.

The first two methods, which are complete for propositional logic may fail to solve some results in the pure predicate calculus. For these a simple and systematic and simple approach known as the 'two-tactic' approach will suffice, or alternatively forward chaining, may suffice to obtain the result with less effort on the part of the user.

These two new methods are described in each of the following subsections, and then a selection of examples are provided which may be attempted by any of the above methods.

3.4.1 The Two Tactic Method

Proof by stripping is effective in discharging a goal only where the reasoning is mainly propositional. Where the proof will depend either on appropriate specialisation of universally quantified assumptions, or on the choice of a suitable witness for proving an existential conclusion stripping will not suffice.

The two tactic method injects into the proof process based on stripping, user directed specialisation of universal assumptions. In the context of a proof by contradiction (in which existential conclusions will not arise) this is sufficient to discharge any goals which are reduced to reasoning in the pure first order predicate calculus.

The method is therefore as follows:

```
SML
|set_goal([],conjecture);
|a contr_tac;                          (* once suffices *)
|a (z_spec_asm_tac ⓩ assumption ⌝ ⓩ value⌝);     (* as many times as necessary *)
```

The choice of universal assumptions and of the values to specialise them to depends on the user identifying one or more specialisations which will result in the derivation of a contradiction from the assumptions.

3.4.2 Forward Chaining

Forward chaining facilities often provide an easier way of achieving proofs requiring instantiation of universal assumptions.

When a proof fails to be solved by *contr_tac* alone, *all_asm_fc_tac* may be capable progressing the proof.

all_asm_fc_tac will attempt to instantiate universally quantified assumptions which are effectively implications to values which will enable forward inference to take place. This is achieved by matching the antecedent of the implication against other assumptions.

If *contr_tac* leaves goals outstanding, try progressing the proof using:

SML
```
a (all_asm_fc_tac[]); (* once or twice *)
```

This may lead to the derivation of a contradiction with less effort from the user, however it will sometimes fail to solve a goal (and often generate a lot of irrelevant new asumptions). If forward chaining is not heading anywhere useful, revert to the two tactic method.

A related tactic suitable for use with Z is *all_fc_tac*, which chains forward using implications derived from a list of theorems supplied as an argument, matching these against the assumptions, using the assumptions to match the antecedents of the implications.

fc_tac and *asm_fc_tac* are also useful (see ProofPower *Reference Manual* [9]), but these are liable to introduce HOL universals, leaving a mixed language subgoal.

3.4.3 Predicate Calculus with Equality

The above facilities primarily support reasoning in the pure predicate calculus, and a proof using these facilities may fail by failing to exploit equations which could be used to complete the proof.

A variety of additional proof facilities are available to make use of equations.

1. *asm_rewrite_tac*

   may be used to cause equations in the assumptions to rewrite the conclusion of a subgoal. This may sometimes prove sufficient to complete a proof.

2. *eq_sym_asm_tac* or *eq_sym_nth_asm_tac*

   may be used to turn round an equation in an assumption which is the wrong way round to achieve the required rewrite.

3. *var_elim_asm_tac* or *var_elim_nth_asm_tac*

   may be used to completely eliminate from the subgoal occurrences of a variable which appears on one side of an equation in the specified assumption. This causes the conclusion and all the other assumptions to be rewritten with the equation, eliminating occurrences of it. The assumption will then be discarded. These tactics will work whichever way round the equation appears in the assumption.

4. *all_var_elim_asm_tac*, *all_var_elim_asm_tac1*

   automatically eliminate from the assumptions all equations of a sufficiently simple kind, by rewriting the subgoal with them and then discarding the equations. They avoid eliminating equations where this might cause a looping rewrite. The first variant only eliminates equations where both sides are either variables or constants, the second variant will eliminate any equation of which one side is a variable which does not appear on the other side.

3.4.4 Rewriting

Rewriting using any collection of theorems from which equations are derivable is supported by the standard HOL rewriting facilities (*rewrite_tac* etc.), using Z specific preprocessing of the rewrite theorems (supplied in the Z proof contexts).

Many Z paragraphs give rise to predicates which can be used without further preparation by these standard rewriting facilities. This applies to given sets, abbreviation definitions and schema definitions.

Axiomatic descriptions, and generic axiomatic descriptions will result in equations which are likely to be effectively conditional. In such cases it is necessary to establish the applicability of the rewrite before it can be undertaken.

One way of achieving this is by forward chaining using the conditional equation after establishing the relevant condition. The relevant conditions are usually the membership assertions corresponding to the declaration part of the outer universal quantifier on the theorem to be used for rewriting.

For example, to prove the goal:

SML
```
set_pc "z_library";
set_goal([], ⓩ ∀ i:ℕ • abs i = abs ~i⌝);
```

using theorem *z_abs_thm* (which is : $\vdash \forall\ i : \mathbb{N} \bullet abs\ i = i \land abs \sim i = i$). First strip the goal:

SML
```
a (REPEAT z_strip_tac);
```

ProofPower output
```
...
(*  1 *)  ⓩ0 ≤ i⌝

(* ?⊢ *)  ⓩabs i = abs ~ i⌝
...
```

Then forward chain using the theorem and rewrite with the results:

SML
```
a (ALL_FC_T rewrite_tac [z_abs_thm]);
save_pop_thm "abs_eq_abs_minus_thm";
```

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

In more complicated cases the proof of the required conditions may be non-trivial, often because reasoning about membership of expressions formed with function application is involved. This topic is further discussed in section 4.1.1.

Z EXPRESSIONS

4.1 Expressions

Z_TERM

```
        (∗ function application: ⓩ f x⌝ ∗)

  |     ZApp        of TERM ∗ TERM   (∗ expressions ∗)

        (∗ lambda expression: ⓩ λ x:ℕ | x > 3 • x ∗ x ⌝ ∗)

  |     Zλ          of TERM          (∗ declaration ∗)
                    ∗ TERM           (∗ predicate ∗)
                    ∗ TERM           (∗ expression ∗)

        (∗ definite description: ⓩ μ x:ℕ | x ∗ x = 4 • x⌝ ∗)

  |     Zμ          of TERM          (∗ declaration ∗)
                    ∗ TERM           (∗ predicate ∗)
                    ∗ TERM           (∗ expression ∗)

        (∗ power set construction: ⓩ ℙ ℤ⌝ ∗)

  |     Zℙ          of TERM          (∗ expression ∗)

        (∗ set display: ⓩ {1,2,3,4} ⌝ ∗)

  |     ZSetd       of TYPE      (∗ HOL type of elements ∗)
                    ∗ TERM list  (∗ expressions ∗)

        (∗ set abstraction: ⓩ {x:ℤ | 1≤x≤4 • x∗x} ⌝∗)

  |     ZSeta       of TERM          (∗ declaration ∗)
                    ∗ TERM           (∗ predicate ∗)
                    ∗ TERM           (∗ expression ∗)

        (∗ tuple: ⓩ (1,2,3,4) ⌝ ∗)

  |     ZTuple      of TERM list     (∗ expressions ∗)
```

```
      (* tuple element selection: ⓩ (x,y).2⌝ *)

  |   ZSel_t       of TERM             (* expression *)
                   * int               (* element number *)

      (* cartesian product: ⓩ (ℤ × ℕ) ⌝ *)

  |   Z×           of TERM list        (* expressions *)

      (* binding display: ⓩ (people ≙ {}, age ≙ {}) ⌝ *)

  |   ZBinding     of (    string      (* component name *)
                   * TERM              (* component value *)
                   ) list

      (* theta term: ⓩ θFile′ ⌝ *)

  |   Zθ           of TERM             (* schema expression *)
                   * string            (* decoration *)

      (* binding component selection: ⓩ (a ≙ 1, b ≙ "4").b ⌝ *)

  |   ZSel_s       of TERM             (* expression *)
                   * string            (* component name *)

      (* horizontal schema expression: ⓩ [x:ℤ | x>0] ⌝ *)

  |   Z_s          of TERM             (* declaration *)
                   * TERM              (* predicate *)

      (* sequence display: ⓩ ⟨1,2,3⟩⌝ *)

  |   Z⟨⟩          of TYPE             (* type of elements *)
                   * TERM list         (* values of elements *)
```

The main new feature here is the binding display, which is important in expressing convenient proof rules.

Though absent from the first edition of the ZRM [3], binding displays have been introduced into the second edition [4] for exposition, but not as part of Z, and have now appeared in the draft standard, with two distinct concrete syntaxes neither of which corresponds to our proposal.

Bag displays and relational image, which were once treated as part of the language, can now be introduced in the library using appropriate fixity declarations.

4.1.1 Function Application

4.1.1.1 Syntax

Z_TERM
```
|        (* function application: ⓩ f x⌝ *)
|
| |      ZApp        of TERM * TERM          (* expressions *)
```

Function application may also use infix, postfix or "fancyfix" notation if an appropriate fixity paragraph has been entered. In such cases the arguments are effectively the name of the global variable and the second is the term consisting of a tuple of arguments.

SML
```
|dest_z_term ⓩf a⌝;
```

ProofPower output
```
|val it = ZApp (ⓩf⌝, ⓩa⌝) : Z_TERM
```

SML
```
|dest_z_term ⓩa ∪ b⌝;
```

ProofPower output
```
|val it = ZApp (ⓩ(_ ∪ _)⌝, ⓩ(a, b)⌝) : Z_TERM
```

In the following case the fixity declaration for bag brackets required a single argument which is a sequence (with sequence brackets elided in the concrete syntax of the bag display).

SML
```
|dest_z_term ⓩ⟦1,2,3,2,1⟧⌝;
```

ProofPower output
```
|val it = ZApp (ⓩ(⟦ ... ⟧)⌝, ⓩ⟨1, 2, 3, 2, 1⟩⌝) : Z_TERM
```

4.1.1.2 Proof Support

Applications of lambda abstractions can be eliminated by (conditional) β-conversion.

SML
```
|z_β_conv ⓩ (λ x:X | P x • f x) a⌝;
```

ProofPower output
```
|val it = P a, a ∈ X ⊢ (λ x : X | P x • f x) a = f a : THM
```

Applications may also be eliminated in favour of definite descriptions (though this is not particularly helpful).

SML
```
|z_app_conv ⓩ f a⌝;
```

ProofPower output

```
val it = ⊢ f a = μ f_a : 𝕌 | (a, f_a) ∈ f • f_a : THM
```

More commonly function applications are eliminated by rewriting with the definition of the relevant function.

Reasoning at a low level, *z_app_eq_tac* may be used to reduce an equation involving an application to sufficient conditions for its truth, in terms of the membership of the function, e.g.:

SML

```
set_goal([],ⓩ f a = v⌝);
a z_app_eq_tac;
```

ProofPower output

```
...
(* ?⊢ *)  ⓩ(∀ f_a : 𝕌 | (a, f_a) ∈ f • f_a = v) ∧ (a, v) ∈ f⌝
...
```

The first conjunct of this result is needed to ensure that f is functional at a (i.e. maps a to only one value). In the case that f is known to be a function, the theorem *z_fun_app_clauses* may be used with forward chaining, avoiding the need to prove that f is functional at a.

```
val z_fun_app_clauses =
  ⊢ ∀ f : 𝕌; x : 𝕌; y : 𝕌; X : 𝕌; Y : 𝕌
    • (f ∈ X ⇸ Y
          ∨ f ∈ X ⤔ Y
          ∨ f ∈ X ⤀ Y
          ∨ f ∈ X → Y
          ∨ f ∈ X ↣ Y
          ∨ f ∈ X ↠ Y
          ∨ f ∈ X ⤖ Y)
        ∧ (x, y) ∈ f
      ⇒ f x = y : THM
```

In this case the result $(a, v) \in f$ would have to be proven and added to the assumptions before undertaking the forward chaining, e.g.:

SML

```
drop_main_goal();
set_goal([], ⓩf ∈ ℕ ↠ ℤ ⇒ (4, ~45) ∈ f ⇒ f 4 = ~45⌝);
a (REPEAT z_strip_tac);
```

ProofPower output

```
...
(*  2 *)  ⓩf ∈ ℕ ↠ ℤ⌝
(*  1 *)  ⓩ(4, ~ 45) ∈ f⌝

(* ?⊢ *)  ⓩf 4 = ~ 45⌝

```

SML

```
a (all_fc_tac [z_fun_app_clauses]);
pop_thm();
```

ProofPower output

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

A common problem is to have to establish that the value of some expression formed by application falls within some particular set. This is often needed to establish the conditions necessary for use of a rewriting equation on the expression.

In these circumstances the theorem *z_fun_∈_clauses* may be used:

```
val z_fun_∈_clauses = ⊢ ∀ 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) : THM
```

The claim that a global variable is a member of a function space will often be obtained from the specification of the constant (as part of the predicate implicit in the declaration part of the specification). Where the function is an expression the result is likely to have been established by forward inference using similar methods.

SML

```
set_goal([], ⓩ[X](∀ b: bag X• count[X] b ∈ X → ℕ)⌝);
a (REPEAT z_strip_tac);
```

ProofPower output

```
...
(∗  1 ∗)  ⓩb ∈ bag X⌝

(∗ ?⊢ ∗)  ⓩcount[X] b ∈ X → ℕ⌝
...
```

We need the fact about *count* which is found in its defining declaration instantiated to X to make the required inference. This is added to the assumptions as follows:

SML

```
a (strip_asm_tac (z_gen_pred_elim [ⓩX⌝] (z_get_spec ⓩcount⌝)));
```

ProofPower output

```
...
(∗  3 ∗)  ⓩb ∈ bag X⌝
(∗  2 ∗)  ⓩcount[X] ∈ bag X ⤖ X → ℕ⌝
(∗  1 ∗)  ⓩ∀ x : X; B : bag X • count[X] B = (λ x : X • 0) ⊕ B⌝

(∗ ?⊢ ∗)  ⓩcount[X] b ∈ X → ℕ⌝
...
```

Assumption 1 is spurious but harmless. Next we forward chain using the theorem *z_fun_∈_clauses*, which suffices to discharge the goal.

SML
```
a (all_fc_tac [z_fun_∈_clauses]);
save_pop_thm "bag_lemma1";
```

ProofPower output
```
Tactic produced 0 subgoals:
Current and main goal achieved
...
```

Care is sometimes needed when proving membership lemmas which require intermediate results which involve constructs such as cartesian products, since most proof contexts will eliminate these.

SML
```
set_goal([],ⓩ[X](∀ b,c: bag X • ((_⊎_)[X](b, c)) ∈ bag X)⌝);
a (REPEAT strip_tac);
a (strip_asm_tac (z_gen_pred_elim [ⓩX⌝] (z_get_spec ⓩ(_⊎_)⌝)));
```

ProofPower output
```
...
(∗ 4 ∗)  ⓩb ∈ bag X⌝
(∗ 3 ∗)  ⓩc ∈ bag X⌝
(∗ 2 ∗)  ⓩ(_ ⊎ _)[X] ∈ (bag X) × (bag X) → bag X⌝
(∗ 1 ∗)  ⓩ∀ B, C : bag X; x : X
             • count ((_ ⊎ _)[X] (B, C)) x = count B x + count C x⌝

(∗ ?⊢ ∗)  ⓩ(_ ⊎ _)[X] (b, c) ∈ bag X⌝
...
```

Here the assumption $(b,c) \in (bag\ X) \times (bag\ X)$ is needed to enable the required forward chaining, but the obvious methods of obtaining this, e.g.:

SML
```
a (lemma_tac ⓩ(b,c) ∈ (bag X) × (bag X)⌝ THEN1 contr_tac);
```

have no effect since the lemma is broken up as it is added to the assumptions. This break-up can be inhibited as follows:

SML
```
a (LEMMA_T ⓩ(b,c) ∈ (bag X) × (bag X)⌝ asm_tac THEN1 contr_tac);
```

ProofPower output
```
...
(∗ 1 ∗)  ⓩ(b, c) ∈ (bag X) × (bag X)⌝

(∗ ?⊢ ∗)  ⓩ(_ ⊎ _)[X] (b, c) ∈ bag X⌝
...
```

where *asm_tac* is used instead of the default *strip_asm_tac* for processing the new assumption. Now the forward chaining will work.

SML
```
a (all_fc_tac [z_fun_∈_clauses]);
save_pop_thm "bag_lemma2";
```

ProofPower output
```
Tactic produced 0 subgoals:
Current and main goal achieved
...
```

4.1.2 Lambda Abstraction

4.1.2.1 Syntax

Z_TERM
```
        (* lambda expression ⓩ λ x:ℕ | x > 3 • x * x ⌝ *)

  |     Zλ          of TERM          (* declaration *)
                    * TERM           (* predicate *)
                    * TERM           (* expression *)
```

4.1.2.2 Proof Support

λ-abstractions when applied to arguments may be eliminated by *z_β_conv* (see 4.1.1).

Assertions about membership of λ-abstractions may be directly eliminated.

SML
```
rewrite_conv [] ⓩz ∈ (λ x:X | P x • f x)⌝;
```

ProofPower output
```
val it = ⊢ z ∈ λ x : X | P x • f x ⇔ z.1 ∈ X ∧ P z.1 ∧ f z.1 = z.2 : THM
```

Since λ-abstractions denote sets they may also be eliminated in favour of set comprehensions using *z_λ_conv*.

SML
```
z_λ_conv ⓩλ x:X | P x • f x⌝;
```

ProofPower output
```
val it = ⊢ (λ x : X | P x • f x) = {x : X | P x • (x, f x)} : THM
```

4.1.3 Definite Description

4.1.3.1 Syntax

Z_TERM
```
|        (∗ definite description ⓩ μ x:ℕ | x ∗ x = 4 • x⌝ ∗)
|
| |      Zμ            of TERM           (∗ declaration ∗)
|                      ∗ TERM            (∗ predicate ∗)
|                      ∗ TERM            (∗ expression ∗)
```

4.1.3.2 Proof Support

Definite descriptions may be eliminated using *z_μ_rule*.

SML
```
|z_μ_rule ⓩ μ x:X | P • y⌝;
```

ProofPower output
```
|val it = ⊢ ∀ x′ : 𝕌
|    • (∀ x : X | P • y = x′) ∧ (∃ x : X | P • y = x′)
|      ⇒ (μ x : X | P • y) = x′ : THM
```

4.1.4 Let Expression

4.1.4.1 Syntax

Z_TERM
```
|             (∗ let expression ⓩ let x ≙ 9 • (x, x+x) ⌝ ∗)
| | ZLet
|             of (string ∗ TERM) list      (∗ local definitions ∗)
|             ∗ TERM        (∗ expression ∗)
```

4.1.4.2 Proof Support

Let expressions may be expanded using *z_let_conv*.

SML
```
|z_let_conv ⓩ let x ≙ 9 • (x, x + x) ⌝;
```

ProofPower output
```
|val it = ⊢
|         (let x ≙ 9 • (x, x + x)) = (9, 9 + 9) : THM
```

4.1.5 The Power Set

4.1.5.1 Syntax

Z_TERM
```
|        (* power set construction, ⓩ ℙ ℤ⌝ *)
|
| |      Zℙ              of TERM              (* expression *)
```

4.1.5.2 Proof Support

Membership statements concerning power sets may be eliminated using *z_∈_ℙ_conv*, or by rewriting in proof context *z_language_ext*.

SML
```
|PC_C1 "z_language_ext" rewrite_conv[] ⓩ z ∈ ℙ y⌝;
```

ProofPower output
```
|val it = ⊢ z ∈ ℙ y ⇔ (∀ x1 : 𝕌 • x1 ∈ z ⇒ x1 ∈ y) : THM
```

4.1.6 Set Displays

4.1.6.1 Syntax

Z_TERM
```
|        (* set display, ⓩ {1,2,3,4} ⌝ *)
|
| |      ZSetd           of TYPE        (* HOL type of elements *)
|                        * TERM list    (* expressions *)
```

4.1.6.2 Proof Support

Membership statements sets displays may be eliminated using *z_∈_setd_conv*, or by rewriting in proof context *z_language*.

SML
```
|rewrite_conv[] ⓩ z ∈ {1,2,3,4,5}⌝;
```

ProofPower output
```
|val it = ⊢ z ∈ {1, 2, 3, 4, 5} ⇔
|        z = 1 ∨ z = 2 ∨ z = 3 ∨ z = 4 ∨ z = 5 : THM
```

4.1.7 Set Abstractions

4.1.7.1 Syntax

Z_TERM

```
(* set abstraction, ⓩ {x:ℤ | 1≤x≤4 • x*x} ⌝*)

| ZSeta    of TERM      (* declaration *)
           * TERM       (* predicate *)
           * TERM       (* expression *)
```

4.1.7.2 Proof Support

Statements about membership of set abstractions may be eliminated using *z_∈_seta_conv*, or by rewriting in proof context *z_language*.

A simple abstraction results in straightforward substitution into the body of the abstraction:

SML

```
rewrite_conv[] ⓩ 9 ∈ {x:ℕ | x < 12}⌝;
```

ProofPower output

```
val it = ⊢ 9 ∈ {x : ℕ | x < 12} ⇔ 9 ∈ ℕ ∧ 9 < 12 : THM
```

Where the signature is more complex tuple projections are introduced:

SML

```
rewrite_conv[]ⓩ z ∈ {x, y:ℕ | x < y}⌝;
```

ProofPower output

```
val it = ⊢ z ∈ {x, y : ℕ | x < y}
⇔ (z.1 ∈ ℕ ∧ z.2 ∈ ℕ) ∧ z ∈ (_ < _) : THM
```

Where membership is asserted of a tuple the projections are undertaken automatically.

SML

```
rewrite_conv[]ⓩ (v,w) ∈ {x, y:ℕ | x < y}⌝;
```

ProofPower output

```
val it = ⊢ (v, w) ∈ {x, y : ℕ | x < y}
⇔ (v ∈ ℕ ∧ w ∈ ℕ) ∧ v < w : THM
```

In the general case introduction of an existential is necessary, though this is avoided whenever possible.

SML

```
rewrite_conv[]ⓩ z ∈ {x, y:ℕ | x < y • x * y − x}⌝;
```

ProofPower Output

```
val it = ⊢ z ∈ {x, y : ℕ | x < y • x * y − x}
⇔ (∃ x, y : ℕ | x < y • x * y − x = z) :THM
```

4.1.8 Tuple Displays

4.1.8.1 Syntax

Z_TERM
```
          (* tuple, ⓩ (1,2,3,4) ⌝ *)

   |      ZTuple       of TERM list          (* expressions *)
```

Note that $n-$tuples for $n > 2$ are not iterated pairs, i.e. ⓩ$(1, (2, 3))$⌝ is not the same as ⓩ$(1, 2, 3)$⌝ (and doesn't have the same type either).

4.1.8.2 Proof Support

Two tuple displays are equal iff each of their respective components are equal. This fact is built into the proof context *z_language* both for rewriting and stripping assumptions or conclusions.

SML
```
rewrite_conv[] ⓩ(x,y) = (a,b)⌝;
```

ProofPower output
```
val it = ⊢ (x, y) = (a, b) ⇔ x = a ∧ y = b : THM
```

4.1.9 Tuple Element Selection

4.1.9.1 Syntax

Z_TERM
```
          (* tuple element selection, ⓩ (x,y).2⌝ *)

   |      ZSelₜ        of TERM               (* expression *)
                       * int                 (* element number *)
```

4.1.9.2 Proof Support

Conversions to effect projection from tuple displays are also built into proof context *z_language*.

SML
```
rewrite_conv[] ⓩ (x,y).1⌝;
```

ProofPower output
```
val it = ⊢ (x, y).1 = x : THM
```

4.1.10 Cartesian Products

4.1.10.1 Syntax

Z_TERM
```
        (* cartesian product, ⓩ (ℤ × ℕ) ⌝ *)

|       Z×           of TERM list        (* expressions *)
```

Note that the n-ary cartesian products for $n > 2$ are not formed by iteration of the binary cartesian product.

4.1.10.2 Proof Support

The membership conversion for n-ary cartesian products is *z_∈_×_conv* which is built into proof context *z_language*.

SML
```
rewrite_conv[] ⓩ (a, b, c) ∈ (x × y × z)⌝;
```

ProofPower output
```
val it = ⊢ (a, b, c) ∈ x × y × z
        ⇔ a ∈ x ∧ b ∈ y ∧ c ∈ z : THM
```

Cartesian products may also be converted into set abstractions using *z_×_conv*.

SML
```
z_×_conv     ⓩ (x × y × z)⌝;
```

ProofPower output
```
val it = ⊢ x × y × z = {t1 : x; t2 : y; t3 : z} : THM
```

Extensional proof contexts incorporate an extensional understanding of equality of cartesian products:

SML
```
PC_C1 "z_language_ext"
rewrite_conv[] ⓩ (x × y × z) = (x' × y' × z')⌝;
```

ProofPower output
```
val it = ⊢ x × y × z = x' × y' × z'
    ⇔ (∀ x1 : 𝕌; x2 : 𝕌; x3 : 𝕌
      • x1 ∈ x ∧ x2 ∈ y ∧ x3 ∈ z
      ⇔ x1 ∈ x' ∧ x2 ∈ y' ∧ x3 ∈ z') : THM
```

Such an equation can also be demonstrated by rewriting if each of the respective components can be proven equal, however this is not a necessary condition for the equality (since any single empty component set will render the cartesian product empty).

4.1.11 Binding Displays

4.1.11.1 Syntax

Z_TERM

```
        (∗ binding ⓩ (people ≙ {}, age ≙ {}) ⌝ ∗)

 |      ZBinding    of (    string          (∗ component name ∗)
                    ∗ TERM                  (∗ component value ∗)
                    ) list
```

4.1.11.2 Proof Support

Two binding displays are equal iff each of their respective components are equal. This fact is built into the proof context *z_language* for rewriting and for stripping assumptions and conclusions.

SML

```
rewrite_conv[] ⓩ (x ≙ a, y ≙ b) = (y ≙ d, x ≙ c)⌝;
```

ProofPower output

```
val it = ⊢ (x ≙ a, y ≙ b) = (x ≙ c, y ≙ d) ⇔ a = c ∧ b = d : THM
```

4.1.12 Theta Terms

4.1.12.1 Syntax

Z_TERM

```
        (∗ theta term ⓩ θFile′ ⌝ ∗)

 |      Zθ          of TERM                 (∗ schema expression ∗)
                    ∗ string                (∗ decoration ∗)
```

The extended syntax allows arbitrary expressions of appropriate type in place of the schema reference usually required.

4.1.12.2 Proof Support

Theta terms may be though of as abbreviations for explicit binding constructions. Rewriting with *z′θ_def* will reveal the underlying binding construction:

SML

```
rewrite_conv[z′θ_def] ⓩ θFile′⌝;
```

ProofPower output

```
val it = ⊢ θFile′ = (age ≙ age′, people ≙ people′) : THM
```

Alternatively *z_θ_conv* may be used to secure the same effect:

SML

```
z_θ_conv ⓩ θFile′⌝;
```

ProofPower output

```
val it = ⊢ θFile′ = (age ≙ age′, people ≙ people′) : THM
```

In most respects θ-terms are treated in the same way as binding displays.

4.1.13 Binding Component Selection

4.1.13.1 Syntax

Z_TERM

```
        (∗ binding component selection ⓩ (a ≙ 1, b ≙ "4").b ⌝ ∗)

|       ZSelₛ        of TERM          (∗ expression ∗)
                     ∗ string         (∗ component name ∗)
```

The $_s$ here and in the following is entered into the source document as:

```
                     ⋎s
```

The 'subscript-shift character', ⋎, here may be obtained from the palette of mathematical symbols or typed directly (as Meta+tab under SunView, or as Meta+d when using `xpp`).

4.1.13.2 Proof Support

Projection from binding displays is built in to proof context *z_language*.

SML

```
rewrite_conv[] ⓩ (x ≙ a, y ≙ b).y⌝;
```

ProofPower output

```
val it = ⊢ (x ≙ a, y ≙ b).y = b : THM
```

Projection from theta terms is also built in to proof context *z_language*.

SML

```
rewrite_conv[] ⓩ (θFile′).age⌝;
```

ProofPower output

```
val it = ⊢ (θFile′).age = age′ : THM
```

4.1.14 Horizontal Schemas

4.1.14.1 Syntax

Z_TERM

```
        (∗ horizontal schema expression: ⓩ [x:ℤ | x>0] ⌝ ∗)

|       Zₛ           of TERM          (∗ declaration ∗)
                     ∗ TERM           (∗ predicate ∗)
```

4.1.14.2 Proof Support

The basic rule for horizontal schemas is the conversion *z_∈_horiz_schema_conv1*, which is built into the standard rewrites for proof context *z_language*.

SML
```
rewrite_conv[]⌜z ∈ [x:ℤ;y:ℕ]⌝;
```
ProofPower output
```
val it = ⊢ z ∈ [x : ℤ; y : ℕ]
        ⇔ z.x ∈ ℤ ∧ z.y ∈ ℕ : THM
```

Where a binding display or theta term is used the selections take place automatically.

SML
```
rewrite_conv[]⌜(x ≙ a, y ≙ b) ∈ [x:ℤ;y:ℕ]⌝;
```
ProofPower output
```
val it = ⊢ (x ≙ a, y ≙ b) ∈ [x : ℤ; y : ℕ]
        ⇔ a ∈ ℤ ∧ b ∈ ℕ : THM
```

4.1.15 Sequence Displays

4.1.15.1 Syntax

Z_TERM
```
        (* sequence display: ⌜ ⟨1,2,3⟩⌝ *)

  |     Z⟨⟩           of TYPE          (* type of elements *)
                      * TERM list      (* values of elements *)
```

4.1.15.2 Proof Support

The basic rules for sequence displays are the conversion *z_⟨⟩_conv* and *z_∈_⟨⟩_conv*, which differ only in that the latter will trigger only for membership assertions. *z_∈_⟨⟩_conv* is built into the standard rewrites for proof context *z_language*.

SML
```
z_⟨⟩_conv ⌜⟨a,b,c⟩⌝;
```
ProofPower output
```
val it = ⊢ ⟨a, b, c⟩ = {(1, a), (2, b), (3, c)} : THM
```

In the context of a membership assertion, rewriting in the proof context *z_language* performs the same elimination:

SML
```
once_rewrite_conv[]⌜z ∈ ⟨a,b,c⟩⌝;
```
ProofPower output
```
val it = ⊢ z ∈ ⟨a, b, c⟩ ⇔
        z ∈ {(1, a), (2, b), (3, c)} : THM
```

Which (without the "once") is further reduced as follows:

SML
```
rewrite_conv[]⌜z ∈ ⟨a,b,c⟩⌝;
```
ProofPower output
```
val it = ⊢ z ∈ ⟨a, b, c⟩ ⇔
        z = (1, a) ∨ z = (2, b) ∨ z = (3, c) : THM
```

4.2 Schema Expressions

z_term

```
(* schema negation: ⓩ(¬ File)⊕⊕𝕌⌝ *)

| Z¬s          of TERM           (* schema expression *)

(* schema conjunction: ⓩ(File ∧ File2)⊕⊕𝕌⌝ *)

| Z∧s          of TERM * TERM    (* schema expressions *)

(* schema disjunction: ⓩ(File ∨ File2)⊕⊕𝕌⌝ *)

| Z∨s          of TERM * TERM    (* schema expressions *)

(* schema implication ⓩ(File ⇒ File2)⊕⊕𝕌⌝ *)

| Z⇒s          of TERM * TERM    (* schema expressions *)

(* schema equivalence: ⓩ(File ⇔ File2)⊕⊕𝕌⌝ *)

| Z⇔s          of TERM * TERM    (* schema expressions *)

(* schema existential: ⓩ(∃ File3 | people = {} • File2)⊕⊕𝕌⌝ *)

| Z∃s          of TERM           (* declaration *)
               * TERM            (* predicate *)
               * TERM            (* schema expression *)

(* schema unique existential: ⓩ(∃1 File3 | people = {} • File2)⊕⊕𝕌⌝ *)

| Z∃1s         of TERM           (* declaration *)
               * TERM            (* predicate *)
               * TERM            (* schema expression *)

(* schema universal: ⓩ(∀ File3 | people = {} • File2)⊕⊕𝕌⌝ *)

| Z∀s          of TERM           (* declaration *)
               * TERM            (* predicate *)
               * TERM            (* schema expression *)

(* decoration: ⓩ File ″⌝ *)

| ZDecors      of TERM           (* schema expression *)
               * string          (* decoration *)
```

```
        (* pre−condition: ⓩ pre FileOp⌝ *)

|       ZPre_s       of TERM             (* schema expression *)

        (* schema hiding: ⓩ FileOp \_s (age, i?)⌝ *)

|       ZHide_s      of TERM             (* schema expression *)
                     * string list       (* component names *)

        (* schema renaming: ⓩFile [aged/age, input/i?]⌝ *)

|       ZRename_s    of TERM             (* schema expression *)
                     * (string * string) list(* rename list *)

        (* schema projection: ⓩFileOp ↾_s File⌝*)

|       Z↾_s         of TERM * TERM      (* schema expressions *)

        (* schema composition: ⓩΔFile ⨾_s ΔFile⌝ *)

|       Z⨾_s    of TERM * TERM  (* schema expressions *)

        (* delta operation: ⓩΔFile⌝ *)

|       ZΔ_s         of TERM             (* schema expression *)

        (* Ξ operation: ⓩΞFile⌝ *)

|       ZΞ_s         of TERM             (* schema expression *)
;
```

Note here that though the logical operators have been overloaded, at present the system does not support the overloading of other schema operators which clash with names in the Z ToolKit. For these operators ($⨾_s$, $\upharpoonright_s$, $\setminus_s$), the name subscripted with s has been used for the schema operator.

4.2.1 Schema Negation

4.2.1.1 Syntax

```
Z_TERM
        (* schema negation ⓩ(¬ File)⊕⊕U⌝ *)

|       Z¬_s         of TERM      (* schema expression *)
```

A negation occurring at the outermost level in a term quotation is interpreted as a logical negation rather than a schema negation, unless a cast is applied.

4.2.1.2 Proof Support

A binding is an element of the schema negation of a schema iff it is not an element of the schema.

This rule is captured by the conversion *z_∈_¬$_s$_conv*, which is built into the standard rewrites for proof context *z_language*.

SML
```
rewrite_conv[]ⓩz ∈ (¬ File)⌝;
```

ProofPower output
```
val it = ⊢ z ∈ (¬ File) ⇔ ¬ z ∈ File : THM
```

4.2.2 Schema Conjunction

4.2.2.1 Syntax

Z_TERM
```
        (* schema conjunction: ⓩ(File ∧ File2)⊕𝕌⌝ *)

  |     Z∧s             of TERM * TERM  (* schema expressions *)
```

The two operands must be schema expressions with compatible types.

A conjunction occurring at the outermost level in a term quotation is interpreted as a logical conjunction rather than a schema conjunction, unless a cast is applied.

4.2.2.2 Proof Support

A binding is an element of the schema conjunction of two schemas iff both the projections of the binding to the signatures of the operands are elements of the corresponding operand schemas.

This rule is captured by the conversion *z_∈_∧$_s$_conv*, which is built into the standard rewrites for proof context *z_language*.

SML
```
rewrite_conv[] ⓩz ∈ (File ∧ File2)⌝;
```

ProofPower output
```
val it = ⊢ z ∈ (File ∧ File2)
    ⇔ (age ≙ z.age, people ≙ z.people) ∈ File
      ∧ (height ≙ z.height, people ≙ z.people) ∈ File2 : THM
```

4.2.3 Schema Disjunction

4.2.3.1 Syntax

Z_TERM
```
        (* schema disjunction: ⓩ(File ∨ File2)⊕𝕌⌝ *)

  |     Z∨s             of TERM * TERM  (* schema expressions *)
```

The two operands must be schema expressions with compatible types.

A disjunction occurring at the outermost level in a term quotation is interpreted as a logical disjunction rather than a schema disjunction, unless a cast is applied.

4.2.3.2 Proof Support

A binding (of appropriate type) is an element of the schema disjunction of two schemas iff either of the projections of the binding to the signatures of the operands are elements of the corresponding operand schemas.

This rule is captured by the conversion *z_∈_∨$_s$_conv*, which is built into the standard rewrites for proof context *z_language*.

SML
```
rewrite_conv[] ⓩz ∈ (File ∨ File2)⌝;
```

ProofPower output
```
val it = ⊢ z ∈ (File ∨ File2)
    ⇔ (age ≙ z.age, people ≙ z.people) ∈ File
      ∨ (height ≙ z.height, people ≙ z.people) ∈ File2 : THM
```

4.2.4 Schema Implication

4.2.4.1 Syntax

Z_TERM
```
        (* schema implication ⓩ(File ⇒ File2)ⓈⓊ⌝ *)

  |     Z⇒s             of TERM * TERM  (* schema expressions *)
```

The two operands must be schema expressions with compatible types.

An implication occurring at the outermost level in a term quotation is interpreted as a logical implication rather than a schema implication, unless a cast is applied.

4.2.4.2 Proof Support

A binding (of appropriate type) is an element of the schema implication of two schemas iff whenever the projection of the binding to the signature of the first operand is an element of the corresponding operand schemas, the projections of the binding to the signature of the second operand is also an element of the second operand schema.

This rule is captured by the conversion *z_∈_⇒$_s$_conv*, which is built into the standard rewrites for proof context *z_language*.

SML
```
rewrite_conv[] ⓩz ∈ (File ⇒ File2)⌝;
```

ProofPower output
```
val it = ⊢ z ∈ (File ⇒ File2)
    ⇔ (age ≙ z.age, people ≙ z.people) ∈ File
      ⇒ (height ≙ z.height, people ≙ z.people) ∈ File2 : THM
```

4.2.5 Schema Equivalence

4.2.5.1 Syntax

Z_TERM
```
      (∗ schema equivalence: ⌜Z(File ⇔ File2)⊕⊕𝕌⌝ ∗)

  |   Z⇔s          of TERM ∗ TERM   (∗ schema expressions ∗)
```

The two operands must be schema expressions with compatible types.

An equivalence occurring at the outermost level in a term quotation is interpreted as a logical equivalence rather than a schema equivalence, unless a cast is applied.

4.2.5.2 Proof Support

A binding (of appropriate type) is an element of the schema equivalence of two schemas iff the projection of the binding to the signature of the first operand is an element of the corresponding operand schema iff the projection of the binding to the signature of the second operand is an element of the second operand schema.

This rule is captured by the conversion *z_∈_⇔$_s$_conv*, which is built into the standard rewrites for proof context *z_language*.

SML
```
rewrite_conv[]⌜Z z ∈ (File ⇔ File2)⌝;
```

ProofPower output
```
val it = ⊢ z ∈ (File ⇔ File2)
    ⇔ (age ≙ z.age, people ≙ z.people) ∈ File
    ⇔ (height ≙ z.height, people ≙ z.people) ∈ File2 : THM
```

4.2.6 Schema Existential

4.2.6.1 Syntax

Z_TERM
```
      (∗ schema existential: ⌜Z(∃ File3 | people = {} • File2)⊕⊕𝕌⌝ ∗)

  |   Z∃s          of TERM          (∗ declaration ∗)
                   ∗ TERM           (∗ predicate ∗)
                   ∗ TERM           (∗ schema expression ∗)
```

The last operand must be a schema expressions with type compatible with the signature of the declaration. The signature of the declaration part must be contained in the signature of the body.

An existential occurring at the outermost level in a term quotation is interpreted as a logical existential rather than a schema existential, unless a cast is applied.

4.2.6.2 Proof Support

The basic rule for schema existentials is the conversion *z_∈_∃$_s$_conv*, which is built into the standard rewrites for proof context *z_language*.

SML
```
rewrite_conv[]ⓩz ∈ (∃ File3 | people = {} • File2)⌝;
```

ProofPower output
```
val it = ⊢ z ∈ (∃ File3 | people = {} • File2)
    ⇔ (∃ x1 : 𝕌
      • ((people ≙ x1.people) ∈ File3
        ∧ x1.people = {})
        ∧ (height ≙ z.height, people ≙ x1.people) ∈ File2) : THM
```

4.2.7 Schema Unique Existence

4.2.7.1 Syntax

Z_TERM
```
        (* schema unique existential: ⓩ(∃₁ File3 | people = {} • File2)⦂𝕌⌝ *)

  |     Z∃₁ₛ        of TERM           (* declaration *)
                    * TERM            (* predicate *)
                    * TERM            (* schema expression *)
```

The last operand must be a schema expressions with type compatible with the signature of the declaration. The signature of the declaration part must be contained in the signature of the body.

A unique existential occurring at the outermost level in a term quotation is interpreted as a logical existential rather than a schema existential, unless a cast is applied.

4.2.7.2 Proof Support

The basic rule for schema existentials is the conversion *z_∈_∃$_{1s}$_conv*, which is built into the standard rewrites for proof context *z_language*.

SML
```
rewrite_conv[]ⓩz ∈ (∃₁ File3 | people = {} • File2)⌝;
```

ProofPower output
```
val it = ⊢ z ∈ (∃₁ File3 | people = {} • File2)
    ⇔ (∃₁ x1 : 𝕌
      • ((people ≙ x1.people) ∈ File3
        ∧ x1.people = {})
        ∧ (height ≙ z.height, people ≙ x1.people) ∈ File2) : THM
```

4.2.8 Schema Universal

4.2.8.1 Syntax

Z_TERM

```
(∗ schema universal: ⓩ(∀ File3 | people = {} • File2)⊕⊕𝕌⌝ ∗)

| Z∀s         of TERM          (∗ declaration ∗)
              ∗ TERM           (∗ predicate ∗)
              ∗ TERM           (∗ schema expression ∗)
```

The last operand must be a schema expressions with type compatible with the signature of the declaration. The signature of the declaration part must be contained in the signature of the body.

A universal occurring at the outermost level in a term quotation is interpreted as a logical universal rather than a schema universal, unless a cast is applied.

4.2.8.2 Proof Support

The basic rule for schema universals is the conversion *z_∈_∀$_s$_conv*, which is built into the standard rewrites for proof context *z_language*.

SML

```
rewrite_conv[]ⓩz ∈ (∀ File3 | people = {} • File2)⌝;
```

ProofPower output

```
val it = ⊢ z ∈ (∀ File3 | people = {} • File2)
    ⇔ (∀ x1 : 𝕌
      • (people ≙ x1.people) ∈ File3 ∧ x1.people = {}
        ⇒ (height ≙ z.height, people ≙ x1.people) ∈ File2) : THM
```

4.2.9 Decoration

4.2.9.1 Syntax

Z_TERM

```
(∗ decoration: ⓩ File ″⌝ ∗)

| ZDecors     of TERM          (∗ schema expression ∗)
              ∗ string         (∗ decoration ∗)
```

4.2.9.2 Proof Support

The operation of decoration is extensionally characterised by *z_∈_dec$_s$_conv* which is built into the proof context *z_language*.

SML

```
rewrite_conv[]ⓩz ∈ File ″⌝;
```

ProofPower output

```
val it = ⊢ z ∈ (File″) ⇔ (age ≙ z.age″, people ≙ z.people″) ∈ File : THM
```

4.2.10 Pre-Condition

4.2.10.1 Syntax

Z_TERM

```
|        (* pre−condition: ⓩ pre FileOp⌝ *)
|
| |      ZPres          of TERM            (* schema expression *)
```

4.2.10.2 Proof Support

The operation of forming a pre-condition is extensionally characterised by *z_∈_pre$_s$_conv* which is built into the proof context *z_language*.

SML

```
|once_rewrite_conv[]ⓩz ∈ (pre FileOp)⌝;
```

ProofPower output

```
|val it = ⊢ z ∈ (pre FileOp) ⇔
| z ∈    [age : 𝕌; i? : 𝕌; people : 𝕌
|          | ∃ age′ : 𝕌; people′ : 𝕌 • FileOp] : THM
```

Normally the membership of the horizontal schema thus introduced would be immediately eliminated. Where a binding display or theta term is used selections are eliminated.

SML

```
|rewrite_conv[]ⓩ(age ≙ age, i? ≙ i?, people ≙ people) ∈ (pre FileOp)⌝;
```

ProofPower output

```
|val it = ⊢ (age ≙ age, i? ≙ i?, people ≙ people) ∈ (pre FileOp)
|    ⇔ (∃ age′ : 𝕌; people′ : 𝕌 • FileOp) : THM
```

4.2.11 Schema Hiding

4.2.11.1 Syntax

Z_TERM

```
|        (* schema hiding: ⓩ FileOp \s (age, i?)⌝  *)
|
| |      ZHides         of TERM            (* schema expression *)
|                       * string list      (* component names *)
```

This is entered into the source document as:

```
|ⓩ FileOp \⋎s (age, i?)⌝
```

4.2.11.2 Proof Support

This is characterised by $z\_\in\_hide_s\_conv$ which is built into the proof context $z\_language$.

SML
```
once_rewrite_conv[]ⓩz ∈ (File \s (age))⌝;
```

ProofPower output
```
val it = ⊢ z ∈ (File \s (age)) ⇔ z ∈ [people : 𝕌 | ∃ age : 𝕌 • File] : THM
```

Normally further membership eliminations, and possibly selection eliminations, will take place:

SML
```
rewrite_conv[]ⓩ(people ≙ people) ∈ (File \s (age))⌝;
```

ProofPower output
```
val it = ⊢ (people ≙ people) ∈ (File \s (age)) ⇔ (∃ age : 𝕌 • File) : THM
```

4.2.12 Schema Renaming

4.2.12.1 Syntax

Z_TERM
```
        (* schema renaming: ⓩFile [aged/age, input/i?]⌝ *)

  |     ZRenames   of TERM            (* schema expression *)
                   * (string * string) list(* rename list *)
```

4.2.12.2 Proof Support

Schema renaming is extensionally characterised by $z\_\in\_rename_s\_conv$ which is built into the proof context $z\_language$.

SML
```
once_rewrite_conv[]ⓩz ∈ File[aged/age]⌝;
```

ProofPower output
```
val it = ⊢ z ∈ (File [aged/age]) ⇔
        (age ≙ z.aged, people ≙ z.people) ∈ File : THM
```

Normally further membership eliminations, and possibly selection eliminations, will take place:

SML
```
rewrite_conv[]ⓩ(aged ≙ age, people ≙ people) ∈ File[aged/age]⌝;
```

ProofPower output
```
val it = ⊢ (aged ≙ age, people ≙ people) ∈ (File [aged/age]) ⇔ File : THM
```

4.2.13 Schema Projection

4.2.13.1 Syntax

Z_TERM
```
│        (∗ schema projection: ⓩFileOp ↾ₛ File⌝∗)
│
│ |      Z↾s            of TERM ∗ TERM  (∗ schema expressions ∗)
```

4.2.13.2 Proof Support

Schema projection is extensionally characterised by *z_∈_↾ₛ_conv*, which is built into proof context *z_language*.

SML
```
│once_rewrite_conv[]ⓩz ∈ (FileOp ↾ₛ File)⌝;
```

ProofPower output
```
│val it = ⊢ z ∈ (FileOp ↾ₛ File)
│    ⇔ z ∈ ((FileOp ∧ File) \ₛ (age′, i?, people′)) : THM
```

Normally further membership eliminations, and possibly selection eliminations, will take place:

SML
```
│rewrite_conv[]ⓩ(age ≘ age, people ≘ people) ∈ (FileOp ↾ₛ File)⌝;
```

ProofPower output
```
│val it = ⊢ (age ≘ age, people ≘ people) ∈ (FileOp ↾ₛ File)
│    ⇔ (∃ age′ : 𝕌; i? : 𝕌; people′ : 𝕌 • FileOp ∧ File) : THM
```

4.2.14 Schema Composition

4.2.14.1 Syntax

Z_TERM
```
│        (∗ schema composition: ⓩFileOp ⨾ₛ FileOp⌝ ∗)
│
│ |      Z⨾s     of TERM ∗ TERM  (∗ schema expressions ∗)
```

4.2.14.2 Proof Support

Schema projection is extensionally characterised by *z_∈_⨾ₛ_conv*, which is built into proof context *z_language*.

SML
```
│once_rewrite_conv[]ⓩz ∈ (FileOp ⨾ₛ FileOp)⌝;
```

ProofPower output
```
val it = ⊢ z ∈ (FileOp ⨾s FileOp)
    ⇔ z
      ∈ [age : 𝕌; i? : 𝕌; people : 𝕌; age′ : 𝕌; people′ : 𝕌
        | ∃ x1 : 𝕌; x2 : 𝕌
          • (age ≙ age, age′ ≙ x1, i? ≙ i?, people ≙ people, people′ ≙ x2)
              ∈ FileOp
            ∧ (age ≙ x1, age′ ≙ age′, i? ≙ i?, people ≙ x2,
                  people′ ≙ people′)
              ∈ FileOp] : THM
```

Normally further membership eliminations, and possibly selection eliminations, will take place:

SML
```
rewrite_conv[]⌜z ∈ (FileOp ⨾s FileOp)⌝;
```

ProofPower output
```
val it = ⊢ z ∈ (FileOp ⨾s FileOp)
    ⇔ (∃ x1 : 𝕌; x2 : 𝕌
      • (age ≙ z.age, age′ ≙ x1, i? ≙ z.i?, people ≙ z.people, people′ ≙ x2)
          ∈ FileOp
        ∧ (age ≙ x1, age′ ≙ z.age′, i? ≙ z.i?, people ≙ x2,
              people′ ≙ z.people′)
          ∈ FileOp) : THM
```

4.2.15 Δ

4.2.15.1 Syntax

Z_TERM
```
        (* delta operation: ⌜ΔFile⌝ *)

|       ZΔs         of TERM         (* schema expression *)
```

Delta is currently supplied in ProofPower-Z as a schema operator rather than a convention.

4.2.15.2 Proof Support

Delta is extensionally characterised by *z_∈_Δs_conv*, which is built into proof context *z_language*.

SML
```
once_rewrite_conv[]⌜z ∈ (ΔFile)⌝;
```

ProofPower output
```
val it = ⊢ z ∈ (Δ File) ⇔ z ∈ [File; File′] : THM
```

Normally further membership eliminations, and possibly selection eliminations, will take place:

SML
```
rewrite_conv[]⌜z ∈ (ΔFile)⌝;
```

ProofPower output
```
val it = ⊢ z ∈ (Δ File)
    ⇔ (age ≙ z.age, people ≙ z.people) ∈ File
      ∧ (age ≙ z.age′, people ≙ z.people′) ∈ File : THM
```

4.2.16 Ξ

4.2.16.1 Syntax

Z_TERM
```
        (* Ξ operation: ⌜ΞFile⌝ *)

| ZΞs          of TERM          (* schema expression *)
```

Rather than a convention, ProofPower-Z currently provides an operator.

4.2.16.2 Proof Support

This operation is extensionally characterised by *z_∈_Ξ$_s$_conv*, which is built into proof context *z_language*.

SML
```
once_rewrite_conv[]⌜z ∈ (ΞFile)⌝;
```

ProofPower output
```
val it = ⊢ z ∈ (Ξ File) ⇔ z ∈ [File; File′ | θFile = θFile′] : THM
```

Normally further membership eliminations, and possibly selection eliminations, will take place:

SML
```
rewrite_conv[]⌜z ∈ (ΞFile)⌝;
```

ProofPower output
```
val it = ⊢ z ∈ (Ξ File)
    ⇔ ((age ≙ z.age, people ≙ z.people) ∈ File
      ∧ (age ≙ z.age′, people ≙ z.people′) ∈ File)
      ∧ z.age = z.age′
      ∧ z.people = z.people′ : THM
```

Z PARAGRAPHS

5.1 Introduction

- Fixity declaration
- Given set definition
- Structured set definition
- Axiomatic definition
- Constraint
- Generic definition
- Abbreviation definition
- Schema boxes

5.1.1 Syntax

Unlike lower level constructs in the Z language, such as predicates and expressions, paragraphs are not represented as HOL terms, though predicates arising from the paragraphs are represented as terms.

Paragraphs in Z are mapped to the facilities in HOL for introducing new constants and types. Consequently, they are not entered using the Z term quotes '$\ulcorner_Z$' and '$\urcorner$', and they are not to be found in the structure of the datatype *Z_TERM*.

In the following description of Z paragraphs in ProofPower the syntax sections show how these paragraphs are rendered in the source documents, and omit any details of abstract syntax.

In general Z paragraphs are entered in documents using one are other of the paragraph introduction symbols, Ⓢ*Z* or Ⓢ*ZAX*, at the beginning of a new line, and are terminated with the character ■ on its own on a line. The characters Ⓢ, ■ may be obtained from the palette of extended characters or typed directly (as Meta+'(' and Meta+')'). However in some cases (schemas, and generics) the characters used to form the relevant boxes suffice to introduce the paragraphs.

5.1.2 Paragraph Processing Modes and Flags

There are several different modes of processing Z paragraphs which are controlled by flags.

- **Type-check-only Mode**

  If the flag *z_type_check_only* is set to *true* then only type checking of Z paragraphs is performed.

This makes the response faster, and permits greater flexibility in amending paragraphs. This mode is suitable for use while developing specifications prior to undertaking any proof work.

In type-check-only mode the definitions of global variables are not saved, only their types. This makes it possible to change the definitions without reprocessing all subsequent definitions. No diagnostics are given in type-check-only mode if a global variable is redefined, though a diagnostic still arises if a new type is entered twice. This facility is only available in theories which have no children.

When a global variable is redefined global variables depending on it are not re-checked, and so it is desirable to re-check the specification as a whole when the changes are complete. If it is required to reason about a specification which has been entered in type-check-only mode then the specification will have to be re-processed with *z_type_check_only* set false. This would normally be done by extracting the specification from the source document using *docsml* and loading it using *use_file*.

- **Axiomatic Mode**

  If the flag *z_use_axioms* is set to true (and *z_type_check_only* is set to *false*) then axiomatic descriptions and free-type descriptions are introduced using axioms.

- **Conservative Mode**

  If both the above flags are set *false* then all Z axiomatic descriptions are introduced using the ProofPower *new_specification* facility, i.e. by conservative extension.

  Consistency proof obligations, unless discharged automatically, will have to be discharged by the user.

  In a future release it is hoped that free-types will also be supported by conservative extension.

5.2 Paragraphs

5.2.1 Fixity Declarations

The fixity paragraphs currently supported by ProofPower generalise a facility required to avoid special treatment to constants introduced in the Z-ToolKit such as relational image and bag brackets. This more general facility has been proposed for inclusion in the standard under development[10].

Fixity declarations may be entered for:

- functions, e.g.:

  z

  |*fun 10 twice _*

  z

  |*fun select ... from _*

- generics, e.g.:

  z

  |*gen _ swap _*

- relations, e.g.:

  z
  $|\mathit{rel}\ \_\ \mathit{is\_even}$

In each of these cases:

- The first word is a keyword indicating the kind of fixity declaration involved, which must be *fun*, *gen* or *rel*.
- The next element is an optional numeric value, which is the precedence of the name or template. This is not permitted in a rel fixity paragraph.
- The final part of the declaration is a template, showing the form of the 'name' and the position and kind of the parameters.
  - '_' is a place for a parameter
  - '...' is a place for a sequence of parameters (with sequence brackets elided)

Where a name or template is declared using a 'fun' fixity paragraph three possible methods of subsequent use are possible.

Firstly, the template may be used in exactly the form used in the fixity declaration in the defining declaration of a global or local variable.

Secondly, within the scope of such a declaration or as a free variable, the template may be applied by substituting expressions in place of the formal parameter markers in the template. In this case the resulting expression is interpreted as a function application, where the variable whose name is the template is applied to a tuple formed from the actual arguments.

Finally the template may be used verbatim as an expression, if enclosed in brackets. This is necessary if the template is declared as a generic global variable and it is necessary to supply actual generic parameters rather than accept the set of all elements of the type inferred by the type inferrer.

Where a name or template is declared using a 'gen' fixity paragraph the forms of use are similar to those described for 'fun' fixity paragraphs above except that:

- the expressions substituted for the formal parameter markers must be sets (the formal parameter may not be '...').
- the expression is interpreted as an instance of the generic rather than an application of the variable.

Where a name or template is declared using a 'rel' fixity paragraph the forms of use are similar to those for 'fun' fixity paragraph except that the variable represented by the name or template must denote a set, and the expression involving use of this name or template will be interpreted as a set membership assertion (provided expressions are supplied in place of the formal parameter markers).

Fixity clauses can be deleted only by deleting the theory they are contained in.

5.2.2 Given Sets

5.2.2.1 Syntax

Given sets are introduced as a list of names enclosed in square brackets. Typed in as:

```
    ⓈZ
    [G1, G2]
    ■
```

The printed form is:

Z
```
[G1, G2]
```

5.2.2.2 Proof Support

Each given set causes the introduction of a new type and a new global variable known to be the set of all elements of that type.

The specification of the given set may be retrieved as follows:

SML
```
val G1_def = z_get_spec ⓩG1⌝;
```

ProofPower output
```
val G1_def = ⊢ G1 = 𝕌 : THM
```

'𝕌' is the generic identity function, and when its actual generic parameter is not printed it may be assumed to be the set of all elements of the relevant (inferred) type. The normal proof contexts have knowledge of '𝕌' so that rewriting with the specification of a given set will enable assertions about membership of the given set to be proven.

SML
```
rewrite_conv [G1_def] ⓩx ∈ G1⌝;
```

ProofPower output
```
val it = ⊢ x ∈ G1 ⇔ true : THM
```

A special facility is provided to enable the information in the given set declarations in a theory to be collected for inclusion in a proof context (so that it may be used in rewriting). *theory_u_simp_eqn_cxt* will extract an equational context from a named theory incorporating conversions which prove results of the form:

```
        x ∈ GIVENSET ⇔ true
```

for each of the given sets declared in the theory.

This may be made into a partial proof context and then merged to form a new proof context, e.g.:

SML
```
new_pc "'usr011";
set_u_simp_eqn_cxt (theory_u_simp_eqn_cxt "usr011")
        "'usr011";
set_merge_pcs ["'usr011", "z_language"];
```

(note here that *'usr011* should come first in the list)

Now knowledge of the given sets defined in the theory "usr011" is built into the rewriting facilities:

SML
```
rewrite_conv[] ⓩx ∈ NAME⌝;
```

ProofPower output
```
val it = ⊢ x ∈ NAME ⇔ true : THM
```

5.2.3 Abbreviation and Generic Definitions

5.2.3.1 Syntax

Typed in as:

```
        ⓈZ
        GPROD ≙ G1 × G2
        ■
```

Displayed as:

Z
```
GPROD ≙ G1 × G2
```

Similarly generics:

Z
```
X swap Y ≙ Y × X
```

and schema definitions:

Z
```
SCHEMA ≙ [x, y :G1 | ¬ x = y]
```

The ProofPower system currently uses '≙', not only for schema definitions, but also instead of '==' for abbreviation definitions and generic definitions. This reflects a proposal to the Z standardisation panel to eliminate unnecessary distinctions between global variables denoting schemas and those denoting other types of value.

The fixity status of the relevant name or template should be declared beforehand.

5.2.3.2 Proof Support

These definitions will generally yield equations when retrieved using *z_get_spec*.

SML
```
│val gprod_def = z_get_spec ⓩGPROD⌝;
```

ProofPower output
```
│val gprod_def = ⊢ GPROD = G1 × G2 : THM
```

If the declaration is of a generic global variable then the resulting predicate will also be generic.

SML
```
│val swap_def = z_get_spec ⓩ(_ swap _)⌝;
```

ProofPower Output
```
│val swap_def = ⊢ [X, Y](X swap Y = Y × X) : THM
```

Specialisation of such a generic predicate will be done automatically by the rewriting facilities when required:

SML
```
│rewrite_conv [swap_def] ⓩℤ swap ℕ⌝;
```

ProofPower Output
```
│val it = ⊢ ℤ swap ℕ = ℕ × ℤ : THM
```

Specialisation may also be done using *∀_elim* or *list_∀_elim*.

Schema declarations may be used in a similar manner, using the rewriting facilities to expand an occurence of the schema name.

SML
```
│rewrite_conv [z_get_spec ⓩSCHEMA⌝] ⓩSCHEMA ∨ x = y⌝;
```

ProofPower output
```
│val it = ⊢ SCHEMA ∨ x = y ⇔ ¬ x = y ∨ x = y : THM
```

(Note here that the predicate implicit in the declaration part of the horizontal schema has been simplified away because we inserted the given set declarations into the current proof context.)

5.2.4 Schema Boxes

5.2.4.1 Syntax

A schema box is typed in as :

```
│      ┌Sch───────────────
│      │     x, y : ℤ;
│      │     z : ℕ
│      ├──────────────
│      │     x = y ∨ y = z
│      └────────────────────
│
```

resulting in the printed text:

```
Z
┌─Sch──────────────────────────────────────────
│     x, y : ℤ;
│     z : ℕ
├──────────────────────
│     x = y ∨ y = z
└──────────────────────────────────────────────
```

The dividing horizontal bar and the following predicate are optional.

The horizontal and vertical bar characters in the source text may also be omitted without affecting the printed form or the logical significance of the paragraph.

Elision of semicolons between declarations and predicates is not supported. Use of semicolons as low precedence conjunctions in the predicate part is supported.

5.2.4.2 Proof Support

The predicate obtained from such a schema declaration is the same as that which would result from the equivalent schema declaration using $\widehat{=}$ and a horizontal schema expression:

```
SML
│val sch_def = z_get_spec ⓩSch⌝;
```

```
ProofPower Output
│val sch_def = ⊢ Sch =
│  [x, y : ℤ; z : ℕ | x = y ∨ y = z] : THM
```

This can be used in rewriting in the normal manner.

```
SML
│rewrite_conv [sch_def]
│ ⓩ∀ x,y:ℤ; z:ℕ • Sch ∨ disjoint ⟨{x},{y},{z}⟩⌝;
```

```
ProofPower Output
│val it = ⊢ (∀ x, y : ℤ; z : ℕ • Sch ∨ disjoint ⟨{x}, {y}, {z}⟩)
│    ⇔ (∀ x, y : ℤ; z : ℕ
│      • ({x, y} ⊆ ℤ ∧ z ∈ ℕ) ∧ (x = y ∨ y = z)
│      ∨ disjoint ⟨{x}, {y}, {z}⟩) : THM
```

Note that where a schema reference as a predicate is expanded into a horizontal schema, the horizontal schema is immediately eliminated to give the constituent predicate (in the proof context *z_language*). Similar observations will also apply to schemas which are defined in terms of schema operations. The operation will normally be eliminated if it appears as a predicate, using the relevant membership conversion (since membership of the relevant theta-term is implicit in the formation of a predicate from a schema expression, i.e. $sexp \Leftrightarrow \theta sexp \in sexp$).

5.2.5 Generic Schema Boxes

5.2.5.1 Syntax

```
Z
┌─DSUBS[X]──────────────────────────────
│      set1, set2: ℙ X
├───────────────────────
│
│      set1 ∩ set2 = {}
└───────────────────────────────────────
```

These differ from ordinary schemas only in having formal generic parameters.

5.2.5.2 Proof Support

The defining predicate is a generic equation with a horizontal schema on the right hand side.

```
SML
│val dsubs_def = z_get_spec ⓩDSUBS⌝;
```

```
ProofPower Output
│val dsubs_def = ⊢ [X](DSUBS[X] =
│  [set1, set2 : ℙ X | set1 ∩ set2 = {}]) : THM
```

This is normally used by rewriting, the actual generic parameters being supplied automatically.

```
SML
│rewrite_conv [dsubs_def]
│ ⓩ∀ DSUBS[ℕ] • set1 ⊆ ℕ ∧ set2 ⊆ ℕ⌝;
```

```
ProofPower Output
│val it = ⊢ (∀ (DSUBS[ℕ]) • set1 ⊆ ℕ ∧ set2 ⊆ ℕ)
│        ⇔ (∀ [set1, set2 : ℙ ℕ | set1 ∩ set2 = {}]
│           • set1 ⊆ ℕ ∧ set2 ⊆ ℕ) : THM
```

Note here that, because the schema reference expanded was in a declaration, the result is not within the language described in [3]. This extension to the language is expected to appear in the forthcoming Z standard.

This works out more naturally in a goal oriented proof as follows:

```
SML
│push_merge_pcs ["'usr011", "z_library"];
│
│set_goal([],ⓩ∀ DSUBS[ℕ] • set1 ⊆ ℕ ∧ set2 ⊆ ℕ⌝);
│a z_strip_tac;
```

```
ProofPower output
│...
│(∗ ?⊢ ∗)  ⓩ(DSUBS[ℕ]) ∧ true ⇒ set1 ⊆ ℕ ∧ set2 ⊆ ℕ⌝
│...
```

The first step of stripping has converted the schema-as-declaration into a schema-as-predicate. Rewriting with the definition is now appropriate.

SML
```
a (rewrite_tac [dsubs_def]);
```

ProofPower output
```
...
(* ?⊢ *)  ⓩ{set1, set2} ⊆ ℙ ℕ ∧ set1 ∩ set2 = {}
                   ⇒ set1 ⊆ ℕ ∧ set2 ⊆ ℕ⌝
...
```

SML
```
a (REPEAT strip_tac);
pop_pc();
```

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

5.2.6 Axiomatic Descriptions

5.2.6.1 Syntax

Entered as:

```
    ⓈZAX
    │       twice _ : ℤ → ℤ
    ├─────────────
    │       ∀i : ℤ • twice i = 2*i
    ■
```

Printed as:

Z
```
│       twice _ : ℤ → ℤ
├──────────────────────────────
│       ∀i : ℤ • twice i = 2*i
```

5.2.6.2 Proof Support

The predicate obtained from *z_get_spec* is a conjunction of which the first part is the predicate implicit in the declaration part of the definition, and the second conjunct is the explicit predicate.

SML
```
val twice_def = z_get_spec ⓩ(twice _)⌝;
```

ProofPower Output

```
val twice_def = ⊢ (twice _) ∈ ℤ → ℤ
        ∧ (∀ i : ℤ • twice i = 2 ∗ i) : THM
```

The predicate can be used for rewriting, provided the condition implicit in the universal quantifier can be proven. In the current proof context this is not achieved:

SML

```
rewrite_conv[twice_def] ⓩtwice 4⌝;
```

ProofPower Output

```
Exception− Fail ∗ no rewriting occurred
  [rewrite_conv.26001] ∗ raised
```

because ℤ is not known to be a given set in this context.

A common way of obtaining a usable rewrite is by forward chaining using *all_fc_tac* or *all_asm_fc_tac* (*fc_tac* and *asm_fc_tac* may also be used, but are likely to give mixed-language results, involving HOL quantifiers).

SML

```
set_goal([],ⓩ∀ n:ℤ • twice n = 2∗n⌝);
a (REPEAT z_strip_tac);
```

ProofPower Output

```
...
(∗  1 ∗)  ⓩn ∈ ℤ⌝

(∗ ?⊢ ∗)  ⓩtwice n = 2 ∗ n⌝
...
```

SML

```
a (all_fc_tac [twice_def]);
```

ProofPower Output

```
Current and main goal achieved
```

5.2.7 Generic Axiomatic Descriptions

5.2.7.1 Syntax

Entered as:

```
    ╒[X]═══════════════════════
    │ length : seq X → ℕ
    ├─────────────────────
    │       length ⟨⟩ = 0;
    │       ∀ h:X; t: seq X•
    │       length (⟨h⟩ ⁀ t) = length t + 1
    └───────────────────────────
```

The double horizontal bar characters are inessential, though the top left hand corner is not.

Printed as:

Z

$[X]$

$length : seq\ X \rightarrow \mathbb{N}$

$$length\ \langle\rangle = 0;$$
$$\forall\ h{:}X;\ t{:}\ seq\ X \bullet$$
$$length\ (\langle h\rangle \frown t) = length\ t + 1$$

The following example shows the use in the declaration part and the predicate part of the schema of a template for which a fixity declaration has been entered.

Use of the template is necessary in the predicate of this generic description because the '...' parameter position in a template requires a parameter which is a list display (with list brackets omitted); no other kind of expression is permitted at this point.

Z

$[X, Y, Z]$

$select\ ...\ from\ \_ : (X \leftrightarrow Y) \times (Y \leftrightarrow Z) \rightarrow (Y \leftrightarrow Z)$

$$\forall\ indexed\_set{:}(X \leftrightarrow Y);\ relation{:}(Y \leftrightarrow Z) \bullet$$
$$(select\ ...\ from\ \_)\ (indexed\_set,\ relation)$$
$$= (ran\ indexed\_set) \lhd relation$$

5.2.7.2 Proof Support

Generic axiomatics definitions give rise to generic predicates whose bodies are the conjunction of the predicate implicit in the declaration part and the explicit predicate in the body of the paragraph.

In the body of the paragraph the normal rules for supply of actual generic parameters where these have been omitted are varied. The formal generic parameters are supplied, rather than the set of all elements of the inferred type. Omission of the actual generic parameters is mandatory in this context.

SML

```
val select_from_def = z_get_spec ⓩ(select ... from _)⌝;
```

ProofPower Output

```
val select_from_def = ⊢ [X,
  Y,
  Z]((select ... from _)[X, Y, Z] ∈ (X ↔ Y) × (Y ↔ Z) → Y ↔ Z
    ∧ (∀ indexed_set : X ↔ Y; relation : Y ↔ Z
      • (select ... from _)[X, Y, Z] (indexed_set, relation)
        = ran indexed_set ◁ relation)) : THM
```

We see in the above that even if the '...' parameter type had not been used the verbatim template would have appeared in the generic predicate, since this is the only way to supply actual generic parameters to a template.

Other than in the defining occurrences the template is applied in the intended fashion, e.g.:

SML
```
        ⓩselect 1,3,8 from sequence⌝;
```

5.2.8 Structured Set Definitions

"Structured Set Definition" is the name in version 1.0 of the Z standard[10] for what has hitherto been called a "Free Type" paragraph.

5.2.8.1 Syntax

Entered as:
```
ⓈZ
TREE ::= tip | fork (ℕ × TREE × TREE)
■
```

Printed as:

Z
```
TREE ::= tip | fork (ℕ × TREE × TREE)
```

Note that ProofPower at present neither requires nor allows the normal chevrons in these definitions. Where a fixity clause is in force it is applied normally in the context of a structured set definition.

5.2.8.2 Proof Support

A structured set definition gives rise to defining axioms as described in the ZRM [3].

For each given set an axiom defines it as the set of all elements of a new type:

SML
```
val tree_def = z_get_spec ⓩTREE⌝;
```

ProofPower Output
```
val tree_def = ⊢ TREE = 𝕌 : THM
```

A single axiom characterises all the constructors of the 'structured set'.

SML
```
val tip_def = z_get_spec ⓩtip⌝;
```

ProofPower Output
```
val tip_def = ⊢ (tip ∈ TREE
 ∧ fork ∈ ℕ × TREE × TREE ↣ TREE)
 ∧ disjoint ⟨{tip}, ran fork⟩
 ∧ (∀ W : ℙ TREE | {tip} ∪ fork ⦇ ℕ × W × W ⦈ ⊆ W•
        TREE ⊆ W) : THM
```

5.2.9 Mutually Recursive Structured Set Definitions

Mutually recursive structured set definitions are also supported by ProofPower.

5.2.9.1 Syntax

Entered as:

```
ⓈZ
TYPE ::= Tvar G1 | Tcon (G1 × seq TERM)
&
TERM ::= Con (G1 × TYPE) | App (TERM × TERM)
■
```

Printed as:

```
Z
│TYPE ::= Tvar G1 | Tcon (G1 × seq TERM)
│&
│TERM ::= Con (G1 × TYPE) | App (TERM × TERM)
```

5.2.9.2 Proof Support

The axiomatisation of these definitions is a generalisation of the simpler cases:

```
SML
│val tvar_def = z_get_spec ⓩTvar⌝;
```

```
ProofPower Output
│val tvar_def = ⊢ (Tvar ∈ G1 ↣ TYPE
│    ∧ Tcon ∈ G1 × (seq TERM) ↣ TYPE
│    ∧ Con ∈ G1 × TYPE ↣ TERM
│    ∧ App ∈ TERM × TERM ↣ TERM)
│    ∧ (disjoint ⟨ran Tvar, ran Tcon⟩
│    ∧ (∀ W : ℙ TYPE
│      | Tvar ⦇ G1 ⦈ ∪ Tcon ⦇ G1 × (seq TERM) ⦈ ⊆ W
│      • TYPE ⊆ W))
│    ∧ disjoint ⟨ran Con, ran App⟩
│    ∧ (∀ W : ℙ TERM
│      | Con ⦇ G1 × TYPE ⦈ ∪ App ⦇ W × W ⦈ ⊆ W
│      • TERM ⊆ W) : THM
```

5.2.10 Constraints

Constraints are arbitrary axioms introduced by the user, which are permitted in ProofPower to be generic.

5.2.10.1 Syntax

A constraint is entered as follows:

```
│    ⓈZ
│     {1} swap {⟨1⟩} = {⟨1⟩} × {1}
│           ∧ Sch ≠ [x, y, z : ℤ]
│    ■
```

which is printed as:

```
Z
│ {1} swap {⟨1⟩} = {⟨1⟩} × {1}
│       ∧ Sch ≠ [x, y, z : ℤ]
```

An example of a *generic* constraint is:

```
Z
│[X] ((∃f : X ↣ G1 • true) ⇔ (∃f : X ↣ G2 • true))
```

5.2.10.2 Proof Support

The axioms resulting from the entry of constraints are stored in the current theory under keys which are of the form "Constraint n" where n is a numeric literal.

They may not be retrieved using *z_get_spec* since they are not associated with any specific global variable.

A constraint is therefore accessed using *get_axiom* with the relevant key.

```
SML
│val c1 = get_axiom "−" "Constraint 2";
```

```
ProofPower output
│val c1 = ⊢ [X]((∃ f : X ↣ G1 • true) ⇔
│              (∃ f : X ↣ G2 • true)) : THM
```

5.2.11 Conjectures

Conjectures are arbitrary predicates mentioned by the user, normally to suggest or claim that they are true.

If entered into the source document as a conjecture paragraph then the conjecture will be syntax and type checked by the system whenever the document is processed by ProofPower.

Such conjectures are permitted to be generic.

5.2.11.1 Syntax

A conjecture paragraph is entered as follows:

```
│    ⓈZ
│     ?⊢ 0=1
│    ■
```

which is printed as:

```
z
│ ?⊢ 0=1
```

This conjecture can be proven false in ProofPower. Its assertion as a constraint would render the relevant theory inconsistent, but its inclusion as a conjecture is harmless.

An example of a *generic* conjecture is:

```
z
│ ?⊢ [X,Y] (∀s:ℙ(X × Y)•(∀x:X• ∃y:Y• (x,y) ∈ s)
│                ⇒ (∃f : X → Y • true))
```

This conjecture should in fact be provable under ProofPower. Asserting it as a constraint would avoid the need to prove it but would not render the theory inconsistent. Including it as a conjecture however leaves the result still in need of proof should the result be required.

5.2.11.2 Proof Support

There is no proof support for conjecture paragraphs. The ProofPower system provides support for syntax and type checking conjectures only. Once checked the contents of a conjecture paragraph are discarded.

If it required to prove a conjecture then it should be entered into the subgoal package using *set_goal* or *push_goal*.

5.3 Proof Support for Paragraphs

5.3.1 Consistency Proofs for Axiomatic Descriptions

Specifications are treated as extension to the logical system supplied by ProofPower. Such extensions take the form of introducing new type constructors (in Z these are always 0-ary type constructors corresponding to new given sets), new constants (called global variables in Z) and new axioms which generally provide information about these new types and constants.

When a new constant is introduced together with an axiom describing that constant, the extension will often be *conservative*. To say that such an extension is conservative is to say that the information in the axiom only serves to define the constant which is introduced, and does not enable any new facts to be proved which do not mention that constant.

Conservative extensions are important since an extension known to be conservative cannot render the logical system inconsistent, whereas an arbitrary logical extension may render the system inconsistent, thus enabling 'false' conjectures to be proven.

Because of the special importance of extensions which are conservative, axioms which are introduced as a part of a conservative extension are known by the system as *definitions* and are kept distinct from axioms which are not known to be conservative. The system provides mechanisms for undertaking conservative extensions in ways which it can check, so that specifications introduced using only these means can be guaranteed by the system not to interfere with the consistency of the logical system.

If Z axiomatic descriptions are entered into ProofPower while flag *z_use_axioms* is false the descriptions will be stored as *definitions* rather than as axioms. In order to do this the system has to establish that the resulting axiom is a conservative extension of the previous logical context. This is done either by the system automatically constructing a proof that this is the case for the axiom as supplied (which the supplied proof contexts are not capable of doing), or by the system adding a consistency caveat to the axiom before storing it as a definition. The proof of consistency of the axiom with caveat is trivial, ensuring that the extension is conservative, but the user will subsequently need to prove the consistency result before the axiom can be used in proof without caveat.

We demonstrate this with a very simple axiomatic description. First ensure that we are in the right mode:

SML
```
set_flag("z_use_axioms", false);
set_pc "z_library";
```

Then enter the axiomatic specification:

Z
```
│      root : ℤ
├──────────────────────────────
│      root ∗ root = 9
```

Now retrieve the specification:

SML
```
z_get_spec ⓩroot⌝;
```

ProofPower output
```
...
 ⊢ root ∈ ℤ ∧ root ∗ root = 9 : THM
```

The indigestible assumption (not shown here) is a consistency caveat. Until the caveat is proven the specification is unusable for proof.

To prove the caveat the goal should be set up by *z_push_consistency_goal*.

SML
```
z_push_consistency_goal ⓩroot⌝;
```

ProofPower output
```
...
(∗ ?⊢ ∗)  ⓩ∃ root′ : ℤ • root′ ∗ root′ = 9⌝
...
```

This is easily discharged by supplying a witness as follows:

SML
```
a (z_∃_tac ⓩ3⌝ THEN rewrite_tac[]);
```

ProofPower output
```
Tactic produced 0 subgoals:
Current and main goal achieved
...
```

Then the result must be saved using *save_consistency_thm* as follows:

SML
```
save_consistency_thm ⓩroot⌝ (pop_thm());
```

We can now retrieve a useful theorem using *z_get_spec*.

SML
```
val root_def = z_get_spec ⓩroot⌝;
```

ProofPower output
```
val root_def = ⊢ root ∈ ℤ ∧ root ∗ root = 9 : THM
```

5.3.2 Consistency Proofs for Generic Axiomatic Descriptions

A similar pattern is necessary when introducing generics, though some additional complications arise. Though ProofPower requires no uniqueness condition to establish a generic global variable, the consistency caveat arising cannot be expressed in Z, since it involves generic local variables, which are not permitted.

The consistency caveat therefore appears in a mixture of Z and HOL and some mixed language working is necessary.

Z
```
╒═[X]═══════════════════════
│ empty : ℙ X
├──────────────
│ empty = {}
└───────────────────────────
```

SML
```
z_push_consistency_goal ⓩempty⌝;
```

ProofPower output
```
...
(∗ ?⊢ ∗)  ⓩ∃ empty′ : 𝕌
            • ∀ X : 𝕌 • ⌜empty′ ⓩ(X)⌝⌝ ∈ ℙ X ∧ ⌜empty′ ⓩ(X)⌝⌝ = {}⌝
...
```

Here the outermost existential in fact quantifies over generic local variable (though this is obscured by the use of 𝕌) and the uses of the generic local variable in the predicate are displayed by switching into HOL.

The witness supplied must be a HOL function which takes a tuple of formal parameters and yields the required instance:

SML
```
a (z_∃_tac ⌜λ X • ⓩ{}⌝⌝ THEN PC_T1 "hol" rewrite_tac[]);
```

ProofPower output
```
...
(* ?⊢ *)  ⓩ⌜λ X • ⓩ{}⌝⌝ ∈ 𝕌 ∧ (∀ X : 𝕌 • {} ∈ ℙ X)⌝
...
```

Here the rewriting was done in a HOL proof context because a HOL beta reduction is needed. Now we rewrite again in the Z proof context:

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

ProofPower output
```
Tactic produced 0 subgoals:
Current and main goal achieved
...
```

SML
```
save_consistency_thm ⓩempty⌝ (pop_thm());
val empty_def = z_get_spec ⓩempty⌝;
```

ProofPower output
```
val empty_def = ⊢ [X](empty[X] ∈ ℙ X ∧ empty[X] = {}) : THM
```

Z
```
╒═[X, Y]══════════════════════════════════
│      cprod : ℙ (X × Y)
├──────────────────────
│      ∀x:ℙX; y:ℙY• cprod = X × Y
└─────────────────────────────────────────
```

SML
```
z_push_consistency_goal ⓩcprod⌝;
```

ProofPower output
```
(* ?⊢ *)  ⓩ∃ cprod′ : 𝕌
              • ∀ X : 𝕌; Y : 𝕌
                • ⌜cprod′ ⓩ(X, Y)⌝⌝ ∈ ℙ (X × Y)
                  ∧ (∀ x : ℙ X; y : ℙ Y • ⌜cprod′ ⓩ(X, Y)⌝⌝ = X × Y)⌝
```

Note here that the parameter to the required HOL function is a Z 2-tuple. In the general case it is an *n − tuple*, where *n* is the number of generic parameters, even when *n = 1* (very limited support for 1-tuples is available). When used the appropriate projection function must therefore be applied. A type cast is likely to be necessary for the type inferrer, as below.

SML

```
a (z_∃_tac ⌜λ xy• Ⓩ(xy⊕(𝕌×𝕌)).1 × xy.2⌝⌝);
a (PC_T1 "hol" rewrite_tac[]);
a (rewrite_tac[]);
```

ProofPower output

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

Later releases of ProofPower may be expected to discharge such consistency results automatically, and proficient users may themselves adapt the available consistency provers for HOL specifications to give a basic capability for Z.

5.4 Theories

Z specifications are held in theories in the same way as specifications in HOL. The content of these theories is the same as the HOL theories in respect of matters of a logical nature, but there are differences in the information held in the theories relating to the concrete presentation of a specification.

In HOL the fixity information held concerns which names are pre-fix, post-fix and infix, and which names are binders. HOL also allows aliases and type abbreviations.

None of this information is appropriate for Z specifications. ProofPower-Z provides a generalised mixfix notation, and the information controlling this is provided in the form of fixity paragraphs. This information is retained in theories whose language is Z. Aliases and type abbreviations are not supported for Z.

The primary facilities for access to Z theories are *z_print_theory* and *z_get_spec*. Other facilities available for HOL theories may also be used on Z theories, e.g. *get_parents*, *get_ancestors*, *get_thm*.

THE Z TOOLKIT IN ProofPower

This chapter introduces the theories forming the Z ToolKit.

The focus is on reasoning within the theories using the proof contexts supplied for the theories.

The Z ToolKit is split aross six theories in the ProofPower database, one for each of the main sections in Annex C to the draft Z standard [10].

Each of these theories is fully populated with appropriate definitions, in the form in which they appear in the standard. In the development of the theories we have not felt it appropriate to populate the theories with collections of theorems such as those given as rules in the ZRM [4]. In general we have found that the results required in applications are rarely found in these collections, and that these rules are not the most satisfactory basis from which to derive the required results. The rules cited in the ZRM are nevertheless (when true) useful tests of whether our proof system is able to conveniently prove results in these theories, and similar results have been used extensively for examples and exercises.

The approach adopted has generally been to prove those results which are most helpful in providing an automatic proof capability in the relevant theory and to incorporate these results into proof contexts which facilitate their automatic application. This may involve quite different techniques in different theories.

In the following descriptions we identify for each theory suitable proof contexts for proving results in that theory, and give a description of the proof methods which are effective in the theory (where these have been established).

A selection proof contexts are listed in the following table:

z_predicates
z_language
z_language_ext
z_sets_alg
z_sets_ext
z_rel_ext
z_fun_ext
z_library
z_library_ext

For general purpose use *z_library* is recommended. Retreat to a context earlier in the list if your current proof context is doing too much. To do proofs using extensionality results use one of the *_ext* contexts.

6.1 Sets

The theory *z_sets* contains those parts of the Z ToolKit corresponding to section C.1 of the draft Z standard [10].

A suitable proof context for establishing results in this theory is *z_sets_ext*.

This proof context attempts proofs of conjectures using the principles of extensionality of sets. These enable relations over sets to be expressed in terms of logical relations between the membership conditions for the sets. These are combined with principles which account for each of the defined operations defined in this theory for constructing sets expressed as elimination rules for claims about membership of the constructed sets. The resulting collections of principles suffice to reduce statements in set theory to statements in the predicate calculus with equality, which can be proven by the established methods.

6.2 Relations

The theory *z_relations* contains those parts of the Z ToolKit corresponding to section C.2 of the draft Z standard [10].

This theory is well supported, the principles of reasoning being straightforward.

This theory adds operators over sets which are specific to sets of ordered pairs, such as *dom* and *ran*. The main principles for reasoning automatically in this theory remain the same, viz: rewriting with extensionality results to reduce the problem to the predicate calculus. The main source of extra complexity, in addition to the extra operators which must be eliminated, is the fact that definitions of these operations often involve ordered pairs, and the resulting predicate calculus results require equational reasoning to complete the proof. We are more often faced with reduction only to predicate calculus with equality, which our resolution facilities do not support.

6.3 Functions

The theory *z_functions* contains those parts of the Z ToolKit corresponding to section C.3 of the draft Z standard [10].

Extensional reasoning suffices for automatic proof of many of the results required in this theory. However, much reasoning in the system, particulary that concerning 'set inference' (demonstrating that values lie in the domain of the functions applied to them, or within the range of applicability of some rewrite rule), is better done using rules about functions rather than eliminating the vocabulary in favour of set theory. No specific support is as yet available for this latter sort of reasoning about functions, though general purpose facilities (particularly forward and backward chaining) are available which provide a measure of automation.

6.4 Numbers and Finiteness

The theory *z_numbers* contains those parts of the Z ToolKit corresponding to section C.4 of the draft Z standard [10].

The basic theory of integers is reasonably well developed. The kind of results needed to do more or less manual proofs in arithmetic are available. Induction tactics of various kinds are available. The proof context *z_lin_arith* provides a proof procedure for linear arithmetic. The proof context for this theory undertakes evaluation of expressions of numeric literals automatically during rewriting or stripping.

The theory of finiteness is not yet properly developed, though all the definitions are present.

6.5 Sequences

The theory *z_sequences* contains those parts of the Z ToolKit corresponding to section C.5 of the draft Z standard [10].

By comparison with the theory of lists in HOL this theory is much more difficult to reason about. This is because sequences are defined in terms of finite functions over natural numbers, by contrast with lists, which are a structured or free type for which induction results come more easily.

In addition to difficulties arising from the representation of sequeces in Z, further difficulties arise from the form of the definitions of the various operators. Since these do not follow and general inductive pattern reasoning about them will not be straigtforwardly supported by induction principles.

Though the theory of sequences is relatively underdeveloped, the definitions are all present, and specifications may therefore be written using them. The proof facilities are adequate for the development of these theories and the provision of a reasonable degree of automation for them. However, the required development is non-trivial and has not yet been completed.

6.6 Bags

The theory *z_bags* contains those parts of the Z ToolKit corresponding to section C.6 of the draft Z standard [10].

This theory is also present but not well developed.

EXERCISES

These exercises are presented in sections which correspond to chapters in the tutorial as follows:

1. The Z Predicate Calculus
   - Forward Propositional Proofs
   - Goal Oriented Propositional Proofs
   - Forward Predicate Calculus Proofs
   - Goal Oriented Predicate Calculus Proofs
   - Rewriting
2. Expressions and Schema Expressions
3. Z Paragraphs and Theories
4. The Z ToolKit
   (a) Sets
   (b) Relations
   (c) Functions
   (d) Numbers and Finiteness
   (e) Sequences
   (f) Bags

In order to save labour at the keyboard, the student may do the exercises by working with a specially prepared database called 'example_zed'. This database should have been built during the installation of ProofPower, if you have any difficulty in obtaining access to the database consult your systems administrator. The database is created by loading this tutorial document into ProofPower with the exception only of the material in Chapter 8. The formal material used to set up the 'exercises' database (and automatically stripped from the source of this document for that purpose) is documented by being marked with a sidebar, thus:

```
SML
│
```

for material in Standard ML. Z paragraphs are also included in the material processed, except where they are specially presented to show the format of the source for such paragraphs.

A sidebar like this:

```
Student
│
```

marks material which the student is expected to enter, either through the keyboard or by cut-and-paste.

The prepared database contains 4 theories, each with some predefined material: in particular, lists of goals. A goal can be conveniently selected by name and set up to be worked on by the subgoal package, by using the function predefined by:

```
SML
fun setlg name goallist = set_goal([],lassoc3 goallist name);
```

For example, to work on the goal which is the first in the list *PM2* (see 7.1.2 below), noting that this goal is named '*2.02', execute:

```
Student
        setlg "*2.02" PM2;
```

Another useful function attempts to prove a conjecture using *prove_rule* and store the results in the current theory. It is predefined by:

```
SML
fun prove_and_store (key, term) = save_thm (key, prove_rule[] term);
```

It is important that each exercise be attempted in the correct context. For this reason, at various points in the sequence of exercises, instructions are given to set up the appropriate context for the group of exercises which follow, up to the next setting of context. Setting up the context covers clearing the goal-stack, opening the appropriate 'exercise' theory, setting the proof-context and flags.

The following lines of ML are for preparing the database:

```
SML
repeat drop_main_goal;
open_theory "z_library";
set_pc "z_library";
set_flags [("z_type_check_only", false), ("z_use_axioms", true)];
```

7.1 Predicate Calculus Exercises

The following line of ML is for preparing the database by setting up the first exercise theory:

```
SML
new_theory "z_exercises_1";
```

7.1.1 Forward Propositional Proofs

Set the context by:

```
Student
repeat drop_main_goal;
open_theory "z_exercises_1";
set_pc "z_library";
```

and then, using the methods described in 3.1.1 prove a selection of the following results by use of *⇒_elim*, *asm_rule* and *⇒_intro*:

(a) $b \Rightarrow c,\ a \Rightarrow b,\ a \vdash c$

(b) $a \Rightarrow b \Rightarrow c,\ a,\ b \vdash c$

(c) $a \Rightarrow b \Rightarrow c,\ b \vdash a \Rightarrow c$

(d) $\vdash (a \Rightarrow b \Rightarrow c) \Rightarrow b \Rightarrow (a \Rightarrow c)$

7.1.2 Goal Oriented Propositional Proofs

- Choose examples from *PM2* below to set up as a goal. E.g. to choose as a goal ⓩ$q \Rightarrow (p \Rightarrow q)$⌝, execute *setlg* "*∗2.02*" *PM2*;. Work the example with
  1. *a z_strip_tac*;
  2. and/or: *a step_strip_tac*;
- Observe the steps taken and try to identify the reasons for discharge of subgoals.
- Select the weakest "proof context":

  Student
  ```
  push_pc "initial";
  ```

  then retry the examples (or previous exercises).
- When you have finished restore the original proof context by:

  Student
  ```
  pop_pc();
  ```

The following lines of ML are for preparing the 'exercises' database. They are taken from Principia Mathematica *2, and are shown together with their reference numbers.

SML
```
val PM2 =[
("*2.02", ⓩq ⇒ ( p ⇒ q)⌝),
("*2.03", ⓩ(p ⇒ ¬ q) ⇒ (q ⇒ ¬ p)⌝),
("*2.15", ⓩ(¬ p ⇒ q) ⇒ (¬ q ⇒ p)⌝),
("*2.16", ⓩ(p ⇒ q) ⇒ (¬ q ⇒ ¬ p)⌝),
("*2.17", ⓩ(¬ q ⇒ ¬ p) ⇒ (p ⇒ q)⌝),
("*2.04", ⓩ(p ⇒ q ⇒ r) ⇒ (q ⇒ p ⇒ r)⌝),
("*2.05", ⓩ(q ⇒ r) ⇒ (p ⇒ q) ⇒ (p ⇒ r)⌝),
("*2.06", ⓩ(p ⇒ q) ⇒ (q ⇒ r) ⇒ (p ⇒ r)⌝),
("*2.08", ⓩp ⇒ p⌝),
("*2.21", ⓩ¬ p ⇒ (p ⇒ q)⌝)];
```

The following are from Principia Mathematica *3.

SML

```
val PM3 =[
("*3.01", ⌜p ∧ q ⇔ ¬(¬ p ∨ ¬ q)⌝),
("*3.2", ⌜p ⇒ q ⇒ p ∧ q⌝),
("*3.26", ⌜p ∧ q ⇒ p⌝),
("*3.27", ⌜p ∧ q ⇒ q⌝),
("*3.3", ⌜(p ∧ q ⇒ r) ⇒ (p ⇒ q ⇒ r)⌝),
("*3.31", ⌜(p ⇒ q ⇒ r) ⇒ (p ∧ q ⇒ r)⌝),
("*3.35", ⌜(p ∧ (p ⇒ q)) ⇒ q⌝),
("*3.43", ⌜(p ⇒ q) ∧ (p ⇒ r) ⇒ (p ⇒ q ∧ r)⌝),
("*3.45", ⌜(p ⇒ q) ⇒ (p ∧ r ⇒ q ∧ r)⌝),
("*3.47", ⌜(p ⇒ r) ∧ (q ⇒ s) ⇒ (p ∧ q ⇒ r ∧ s)⌝)];
```

Results from Principia Mathematica *4

SML

```
val PM4 =[
("*4.1", ⌜p ⇒ q ⇔ ¬ q ⇒ ¬ p⌝),
("*4.11", ⌜(p ⇔ q) ⇔ (¬ p ⇔ ¬ q)⌝),
("*4.13", ⌜p ⇔ ¬ ¬ p⌝),
("*4.2", ⌜p ⇔ p⌝),
("*4.21", ⌜(p ⇔ q) ⇔ (q ⇔ p)⌝),
("*4.22", ⌜(p ⇔ q) ∧ (q ⇔ r) ⇒ (p ⇔ r)⌝),
("*4.24", ⌜p ⇔ p ∧ p⌝),
("*4.25", ⌜p ⇔ p ∨ p⌝),
("*4.3", ⌜p ∧ q ⇔ q ∧ p⌝),
("*4.31", ⌜p ∨ q ⇔ q ∨ p⌝),
("*4.33", ⌜(p ∧ q) ∧ r ⇔ p ∧ (q ∧ r)⌝),
("*4.4", ⌜p ∧ (q ∨ r) ⇔ (p ∧ q) ∨ (p ∧ r)⌝),
("*4.41", ⌜p ∨ (q ∧ r) ⇔ (p ∨ q) ∧ (p ∨ r)⌝),
("*4.71", ⌜(p ⇒ q) ⇔ (p ⇔ (p ∧ q))⌝),
("*4.73", ⌜q ⇒ (p ⇔ (p ∧ q))⌝)];
```

Results from Principia Mathematica *5

SML

```
val PM5 =[
("*5.1", ⌜p ∧ q ⇒ (p ⇔ q)⌝),
("*5.32", ⌜(p ⇒ (q ⇔ r)) ⇒ ((p ∧ q) ⇔ (p ∧ r))⌝),
("*5.6", ⌜(p ∧ ¬ q ⇒ r) ⇒ (p ⇒ (q ∨ r))⌝)];
```

7.1.3 Forward Predicate Calculus Proofs

The following exercises concern proof in the predicate calculus in Z. Set the context by:

Student

```
repeat drop_main_goal;
open_theory "z_exercises_1";
set_pc "z_library";
```

and then

1. Using *z_∀_elim* with *z_ℕ_¬_plus1_thm* prove:

   (a) $0 \in \mathbb{N} \land true \Rightarrow \neg\ 0 + 1 = 0$

   (b) $x * x \in \mathbb{N} \land true \Rightarrow \neg\ x * x + 1 = 0$

2. Using *prove_rule* with *z_≤_trans_thm* prove:

   $i \leq j \land j \leq k \Rightarrow i \leq k$

3. Using *prove_rule* and *z_ℕ_¬_plus1_thm* and *z_0_ℕ_thm* prove:

   (a) $\neg\ 0 + 1 = 0$

   (b) $x * x \in \mathbb{N} \Rightarrow \neg\ x * x + 1 = 0$

4. Using *prove_rule* prove:

   (a) $\neg\ 0 < 1 \Leftrightarrow 1 \leq 0$ (using *z_¬_less_thm*), and

   (b) $\forall\ n{:}\mathbb{Z} \bullet 3 \leq x * x \land x * x \leq n \Rightarrow 3 \leq n$ (using *z_≤_trans_thm*)

5. Using *prove_rule* with *≤_clauses* prove:

   $\forall\ i,\ m,\ n{:}\ \mathbb{Z} \bullet i + m \leq i + n \Leftrightarrow m \leq n$

7.1.4 Goal Oriented Predicate Calculus Proofs

The methods of proof, described in Chapter 3, to be illustrated in these exercises are:

1. Proof by stripping.
2. Automatic proof.
3. Proof by the "two tactic method".
4. Proof using forward chaining.

Use the following bits and pieces to try a variety of proofs of the following conjectures (PM9 to PM11b) in the predicate calculus in Z,

```
a contr_tac;
a z_strip_tac;
a strip_tac;
a step_strip_tac;
a (prove_tac[]);
a (asm_prove_tac[]);
a (z_spec_asm_tac ⓩ ⌝ ⓩ ⌝);
a (z_spec_nth_asm_tac 1 ⓩ ⌝);
a (all_asm_fc_tac[]);
```

The following are essentially the same results, taken from Principia Mathematica, as were previously used for exercises in HOL. However, the results are set generic rather than polymorphic, and quantifiers range over sets rather than types. The required proofs are similar to those in HOL, but slightly complicated by the set relativisation of the quantification.

The following are from Principia Mathematica *9.

SML
```
val PM9 =[
("*9.01", ⓩ[X](¬ (∀x:X• φx)) ⇔ (∃x: X• ¬ φx)⌝),
("*9.02", ⓩ[X](¬ (∃x:X• φx) ⇔ (∀x:X• ¬ φx))⌝),
("*9.03", ⓩ[X](∀x:X• φx ∨ p) ⇔ (∀x:X• φx) ∨ p⌝),
("*9.04", ⓩ[X]p ∨ (∀x:X• φx) ⇔ (∀x:X• p ∨ φx)⌝),
("*9.05", ⓩ[X](∃ x:X•true) ⇒ ((∃x:X• φx ∨ p) ⇔ (∃x:X• φx) ∨ p)⌝),
("*9.06", ⓩ[X]p ∨ (∃x:X• φx) ⇔ p ∨ (∃x:X• φx)⌝)];
```

The following are from Principia Mathematica *10.

SML
```
val PM10 =[
("*10.01", ⓩ[X](∃x:X• φx) ⇔ ¬ (∀y:X• ¬ φy)⌝),
("*10.1", ⓩ(∀x:𝕌• φx) ⇒ φy⌝),
("*10.21", ⓩ[X](∀x:X• p ⇒ φx) ⇔ p ⇒ (∀y:X• φy)⌝),
("*10.22", ⓩ[X](∀x:X• φx ∧ ψx) ⇔ (∀y:X• φy) ∧ (∀z:X• ψz)⌝),
("*10.24", ⓩ[X](∀x:X• φx ⇒ p) ⇔ (∃y:X• φy) ⇒ p⌝),
("*10.27", ⓩ[X](∀x:X• φx ⇒ ψx) ⇒ ((∀y:X• φy) ⇒ (∀z:X• ψz))⌝),
("*10.28", ⓩ[X](∀x:X• φx ⇒ ψx) ⇒ ((∃y:X• φy) ⇒ (∃z:X• ψz))⌝),
("*10.35", ⓩ[X](∃x:X• p ∧ φx) ⇔ p ∧ (∃y:X• φy)⌝),
("*10.42", ⓩ[X](∃x:X• φx) ∨ (∃y:X• ψy) ⇔ (∃z:X• φz ∨ ψz)⌝),
("*10.5", ⓩ[X](∃x:X• φx ∧ ψx) ⇒ (∃y:X• φy) ∧ (∃z:X• ψz)⌝),
("*10.51", ⓩ[X] (¬ (∃x:X• φx ∧ ψx) ⇒ (∀y:X• φy ⇒ ¬ ψy))⌝)];
```

SML
```
val PM10b =[
("*10.271", ⓩ[X](∀x:X• φx ⇔ ψx) ⇒ ((∀y:X• φy) ⇔ (∀z:X• ψz))⌝),
("*10.281", ⓩ[X](∀x:X• φx ⇔ ψx) ⇒ ((∃y:X• φy) ⇔ (∃z:X• ψz))⌝)];

```

The following are from Principia Mathematica *11.

SML
```
val PM11 =[
("*11.1", ⓩ(∀x, y:𝕌• φ(x,y)) ⇒ φ(z,w)⌝),
("*11.2", ⓩ[X](∀x, y:X• φ(x,y)) ⇔ (∀y, x:X• φ(x,y))⌝),
("*11.3", ⓩ[Y](p ⇒ (∀x, y:Y• φ(x,y)))
                ⇔ (∀x, y:Y• p ⇒ φ(x,y))⌝),
("*11.35", ⓩ[Y](∀x, y:Y• φ(x,y) ⇒ p) ⇔ (∃x, y:Y• φ(x,y)) ⇒ p⌝)
];
```

SML

```
val PM11b =[
("*11.32", ⓩ[Y](∀x, y:Y• φ(x,y) ⇒ ψ(x,y))
                ⇒ (∀x, y:Y• φ(x,y)) ⇒ (∀x, y:Y• ψ(x,y))⌝),
("*11.45", ⓩ[Y](∃x, y:Y• true) ⇒ ((∃x, y:Y• p ⇒ φ(x,y))
                ⇔ (p ⇒ (∃x, y:Y• φ(x,y))))⌝),
("*11.54", ⓩ[Y](∃x, y:Y• φx ∧ ψy) ⇔ (∃x:Y• φx) ∧ (∃y:Y• ψy)⌝),
("*11.55", ⓩ[Y](∃x, y:Y• φx ∧ ψ(x,y))
                ⇔ (∃x:Y• φx ∧ (∃y:Y• ψ(x,y)))⌝),
("*11.6", ⓩ[X](∃x:X• (∃y:Y• φ(x,y) ∧ ψy) ∧ χx)
                ⇔ (∃y:Y• (∃x:X• φ(x,y) ∧ χx) ∧ ψy)⌝),
("*11.62", ⓩ(∀x:X; y:Y• φx ∧ ψ(x,y) ⇒ χ(x,y))
                ⇔ (∀x:X• φx ⇒ (∀y:Y• ψ(x,y) ⇒ χ(x,y)))⌝)
];
```

7.1.5 Rewriting

7.1.5.1 Rewriting with the Subgoal Package

Set the context with:

Student

```
repeat drop_main_goal;
open_theory "z_exercises_1";
set_pc "z_library_ext";
```

and then

1. choose a goal from set theory, e.g.:

   Student

   ```
   set_goal([],ⓩ a \ (b ∩ c) = (a \ b) ∪ (a \ c)⌝);
   ```

2. rewrite the goal using the current proof context:

   Student

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

3. step back using undo:

   Student

   ```
           undo 1;
   ```

4. now try rewriting without using the proof context:

   Student

   ```
           a (pure_rewrite_tac[]);
   ```

   (this should fail)

5. try rewriting one layer at a time:

Student
```
        a (once_rewrite_tac[]);
```

repeat until it fails.

6. now try rewriting with specific theorems:

Student
```
        set_goal([],ⓩ a \ (b ∩ c) = (a \ b) ∪ (a \ c)⌝);
        a (pure_rewrite_tac[z_sets_ext_clauses]);
        a (pure_rewrite_tac[z_set_dif_thm]);
        a (pure_rewrite_tac[z_∩_thm, z_∪_thm]);
        a (pure_rewrite_tac[z_set_dif_thm]);
```

7. finish the proof by stripping:

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

8. extract the theorem

Student
```
        top_thm();
```

9. repeat the above then try repeating:

Student
```
        pop_thm();
```

7.1.5.2 Combining Forward and Backward Proof

The following exercise illustrates how forward inference may be helpful in specialising results for use in rewriting. Some hints are given about the method. For each example try the methods suggested for the previous example to see how they fail before following the hint.

Set the context with

Student
```
repeat drop_main_goal;
open_theory "z_exercises_1";
set_pc "z_library";
```

and then prove the following results by rewriting using the goal package.

1. :

Student
```
set_goal([],ⓩ x + y = y + x⌝);
```

2. :

Student
```
set_goal([],⌜Z x + y + z = (x + y) + z⌝);
(* hint : try using z_plus_assoc_thm *)
```

3. :

Student
```
set_goal([],⌜Z z + y + x = y + z + x⌝);
(* hint : try using z_plus_assoc_thm1 *)
```

4. :

Student
```
set_goal([],⌜Z x + y + z = y + z + x⌝);
(* hint : try using z_∀_elim with z_plus_assoc_thm1 *)
```

5. :

Student
```
set_goal([],⌜Z x + y + z + v = y + v + z + x⌝);
(* hint : try using z_∀_elim with z_plus_order_thm *)
```

7.1.6 Stripping

- Use the examples from Principia Mathematica and also ZRM, e.g.:

Student
```
        set_goal([],⌜Z p ∧ q ⇒ (p ⇔ q)⌝);
```

with

1. :

Student
```
        a z_strip_tac;
```

2. and/or:

Student
```
        a step_strip_tac;
```

- Observe the steps taken and try to identify the reasons for discharge of subgoals.
- Select the weakest “proof context”:

Student
```
        push_pc "initial";
```

then retry the examples (or previous exercises).

- When you have finished restore the original proof context by:

Student
```
        pop_pc();
```

7.2 Expressions and Schema Expressions

The following lines of ML are for preparing the exercise database:

SML
```
open_theory "z_library";
new_theory "z_exercises_2";
new_parent "usr011";
```

7.2.1 Expressions

The following examples are provable by *prove_tac* or *prove_rule* in proof context *z_library*, so first set the context by:

Student
```
repeat drop_main_goal;
open_theory "z_exercises_2";
set_pc "z_library";
```

SML
```
val ZE1 = [
("ZE1.1", ⓩ(2,4) ∈ (λx:ℕ • 2*x)⌝),
("ZE1.2", ⓩ{1,2,3} ∈ ℙ {1,2,3,4}⌝),
("ZE1.3", ⓩℕ ∈ ℙ ℤ⌝),
("ZE1.4", ⓩ"a" ∈ {"a", "b"}⌝),
("ZE1.5", ⓩ¬ 2 ∈ {3,4}⌝),
("ZE1.6", ⓩx ∈ {1,2} ⇒ x ∈ {1,2,3}⌝),
("ZE1.7", ⓩx*x ∈ {y:ℤ | ∃z:ℤ • y = z*z}⌝),
("ZE1.8", ⓩ(x,y,z) = (v,w,x) ⇒ (y,z) = (w,v)⌝),
("ZE1.9", ⓩ(x ≙ a, y ≙ b) = (x ≙ v, y ≙ w) ⇒ (v ≙ a, w ≙ w) = (w ≙ b, v ≙ v)⌝),
("ZE1.10", ⓩ∀File;File′• θFile = θFile′ ⇒ age = age′⌝),
("ZE1.11", ⓩ∀File• (θFile′).age = age′⌝),
("ZE1.12", ⓩ∀File;File′•(θFile).age = age′ ∧ (θFile).people = people′
                    ⇒ θFile = θFile′⌝)];
```

The next examples are provable by *prove_tac* or *prove_rule* in proof context *z_language_ext*, so set the context by:

Student
```
repeat drop_main_goal;
open_theory "z_exercises_2";
set_pc "z_language_ext" ;
```

SML
```
val ZE2 = [
("ZE2.1", ⓩ∀a:𝕌×𝕌•(a.1,a.2) = a⌝),
("ZE2.2", ⓩ[X,Y](∀ p: ℙ (X × Y)•
                        {x:X; y:Y | (x,y) ∈ p}
                =       {z:X × Y | z ∈ p})⌝),
("ZE2.3", ⓩ[x:ℤ | x > 0] = {x:ℤ | x>0 • (x ≙ x)}⌝)];
```

The following problems are more difficult, typically the proofs involve about four steps each.

SML
```
val ZE3 = [
("ZE3.1", ⓩ(λx:ℤ• x+1) 3 = 4⌝),
("ZE3.2", ⓩ{(1,2), (3,4)} 3 = 4⌝),
("ZE3.3", ⓩ(1, ~2) ∈ (abs _) ⇒ abs 1 = ~2⌝),
("ZE3.4", ⓩ∀ i,j:ℤ• (i,j) ∈ (abs _) ⇒ abs i = j⌝),
("ZE3.5", ⓩ∀i:ℤ• abs i ∈ ℕ⌝),
("ZE3.6", ⓩ(μx:ℤ | x=3 • x*x) = 9⌝),
("ZE3.7", ⓩ25 ∈ {y:ℤ • y*y}⌝),
("ZE3.8", ⓩ(a × b × c) = (d × e × f) ⇒ (a × b) = (d × e) ∨ (c ∩ f) = ∅⌝),
("ZE3.9", ⓩ[X,Y](∀ p: ℙ (X × Y)•
                        (∀ x:X; y:Y• (x,y) ∈ p)
                ⇔       (∀ z:X × Y• z ∈ p))⌝),
("ZE3.10", ⓩ[File | people = {}] = {File | people = {}}⌝),
("ZE3.11", ⓩ⟨a,b⟩ = ⟨c,d⟩ ⇒ a=c ∧ b=d⌝),
("ZE3.12", ⓩ⟨a,b⟩ = ⟨d,e⟩ ⇒ ⟨b,d⟩ = ⟨e,a⟩⌝)];
```

Hints for group ZE3:

1. Use *conv_tac(MAP_C z_β_conv)*.
2. Use *z_app_eq_tac*.
3. Forward chain (*all_fc_tac*) using *z_→_∈_rel_⇔_app_eq_thm*. But first you need to get into the assumptions the things it needs to chain on.
4. Very similar to number 3.
5. Forward chain using *z_fun_∈_clauses*.
6. Specialise the result of applying *z_μ_rule* to the μ expression (using *z_∀_elim*) and strip this into the assumptions. Then use the "two tactic" method (i.e. specialise assumptions as necessary) to derive a contradiction. The last step requires rewriting an assumption to make the contradiction apparent.
7. Rewriting gives and existential which can be solved using *z_∃_tac*. Alternatively a proof by contradiction can be used, but this needs rewriting an assumption at the end to get the contradiction out.
8. The proof must begin by using extensionality (either use proof context *z_library_ext* or rewrite with *z_sets_ext_clauses*). A straightforward proof by contradiction is possible using the "two tactic method".
9. Use of $z\_sel_t\_intro\_conv$ is necessary in this proof.
10. The easiest proof is obtained by a single *z_strip_tac* in proof context *z_library_ext* followed by *prove_tac* in proof context *z_library* which leaves just one existential subgoal.
11. Two tactic method in proof context *z_library_ext* suffices.
12. Similar to the previous example.

7.2.2 Propositional Schema Calculus

We define five schemas with distinct but compatible signatures called Pab, Qac, Rbc, Sabc and Tde and then prove the following goals showing that the schema calculus operators behave in the same way as the ordinary logical connectives. The names of the schemas are chosen to remind us of the signatures, since this is of significance in the examples. (The following lines of Z are for preparing the exercises database.)

```
Z
│Pab ≙ [a,b:ℤ]
```

```
Z
│Qac ≙ [a,c:ℤ]
```

```
Z
│Rbc ≙ [b,c:ℤ]
```

```
Z
│Sabc ≙ [a,b,c:ℤ]
```

```
Z
│Tde ≙ [d,e:ℤ]
```

The following problems are analogous to theorems taken from Principia Mathematica *2, and are shown together with their reference numbers. Set the context with:

```
Student
│repeat drop_main_goal;
│open_theory "z_exercises_2";
│set_pc "z_language";
```

```
SML
│val SCPM2 = [
│("*2.02", ⓩ Π((Qac ⇒ (Pab ⇒ Qac))⦂𝕌)⌝),
│("*2.03", ⓩ Π(((Pab ⇒ ¬ Qac) ⇒ (Qac ⇒ ¬ Pab))⦂𝕌)⌝),
│("*2.15", ⓩ Π(((¬ Pab ⇒ Qac) ⇒ (¬ Qac ⇒ Pab))⦂𝕌)⌝),
│("*2.16", ⓩ Π(((Pab ⇒ Qac) ⇒ (¬ Qac ⇒ ¬ Pab))⦂𝕌)⌝),
│("*2.17", ⓩ Π(((¬ Qac ⇒ ¬ Pab) ⇒ (Pab ⇒ Qac))⦂𝕌)⌝),
│("*2.04", ⓩ Π(((Pab ⇒ Qac ⇒ Rbc) ⇒ (Qac ⇒ Pab ⇒ Rbc))⦂𝕌)⌝),
│("*2.05", ⓩ Π(((Qac ⇒ Rbc) ⇒ (Pab ⇒ Qac) ⇒ (Pab ⇒ Rbc))⦂𝕌)⌝),
│("*2.06", ⓩ Π(((Pab ⇒ Qac) ⇒ (Qac ⇒ Rbc) ⇒ (Pab ⇒ Rbc))⦂𝕌)⌝),
│("*2.08", ⓩ Π((Pab ⇒ Pab)⦂𝕌)⌝),
│("*2.21", ⓩ Π((¬ Pab ⇒ (Pab ⇒ Qac))⦂𝕌)⌝)];
```

The following are analogous to Principia Mathematica *3

SML
```
val SCPM3 = [
("*3.01", ⓩ Π((Pab ∧ Qac ⇔ ¬(¬ Pab ∨ ¬ Qac))⊕⊕𝕌)⌝),
("*3.2", ⓩ Π((Pab ⇒ Qac ⇒ Pab ∧ Qac)⊕⊕𝕌)⌝),
("*3.26", ⓩ Π((Pab ∧ Qac ⇒ Pab)⊕⊕𝕌)⌝),
("*3.27", ⓩ Π((Pab ∧ Qac ⇒ Qac)⊕⊕𝕌)⌝),
("*3.3", ⓩ Π(((Pab ∧ Qac ⇒ Rbc) ⇒ (Pab ⇒ Qac ⇒ Rbc))⊕⊕𝕌)⌝),
("*3.31", ⓩ Π(((Pab ⇒ Qac ⇒ Rbc) ⇒ (Pab ∧ Qac ⇒ Rbc))⊕⊕𝕌)⌝),
("*3.35", ⓩ Π(((Pab ∧ (Pab ⇒ Qac)) ⇒ Qac)⊕⊕𝕌)⌝),
("*3.43", ⓩ Π(((Pab ⇒ Qac) ∧ (Pab ⇒ Rbc) ⇒ (Pab ⇒ Qac ∧ Rbc))⊕⊕𝕌)⌝),
("*3.45", ⓩ Π(((Pab ⇒ Qac) ⇒ (Pab ∧ Rbc ⇒ Qac ∧ Rbc))⊕⊕𝕌)⌝),
("*3.47", ⓩ Π(((Pab ⇒ Rbc) ∧ (Qac ⇒ Sabc) ⇒ (Pab ∧ Qac ⇒ Rbc ∧ Sabc))⊕⊕𝕌)⌝)];

```

Problems analogous to results in Principia Mathematica *4.

SML
```
val SCPM4 = [
("*4.1", ⓩ Π((Pab ⇒ Qac) ⇔ (¬ Qac ⇒ ¬ Pab)⊕⊕𝕌)⌝),
("*4.11", ⓩ Π((Pab ⇔ Qac) ⇔ (¬ Pab ⇔ ¬ Qac)⊕⊕𝕌)⌝),
("*4.13", ⓩ Π((Pab ⇔ (¬ ¬ Pab))⊕⊕𝕌)⌝),
("*4.2", ⓩ Π((Pab ⇔ Pab)⊕⊕𝕌)⌝),
("*4.21", ⓩ Π((Pab ⇔ Qac) ⇔ (Qac ⇔ Pab)⊕⊕𝕌)⌝),
("*4.22", ⓩ Π(((Pab ⇔ Qac) ∧ (Qac ⇔ Rbc) ⇒ (Pab ⇔ Rbc))⊕⊕𝕌)⌝),
("*4.24", ⓩ Π((Pab ⇔ (Pab ∧ Pab))⊕⊕𝕌)⌝),
("*4.25", ⓩ Π((Pab ⇔ (Pab ∨ Pab))⊕⊕𝕌)⌝),
("*4.3", ⓩ Π((Pab ∧ Qac ⇔ Qac ∧ Pab)⊕⊕𝕌)⌝),
("*4.31", ⓩ Π((Pab ∨ Qac ⇔ Qac ∨ Pab)⊕⊕𝕌)⌝),
("*4.33", ⓩ Π(((Pab ∧ Qac) ∧ Rbc ⇔ Pab ∧ (Qac ∧ Rbc))⊕⊕𝕌)⌝),
("*4.4", ⓩ Π((Pab ∧ (Qac ∨ Rbc) ⇔ (Pab ∧ Qac) ∨ (Pab ∧ Rbc))⊕⊕𝕌)⌝),
("*4.41", ⓩ Π((Pab ∨ (Qac ∧ Rbc) ⇔ (Pab ∨ Qac) ∧ (Pab ∨ Rbc))⊕⊕𝕌)⌝),
("*4.71", ⓩ Π(((Pab ⇒ Qac) ⇔ (Pab ⇔ (Pab ∧ Qac)))⊕⊕𝕌)⌝),
("*4.73", ⓩ Π((Qac ⇒ (Pab ⇔ (Pab ∧ Qac)))⊕⊕𝕌)⌝)];
```

Problems analogous to results in Principia Mathematica *5.

SML
```
val SCPM5 = [
("*5.1", ⓩ Π((Pab ∧ Qac ⇒ (Pab ⇔ Qac))⊕⊕𝕌)⌝),
("*5.32", ⓩ Π(((Pab ⇒ (Qac ⇔ Rbc)) ⇒ ((Pab ∧ Qac) ⇔ (Pab ∧ Rbc)))⊕⊕𝕌)⌝),
("*5.6", ⓩ Π(((Pab ∧ ¬ Qac ⇒ Rbc) ⇒ (Pab ⇒ (Qac ∨ Rbc)))⊕⊕𝕌)⌝)];

repeat drop_main_goal;
```

7.2.3 Schema Calculus Quantification

Set the context with:

Student
```
open_theory "z_exercises_2";
set_pc "z_library";
```

SML
```
val SCPM9 =[
("*9.01", ⓩΠ(((¬ (∀Qac• Sabc)) ⇔ (∃Qac• ¬ Sabc))⦂𝕌)⌝),
("*9.02", ⓩΠ(((¬ (∃Qac• Sabc) ⇔ (∀Qac• ¬ Sabc)))⦂𝕌)⌝)];
```

SML
```
val SCPM10 =[
("*10.01", ⓩΠ(((∃Qac• Sabc) ⇔ ¬ (∀Qac• ¬ Sabc))⦂𝕌)⌝),
("*10.21", ⓩΠ(((∀Qac• Tde ⇒ Sabc) ⇔ Tde ⇒ (∀Qac• Sabc))⦂𝕌)⌝),
("*10.22", ⓩΠ(((∀Rbc• Sabc ∧ Rbc) ⇔ (∀Rbc• Sabc) ∧ (∀Rbc• Rbc))⦂𝕌)⌝)];
```

7.3 Paragraphs

SML
```
open_theory "z_library";
new_theory "z_exercises_3";
```

7.3.1 Axiomatic Descriptions and Generics

Do the following exercises in axiomatic mode to avoid the consistency proofs which would otherwise be necessary. (Consistency-proof examples follow.) Thus set the context as follows:

Student
```
repeat drop_main_goal;
open_theory "z_exercises_3";
set_pc "z_library";
set_flags [("z_type_check_only", false), ("z_use_axioms", true)];
```

1. In this context, using a *fun* fixity paragraph and a generic axiomatic description define a conditional construct *if* a *then* b *else* c. (Hints: you will need to use the higher order capabilities of ProofPower HOL for this one. The first parameter will have type BOOL.)

2. Using the specification prove the following result:

```
set_goal ([], ⓩif 2>1 then 1 else 0 = 1⌝);
```

7.3.2 Consistency Proofs

1. Set the context as follows:

   Student
   ```
   repeat drop_main_goal;
   open_theory "z_exercises_3";
   set_pc "z_library";
   set_flags [("z_type_check_only", false), ("z_use_axioms", false)];
   ```

2. Use an axiomatic description to define a global variable *num* whose value is a natural number between 4 and 50.

3. Prove the consistency result for this description and save it.

4. now use the specification to prove that:

   $?\vdash num \geq 0$

   and save the result in the theory.

7.3.3 Reasoning using Schema Definitions

Set the context as follows:

Student
```
repeat drop_main_goal;
open_theory "z_exercises_3";
set_pc "z_library";
set_flags [("z_type_check_only", false), ("z_use_axioms", false)];
```

7.3.3.1 Simple Pre-conditions and Refinement

Using the following definitions (which are predefined in the 'exercises' database):

Z
```
┌─STATE──────────────
│ r : ℤ ↔ ℤ
└────────────────────
```

Z
```
┌─OP─────────────────
│ STATE; STATE′; i?:ℕ
└────────────────────
```

Z
```
┌─OP2────────────────
│ OP
├──────────
│ r′ = {i?} ⩤ r
└────────────────────
```

formulate and prove the following simple conjectures:

1. $?\vdash$ *pre OP* $\Leftrightarrow$ *i?* $\geq$ *0*
2. $?\vdash$ (*pre OP* $\Rightarrow$ *pre OP2*) $\wedge$ (*pre OP* $\wedge$ *OP2* $\Rightarrow$ *OP*) that is, *OP2* is a correct refinement of *OP*

7.3.3.2 The Vending Machine

Write a specification of a vending machine which has a stock of a single variety of commodity (chocolate bar for example), and accepts cash (input), outputs (a number of) goods and an amount of cash in change.

Do this in two stages of which the second is a correct (algorithmic) refinement of the first, and prove that this is the case.

Define the property of such systems that they do not undercharge for the goods they deliver and prove that the specified systems have this property.

7.4 The Z ToolKit

SML
```
open_theory "z_library";
new_theory "z_exercises_4";
```

7.4.1 Sets

The following problems are all in the theory of elementary sets as defined in section 4.1 of *The Z Notation* [4]. Following [4] we have used free-variable formulations, though this is not usually recommended, since universally quantified results (over “$\mathbb{U}$”) are usually needed for rewriting. Proofs of the results quantified over $\mathbb{U}$ are very similar.

All the results are provable using the proof context `z_sets_ext` (which provides for extensional proofs of results in elementary set theory). Set the context as follows:

Student
```
repeat drop_main_goal;
open_theory "z_exercises_4";
set_pc "z_sets_ext";
```

7.4.1.1 Results Provable by Stripping

SML
```
val Z1 = [
("Z1.1", ⓩ a ∪ a = a ∪ {} = a ∩ a = a \ {} = a⌝),
("Z1.2", ⓩ a ∩ {} = a \ a = {} \ a = {}⌝),
("Z1.3", ⓩ a ∪ b = b ∪ a⌝),
("Z1.4", ⓩ a ∩ b = b ∩ a⌝),
("Z1.5", ⓩ a ∪ (b ∪ c) = (a ∪ b) ∪ c⌝),
("Z1.6", ⓩ a ∩ (b ∩ c) = (a ∩ b) ∩ c⌝),
("Z1.7", ⓩ a ∪ (b ∩ c) = (a ∪ b) ∩ (a ∪ c)⌝),
("Z1.8", ⓩ a ∩ (b ∪ c) = (a ∩ b) ∪ (a ∩ c)⌝),
("Z1.9", ⓩ (a ∩ b) ∪ (a \ b) = a⌝),
("Z1.10", ⓩ ((a \ b) ∩ b) = {}⌝),
("Z1.11", ⓩ a \ (b \ c) = (a \ b) ∪ (a ∩ c)⌝),
("Z1.12", ⓩ (a \ b) \ c = a \ (b ∪ c)⌝),
("Z1.13", ⓩ a ∪ (b \ c) = (a ∪ b) \ (c \ a)⌝),
("Z1.14", ⓩ a ∩ (b \ c) = (a ∩ b) \ c⌝),
("Z1.15", ⓩ (a ∪ b) \ c = (a \ c) ∪ (b \ c)⌝)];
```

SML

```
val Z2 = [
("Z2.1", ⓩ a \ (b ∩ c) = (a \ b) ∪ (a \ c)⌝),
("Z2.2", ⓩ ¬ x ∈ {}⌝),
("Z2.3", ⓩ a ⊆ a⌝),
("Z2.4", ⓩ ¬ a ⊂ a⌝),
("Z2.5", ⓩ {} ⊆ a⌝),
("Z2.6", ⓩ ⋃ {} = {}⌝),
("Z2.7", ⓩ ⋂ {} = 𝕌⌝)];
```

For the following stripping alone will not suffice. The “two tactic method” will solve them all, so will “prove_tac”.

SML

```
val Z3 = [
("Z3.1", ⓩ a ⊆ b ⇔ a ∈ ℙ b⌝),
("Z3.2", ⓩ a ⊆ b ∧ b ⊆ a ⇔ a = b⌝),
("Z3.3", ⓩ ¬ (a ⊂ b ∧ b ⊂ a)⌝),
("Z3.4", ⓩ a ⊆ b ∧ b ⊆ c ⇒ a ⊆ c⌝),
("Z3.5", ⓩ a ⊂ b ∧ b ⊂ c ⇒ a ⊂ c⌝),
("Z3.6", ⓩ {} ⊂ a ⇔ ¬ a = {}⌝),
("Z3.7", ⓩ ⋃ (a ∪ b) = (⋃ a) ∪ (⋃ b)⌝),
("Z3.8", ⓩ ⋂ (a ∪ b) = (⋂ a) ∩ (⋂ b)⌝),
("Z3.9", ⓩ  a ⊆ b ⇒ ⋃ a ⊆ ⋃ b ⌝),
("Z3.10", ⓩ  a ⊆ b ⇒ ⋂ b ⊆ ⋂ a ⌝)];
```

SML

```
val Z3b = [
("Z3b.1", ⓩ a ⊆ b ∧ b ⊆ a ⇔ a = b⌝),
("Z3b.2", ⓩ a ⊂ b ∧ b ⊂ c ⇒ a ⊂ c⌝),
("Z3b.3", ⓩ {} ⊂ a ⇔ ¬ a = {}⌝)];
```

7.4.2 Relations

Set the context by:

Student
```
repeat drop_main_goal;
open_theory "z_exercises_4";
set_pc "z_rel_ext";
```

SML
```
val Z4 = [
("Z4.1", ⓩ[X,Y] (R ∈ X ↔ Y) ⇒ (∀ x : X • x ∈ dom R ⇔ (∃ y : Y • (x,y) ∈ R))⌝),
("Z4.2", ⓩ[X,Y] (R ∈ X ↔ Y) ⇒ (∀ y : Y • y ∈ ran R ⇔ (∃ x : X • (x,y) ∈ R))⌝),
("Z4.3", ⓩdom {x1 ↦ y1, x2 ↦ y2} = {x1, x2}⌝),
("Z4.4", ⓩran {x1 ↦ y1, x2 ↦ y2} = {y1, y2}⌝),
("Z4.5", ⓩdom (Q ∪ R) = dom Q ∪ dom R⌝),
("Z4.6", ⓩran (Q ∪ R) = ran Q ∪ ran R⌝),
("Z4.7", ⓩdom (Q ∩ R) ⊆ dom Q ∩ dom R⌝),
("Z4.8", ⓩran (Q ∩ R) ⊆ ran Q ∩ ran R⌝),
("Z4.9", ⓩdom {} = {}⌝),
("Z4.10", ⓩran {} = {}⌝)];
```

SML
```
val Z5 = [
("Z5.1", ⓩ[X,Y,Z] P ∈ X ↔ Y ∧ Q ∈ Y ↔ Z ⇒
        ((x ↦ z) ∈ P ⨾ Q ⇔ (∃ y : Y • (x,y) ∈ P ∧ (y,z) ∈ Q))⌝),
("Z5.2", ⓩP ⨾ (Q ⨾ R) = (P ⨾ Q) ⨾ R⌝)];
```

SML
```
val Z5b = [
("Z5b.1", ⓩ[X] (x ↦ x′) ∈ id X ⇔ x = x′ ∈ X⌝),
("Z5b.2", ⓩ(id X) ⨾ P = X ◁ P⌝),
("Z5b.3", ⓩP ⨾ id Y = P ▷ Y⌝),
("Z5b.4", ⓩ(id V) ⨾ id W = id (V ∩ W)⌝)];
```

SML
```
val Z5c = [
("Z5c.1", ⓩ[X,Y,Z] (∀x:𝕌;g:𝕌• g ∈ X ⇸ Y ∧ x ∈ dom g ⇒ (x, g x) ∈ g)⌝),
("Z5c.2", ⓩ[X,Y,Z] x ∈ dom g ∧ g x ∈ dom f ∧ g ∈ X ⇸ Y
        ∧ f ∈ Y ⇸ Z ⇒ (f ∘ g)(x) = f(g(x))⌝)];
```

SML
```
val Z6 = [
("Z6.1", ⓩdom (S ◁ R) = S ∩ dom R⌝),
("Z6.2", ⓩran (R ▷ T) = ran R ∩ T⌝),
("Z6.3", ⓩS ◁ R ⊆ R⌝),
("Z6.4", ⓩR ▷ T ⊆ R⌝),
("Z6.5", ⓩ(S ◁ R) ▷ T = S ◁ (R ▷ T)⌝),
("Z6.6", ⓩS ◁ (V ◁ R) = (S ∩ V) ◁ R⌝),
("Z6.7", ⓩ(R ▷ T) ▷ W = R ▷ (T ∩ W)⌝)];
```

SML
```
val Z6b = [
("Z6b.1", ⓩran R ⊆ Y ⇒ (S ◁ R = (id S) ⨾ R = (S × Y) ∩ R)⌝),
("Z6b.2", ⓩdom R ⊆ X ⇒ (R ▷ T = R ⨾ (id T) = R ∩ (X × T))⌝)];
```

SML
```
val Z7 = [
("Z7.1", ⓩdom R ⊆ X ⇒ S ⩤ R = (X \ S) ◁ R⌝),
("Z7.2", ⓩran R ⊆ Y ⇒ R ⩥ T = R ▷ (Y \ T)⌝),
("Z7.3", ⓩ(S ◁ R) ∪ (S ⩤ R) = R⌝),
("Z7.4", ⓩ(R ▷ T) ∪ (R ⩥ T) = R⌝)];
```

SML
```
val Z8 = [
(* ("Z8.1", ⓩ(y ↦ x) ∈ R ∼ ⇔ (x ↦ y) ∈ R⌝), *)
("Z8.2", ⓩ(R ∼) ∼ = R⌝),
("Z8.3", ⓩ(Q ⨾ R) ∼ = R ∼ ⨾ Q ∼⌝),
("Z8.4", ⓩdom(R ∼) = ran R⌝),
("Z8.5", ⓩran(R ∼) = dom R⌝)];
```

SML
```
val Z8b = [
("Z8b.1", ⓩ(id V) ∼ = id V⌝),
("Z8b.2", ⓩid(dom R) ⊆ R ⨾ (R ∼)⌝),
("Z8b.3", ⓩid(ran R) ⊆ (R ∼) ⨾ R⌝)];
```

SML

```
val Z9 = [
("Z9.1", ⓩS ⊆ X ⇒ (y ∈ R ⦇ S ⦈ ⇔ (∃ x : X • x ∈ S ∧ (x,y) ∈ R))⌝),
("Z9.2", ⓩR ⦇ S ⦈ = ran(S ◁ R)⌝),
("Z9.3", ⓩdom(Q ⨾ R) = (Q ~) ⦇ dom R ⦈⌝),
("Z9.4", ⓩran(Q ⨾ R) = R ⦇ ran Q ⦈⌝),
("Z9.5", ⓩR ⦇ S ∪ T ⦈ = R ⦇ S ⦈ ∪ R ⦇ T ⦈⌝),
("Z9.6", ⓩR ⦇ S ∩ T ⦈ ⊆ R ⦇ S ⦈ ∩ R ⦇ T ⦈⌝),
("Z9.7", ⓩR ⦇ dom R ⦈ = ran R⌝)];
```

SML

```
val Z9b = [
("Z9b.1", ⓩdom R = first ⦇ R ⦈⌝),
("Z9b.2", ⓩran R = second ⦇ R ⦈⌝)];
```

SML

```
val Z10 = [
("Z10.1", ⓩR ⊕ R = R⌝),
("Z10.2", ⓩP ⊕ (Q ⊕ R) = (P ⊕ Q) ⊕ R⌝),
("Z10.3", ⓩ∅ ⊕ R = R ⊕ ∅ = R⌝),
("Z10.4", ⓩdom Q ∩ dom R = ∅ ⇒ Q ⊕ R = Q ∪ R⌝),
("Z10.5", ⓩV ◁ (Q ⊕ R) = (V ◁ Q) ⊕ (V ◁ R)⌝),
("Z10.6", ⓩ(Q ⊕ R) ▷ W ⊆ (Q ▷ W) ⊕ (R ▷ W)⌝)];
```

SML

```
val Z10b = [
("Z10b.1", ⓩdom(Q ⊕ R) = (dom Q) ∪ (dom R)⌝),
("Z10b.2", ⓩf ∈ X ⇸ Y ∧ g ∈ X ⇸ Y ⇒
        x ∈ (dom f) \ (dom g) ⇒ (f ⊕ g) x = f x⌝),
("Z10b.3", ⓩg ∈ X ⇸ Y ∧ x ∈ dom g ⇒ (f ⊕ g) x = g x⌝)];
```

SML

```
val Z11 = [
("Z11.1", ⓩR ⊆ R ⁺⌝),
("Z11.2", ⓩid X ⊆ R *⌝),
("Z11.3", ⓩR ⊆ R*⌝)];
```

SML
```
val Z11b = [
("Z11b.1", ⓩR + ⨾ (R +) ⊆ R +⌝),
("Z11b.2", ⓩ(R +) + = R +⌝),
("Z11b.3", ⓩ(R *) * = R *⌝),
("Z11b.4", ⓩR ⊆ Q ∧ Q ⨾ Q ⊆ Q ⇒ R + ⊆ Q⌝),
("Z11b.5", ⓩR* ⨾ R* = R*⌝),
("Z11b.6", ⓩid X ⊆ Q ∧ R ⊆ Q ∧ Q ⨾ Q ⊆ Q ⇒ R* ⊆ Q⌝),
("Z11b.7", ⓩR* = R + ∪ id X = (R ∪ id X)+⌝),
("Z11b.8", ⓩR+ = R ⨾ R * = R * ⨾ R⌝),
("Z11b.9", ⓩS ⊆ R * ⦇ S ⦈⌝),
("Z11b.10", ⓩS ⊆ T ∧ R ⦇ T ⦈ ⊆ T ⇒ R * ⦇S⦈ ⊆ T⌝),
("Z11b.11", ⓩR ⦇R * ⦇ S ⦈ ⦈ ⊆ R * ⦇S⦈⌝)];
```

7.4.3 Functions

Set the context:

Student
```
repeat drop_main_goal;
open_theory "z_exercises_4";
set_pc "z_fun_ext";
```

SML
```
val Z12 = [
("Z12.1", ⓩ f ∈ X ⇸ Y ∧ (x, y) ∈ f ⇒ f x = y ⌝),
("Z12.2", ⓩ f ∈ X ⤔ Y ∧ (x, y) ∈ f ⇒ f x = y ⌝),
("Z12.3", ⓩ f ∈ X ↣ Y ∧ (x, y) ∈ f ⇒ f x = y ⌝),
("Z12.4", ⓩ f ∈ X ⤔ Y ∧ (x, y) ∈ f ⇒ f x = y ⌝),
("Z12.5", ⓩ f ∈ X ⤖ Y ∧ (x, y) ∈ f ⇒ f x = y ⌝),
("Z12.6", ⓩ f ∈ X ⤀ Y ∧ (x, y) ∈ f ⇒ f x = y ⌝)
];
val Z12a = [
("Z12a.1", ⓩ f ∈ (X ⇸ Y) ∪ (X ⤔ Y) ∪ (X ↣ Y)
                ∪ (X ⤔ Y) ∪ (X ⤖ Y) ∪ (X ⤀ Y)
                ∧ (x, y) ∈ f ⇒ f x = y ⌝)];
```

SML
```
val Z12b = [
("Z12b.1", ⓩf ∈ X ↔ Y ⇒ (f ∈ X ⇸ Y ⇔ f ∘ f∼ = id(ran f))⌝),
("Z12b.2", ⓩf ∈ X ⤔ Y ⇔ (f ∈ X ⇸ Y ∧ f ∼ ∈ Y ⇸ X)⌝),
("Z12b.3", ⓩf ∈ X ⤔ Y ⇔ (f ∈ X ⇸ Y ∧ f ∼ ∈ Y ⇸ X)⌝),
("Z12b.4", ⓩf ∈ X ↣ Y ⇔ (f ∈ X → Y ∧ f ∼ ∈ Y ⇸ X)⌝),
("Z12b.5", ⓩf ∈ X ⤔ Y ⇒ f ⦇S ∩ T⦈ = f ⦇S⦈ ∩ f ⦇T⦈⌝),
("Z12b.6", ⓩf ∈ X ⤖ Y ⇔ (f ∈ X → Y ∧ f ∼ ∈ Y → X)⌝),
("Z12b.7", ⓩf ∈ X ⤀ Y ⇒ f ∘ f ∼ = id Y⌝)];
```

7.4.4 Numbers and Finiteness

1. Set the context to axiomatic mode as follows:

Student
```
repeat drop_main_goal;
open_theory "z_exercises_4";
set_pc "z_library";
set_flags [("z_type_check_only", false), ("z_use_axioms", true)];
```

   and then give an inductive definition using a Z axiomatic description of the function Σ which maps the natural number n to the sum of the first n natural numbers.

2. Using *z_ℕ_induction_tac* prove that: $\forall n \colon \mathbb{N} \bullet (\Sigma\ n) * 2 = n * (n+1)$‘

3. Set the context as follows:

Student
```
repeat drop_main_goal;
open_theory "z_exercises_4";
set_pc "z_library_ext";
set_flags [("z_type_check_only", false), ("z_use_axioms", true)];
```

   and then, using the specification of (_.._)(obtained using *z_get_spec*) and forward chaining (*all_fc_tac*) with the theorem *z_≤_trans_thm* prove that:

   $\forall\ x, y : \mathbb{Z} \bullet x \leq y \Rightarrow (0 .. x) \subseteq (0 .. y)$

4. Now prove the harder result: $\forall\ x, y : \mathbb{Z} \bullet \neg\ x \leq y \Rightarrow (0 .. y) \subseteq (0 .. (x - 1))$

   Helpful theorems in this case are *z_≤_less_trans_thm*, *z_≤_≤_0_thm*, *z_plus_order_thm* and *z_minus_thm*.

SML
```
val ZNum = [
("ZNum.2", ⓩ∀ x, y : ℤ • x ≤ y ⇒ (0 .. x) ⊆ (0 .. y)⌝),
("ZNum.3", ⓩ∀ x, y : ℤ • ¬ x ≤ y ⇒ (0 .. y) ⊆ (0 .. (x − 1))⌝)];
```

(The term for the first goal cannot be entered until the definition of Σ has been processed).

SOLUTIONS TO EXERCISES

The section numbers of this chapter correspond to those in the Exercises chapter.

The source script for this chapter is the file usr011S.doc. This may be converted into an ML script using *docsml*, or alternatively the formal text can be entered interactively by cut-and-paste from the source document into a command tool in which ProofPower is running.

8.1 The Z Predicate Calculus

8.1.1 Forward Propositional Proofs

Solutions to forward proof exercises.

SML
```
repeat drop_main_goal;
open_theory "z_exercises_1";
set_pc "z_library";
```

SML
```
(* (a) *)
val ex1a_thm1 = asm_rule ⓩ a⇒b⌝;
val ex1a_thm2 = asm_rule ⓩ b⇒c⌝;
val ex1a_thm3 = asm_rule ⓩ Π(a)⌝;
val ex1a_thm4 = ⇒_elim ex1a_thm1 ex1a_thm3;
val ex1a_thm = ⇒_elim ex1a_thm2 ex1a_thm4;

save_thm ("ex1a_thm", ex1a_thm);

(* (b) *)
val ex1b_thm1 =
 ⇒_elim (asm_rule ⓩ a⇒b⇒c⌝)(asm_rule ⓩ Π(a)⌝);
val ex1b_thm =
 ⇒_elim ex1b_thm1 (asm_rule ⓩ Π(b)⌝);

save_thm ("ex1b_thm", ex1b_thm);

(* (c) *)
val ex1c_thm = ⇒_intro ⓩ Π(a)⌝ ex1b_thm;

save_thm ("ex1c_thm", ex1c_thm);
```

```
(* (d) *)
val ex1d_thm1 = ⇒_intro ⓩ Π(b)⌝ ex1c_thm;
val ex1d_thm = ⇒_intro ⓩ a ⇒ b ⇒ c⌝ ex1d_thm1;

save_thm ("ex1d_thm", ex1d_thm);
```

8.1.2 Goal Oriented Propositional Proofs

For each of the exercises either:

SML
```
a (prove_tac[]);
```

or:

SML
```
a (REPEAT z_strip_tac);
```

will complete the proof in one step.

To get an understanding of how this is done the proofs may be obtained in steps by manually repeating:

SML
```
a z_strip_tac;
```

or:

SML
```
a step_strip_tac;
```

We show here just one example of proof by stripping, you may work through as many other examples as you like.

SML
```
setlg "*2.02" PM2;
```

The results may be proven automatically as follows:

SML
```
map prove_and_store PM2;
map prove_and_store PM3;
map prove_and_store PM4;
map prove_and_store PM5;
```

Though technically these may be considered forward proofs since they use *prove_rule*, prove rule itself uses *prove_tac*, and so the difference between completely automatic forward and backward proof is insignificant.

8.1.3 Forward Predicate Calculus Proofs

Forward proof using elementary rules is less convenient in Z because of the extra complications arising when quantifiers are eliminated or introduced.

SML

```
repeat drop_main_goal;
open_theory "z_exercises_1";
set_pc "z_library";


(* 1(a) *)
val ex1a_thm1 = z_∀_elim ⓩ 0⌝ z_ℕ_¬_plus1_thm;
(* 1(b) *)
val ex1b_thm = z_∀_elim ⓩ x∗x⌝ z_ℕ_¬_plus1_thm;

(* 2 *)
val ex2_thm = prove_rule [z_≤_trans_thm]
        ⓩ i ≤ j ∧ j ≤ k ⇒ i ≤ k⌝ ;

(* note that:
z_∀_elim ⓩ(i≘i⦂ℤ, j≘j⦂ℤ, k≘k⦂ℤ)⌝ z_≤_trans_thm;
doesn't do the job.
*)

(* 3(a) *)
val ex3a_thm = prove_rule [z_ℕ_¬_plus1_thm]
        ⓩ¬ 0 + 1 = 0⌝;
(* 3(b) *)
val ex3b_thm = prove_rule [z_ℕ_¬_plus1_thm]
        ⓩ x ∗ x ∈ ℕ ⇒ ¬ x ∗ x + 1 = 0⌝;

(* 4(a) *)
val ex4a_thm = prove_rule[z_¬_less_thm]
        ⓩ¬ 0 < 1 ⇔ 1 ≤ 0⌝;
(* 4(b) *)
val ex4b_thm = prove_rule[z_≤_trans_thm]
        ⓩ∀ n:ℤ • 3 ≤ x ∗ x ∧ x ∗ x ≤ n ⇒ 3 ≤ n⌝;

(* 5(a) *)
val ex5a_thm = prove_rule[z_≤_clauses]
        ⓩ∀ i, m, n :ℤ• i + m ≤ i + n ⇔ m ≤ n⌝;
(* 5(b) *)
val ex5b_thm = prove_rule[z_≤_clauses]
        ⓩ∀ m, i, n :ℤ• i + m ≤ i + n ⇔ m ≤ n⌝;
```

8.1.4 Goal Oriented Predicate Calculus Proof

These proofs are also conducted automatically.

SML
```
map prove_and_store PM9;
map prove_and_store PM10;
map prove_and_store PM10b;
map prove_and_store PM11;
```

The problems in PM11b cannot be solved by *prove_tac*.

SML
```
set_goal([], lassoc3 PM11b "∗11.32");
```

ProofPower output
```
...
(∗ ?⊢ ∗)  ⌜[Y]((∀ x, y : Y • φ (x, y) ⇒ ψ (x, y))
               ⇒ (∀ x, y : Y • φ (x, y))
               ⇒ (∀ x, y : Y • ψ (x, y)))⌝
...
```

SML
```
a contr_tac;
```

ProofPower output
```
...
(∗  4 ∗)  ⌜∀ x, y : Y • φ (x, y) ⇒ ψ (x, y)⌝
(∗  3 ∗)  ⌜∀ x, y : Y • φ (x, y)⌝
(∗  2 ∗)  ⌜x ∈ Y⌝
(∗  1 ∗)  ⌜y ∈ Y⌝

(∗ ?⊢ ∗)  ⌜ψ (x, y)⌝
...
```

SML
```
a(z_spec_nth_asm_tac 5 ⌜(x ≙ x, y ≙ y)⌝);
```

ProofPower output
```
...
(∗  6 ∗)  ⌜∀ x, y : Y • φ (x, y) ⇒ ψ (x, y)⌝
(∗  5 ∗)  ⌜∀ x, y : Y • φ (x, y)⌝
(∗  4 ∗)  ⌜x ∈ Y⌝
(∗  3 ∗)  ⌜y ∈ Y⌝
(∗  2 ∗)  ⌜¬ ψ (x, y)⌝
(∗  1 ∗)  ⌜¬ φ (x, y)⌝

(∗ ?⊢ ∗)  ⌜false⌝
...
```

SML

```
a(z_spec_nth_asm_tac 5 ⓩ(x ≙ x, y ≙ y)⌝);
```

ProofPower output

```
Tactic produced 0 subgoals:
Current and main goal achieved
val it = () : unit
```

SML

```
save_pop_thm("*11.32");
```

SML

```
setlg "*11.45" PM11b;
a contr_tac;
(* *** Goal "1" *** *)
a(z_spec_nth_asm_tac 1 ⓩ(x ≙ x', y ≙ y')⌝);
(* *** Goal "2" *** *)
a(z_spec_nth_asm_tac 1 ⓩ(x ≙ x, y ≙ y)⌝);
(* *** Goal "3" *** *)
a(z_spec_nth_asm_tac 1 ⓩ(x ≙ x', y ≙ y')⌝);
save_pop_thm("*11.45");
```

SML

```
setlg "*11.54" PM11b;
a contr_tac;
(* *** Goal "1" *** *)
a(z_spec_nth_asm_tac 1 ⓩ(x ≙ x)⌝);
(* *** Goal "2" *** *)
a(z_spec_nth_asm_tac 1 ⓩ(y ≙ y)⌝);
(* *** Goal "3" *** *)
a(z_spec_nth_asm_tac 1 ⓩ(x ≙ x, y ≙ y)⌝);
save_pop_thm("*11.54");
setlg "*11.55" PM11b;
a contr_tac;
(* *** Goal "1" *** *)
a(z_spec_nth_asm_tac 1 ⓩ(x ≙ x)⌝);
a(z_spec_nth_asm_tac 1 ⓩ(y ≙ y)⌝);
(* *** Goal "2" *** *)
a(z_spec_nth_asm_tac 1 ⓩ(x ≙ x, y ≙ y)⌝);
save_pop_thm("*11.55");
```

SML
```
setlg "∗11.6" PM11b;
a contr_tac;
(∗ ∗∗∗ Goal "1" ∗∗∗ ∗)
a(z_spec_nth_asm_tac 1 ⓩ(y ≙ y)⌝);
a(z_spec_nth_asm_tac 1 ⓩ(x ≙ x)⌝);
(∗ ∗∗∗ Goal "2" ∗∗∗ ∗)
a(z_spec_nth_asm_tac 1 ⓩ(x ≙ x)⌝);
a(z_spec_nth_asm_tac 1 ⓩ(y ≙ y)⌝);
save_pop_thm("∗11.6");
```

SML
```
setlg "∗11.62" PM11b;
a contr_tac;
(∗ ∗∗∗ Goal "1" ∗∗∗ ∗)
a(z_spec_nth_asm_tac 6 ⓩ(x ≙ x, y ≙ y)⌝);
(∗ ∗∗∗ Goal "2" ∗∗∗ ∗)
a(z_spec_nth_asm_tac 6 ⓩ(x ≙ x)⌝);
a(z_spec_nth_asm_tac 1 ⓩ(y ≙ y)⌝);
save_pop_thm("∗11.62");
```

Forward chaining suffices. To show this we first delete the theorems from the theory:

SML
```
map delete_thm (map fst PM11b);
```

Then we write a function to do the proofs using a simpler approach:

SML
```
fun prove_and_store2 (key, term) = save_thm (key,
        tac_proof (([],term),
        (contr_tac
                THEN (all_asm_fc_tac[])
                THEN (all_asm_fc_tac[]))));

map prove_and_store2 PM11b;
```

8.1.5 Rewriting

8.1.5.1 Rewriting with the Subgoal Package

SML
```
repeat drop_main_goal;
open_theory "z_exercises_1";
set_pc "z_library_ext";
```

No solutions.

8.1.5.2 Combining Forward and Backward Proof

SML
```
repeat drop_main_goal;
open_theory "z_exercises_1";
set_pc "z_library";


```

1. :

SML
```
set_goal([],⌜Z x + y = y + x⌝);
a (rewrite_tac[]);
save_pop_thm "X6.1";
```

2. :

SML
```
set_goal([],⌜Z x + y + z = (x + y) + z⌝);
a (rewrite_tac[z_plus_assoc_thm]);
save_pop_thm "X6.2";
```

3. :

SML
```
set_goal([],⌜Z z + y + x = y + z + x⌝);
a (rewrite_tac[z_plus_assoc_thm1]);
save_pop_thm "X6.3";
```

4. :

SML
```
set_goal([],⌜Z x + y + z = y + z + x⌝);
a (rewrite_tac[z_∀_elim ⌜Z(i≙y,j≙z,k≙x)⌝
        z_plus_assoc_thm1]);
save_pop_thm "X6.4";
```

5. :

SML
```
set_goal([],⌜Z x + y + z + v = y + v + z + x⌝);
a (rewrite_tac[z_∀_elim ⌜Z x⌝ z_plus_order_thm]);
save_pop_thm "X6.5";
```

8.1.6 Stripping

No solutions.

8.2 Expressions and Schema Expressions

8.2.1 Expressions

SML
```
repeat drop_main_goal;
open_theory "z_exercises_2";
set_pc "z_library";
```

Group ZE1 are all provable automatically in proof context *z_library*.

SML
```
map prove_and_store ZE1;
```

Group ZE2 are provable automatically in proof context *z_library_ext*.

SML
```
repeat drop_main_goal;
open_theory "z_exercises_2";
set_pc "z_language_ext" ;

map prove_and_store ZE2;
```

Group ZE3 results are not automatically provable. Thought the results are primarily about the language, they make use of definitions in the Z ToolKit and therefore need to be conducted in the proof context *z_library*.

SML
```
set_pc "z_library";
```

SML
```
(* ZE3.1 *)
set_goal ([], ⓩ(λx:ℤ• x+1) 3 = 4⌝);
a (conv_tac (MAP_C z_β_conv));
a (rewrite_tac[]);
save_pop_thm "ZE3.1";
```

SML
```
(* ZE3.2 *)
set_goal ([], ⓩ{(1,2), (3,4)} 3 = 4⌝);
a (z_app_eq_tac);
a (rewrite_tac []);
a (REPEAT strip_tac);
save_pop_thm "ZE3.2";
```

SML
```
(* ZE3.3 *)
set_goal ([], ⓩ(1,~2) ∈ (abs _) ⇒ abs 1 = ~2⌝);
a (REPEAT strip_tac);
a (strip_asm_tac (z_get_spec ⓩ(abs _)⌝));
a (asm_tac (prove_rule[] ⓩ 1 ∈ ℤ⌝));
a (all_fc_tac [z_→_∈_rel_⇔_app_eq_thm]);
save_pop_thm "ZE3.3";
```

SML
```
(* ZE3.4 *)
set_goal ([], ⓩ∀ i,j:ℤ• (i,j) ∈ (abs _) ⇒ abs i = j⌝);
a (REPEAT strip_tac);
a (strip_asm_tac (z_get_spec ⓩ(abs _)⌝));
a (asm_tac (prove_rule[] ⓩ i ∈ ℤ⌝));
a (all_fc_tac [z_→_∈_rel_⇔_app_eq_thm]);
save_pop_thm "ZE3.4";
```

SML
```
(* ZE3.5 *)
set_goal ([], ⓩ∀i:ℤ• abs i ∈ ℕ⌝);
a (REPEAT strip_tac);
a (strip_asm_tac (z_get_spec ⓩ(abs _)⌝));
a (asm_tac (prove_rule[] ⓩ i ∈ ℤ⌝));
a (all_fc_tac [z_fun_∈_clauses]);
save_pop_thm "ZE3.5";
```

SML
```
(* ZE3.6 *)
set_goal ([], ⓩ(μx:ℤ | x=3 • x*x) = 9⌝);
a (strip_asm_tac (z_∀_elim ⓩ9⌝ (z_μ_rule ⓩ(μx:ℤ | x=3 • x*x)⌝)));
(* *** Goal "1" *** *)
a (var_elim_nth_asm_tac 2);
a (asm_ante_tac ⓩ¬ 3 * 3 = 9⌝ THEN rewrite_tac[]);
(* *** Goal "2" *** *)
a (z_spec_nth_asm_tac 1 ⓩ3⌝);
a (asm_ante_tac ⓩ¬ 3 * 3 = 9⌝ THEN rewrite_tac[]);
save_pop_thm "ZE3.6";
```

SML
```
(* ZE3.7 *)
set_goal ([], ⓩ25 ∈ {y:ℤ • y*y}⌝);
a (rewrite_tac[]);
a (z_∃_tac ⓩ5⌝);
a (rewrite_tac[]);
save_pop_thm "ZE3.7";
```

SML

```
(* ZE3.8 *)
set_goal ([], ⓩ(a × b × c) = (d × e × f) ⇒ (a × b) = (d × e) ∨ (c ∩ f) = ∅ ⌝);
a (PC_T1 "z_library_ext" rewrite_tac[]);
a (contr_tac THEN all_asm_fc_tac[]);
a (z_spec_nth_asm_tac 6 ⓩ(x1 ≜ x1, x2 ≜ x2, x3 ≜ x1′)⌝);
a (z_spec_nth_asm_tac 6 ⓩ(x1 ≜ x1, x2 ≜ x2, x3 ≜ x1′)⌝);
save_pop_thm "ZE3.8";
```

SML

```
(* ZE3.9 *)
set_goal ([], ⓩ[X,Y](∀ p: ℙ (X × Y)•
                    (∀ x:X; y:Y• (x,y) ∈ p)
              ⇔     (∀ z:X × Y• z ∈ p))⌝);
a (REPEAT strip_tac);
(* *** Goal "1" *** *)
a (z_spec_nth_asm_tac 3 ⓩ(x ≜ z.1, y ≜ z.2)⌝);

a (conv_tac (ONCE_MAP_C z_sel_t_intro_conv));

a (asm_rewrite_tac[]);
(* *** Goal "2" *** *)
a (z_spec_nth_asm_tac 3 ⓩ(x,y)⌝);
save_pop_thm "ZE3.9";
```

SML

```
(* ZE3.10 *)
set_goal ([], ⓩ[File | people = {}] = {File | people = {}}⌝);
a (PC_T "z_library_ext" z_strip_tac);
a (prove_tac[]);
a (z_∃_tac ⓩ(age ≜ x1, people ≜ x2)⌝);
a (asm_rewrite_tac[]);
save_pop_thm "ZE3.10";
```

SML

```
(* ZE3.11 *)
set_goal ([], ⓩ⟨a,b⟩ = ⟨c,d⟩ ⇒ a=c ∧ b=d⌝);
a (PC_T "z_library_ext" contr_tac);
(* *** Goal "1" *** *)
a (z_spec_nth_asm_tac 2 ⓩ(x1 ≜ 1, x2 ≜ a)⌝);
(* *** Goal "2" *** *)
a (z_spec_nth_asm_tac 2 ⓩ(x1 ≜ 2, x2 ≜ b)⌝);
save_pop_thm "ZE3.11";
```

SML
```
(* ZE3.12 *)
set_goal ([], ⓩ⟨a,b⟩ = ⟨d,e⟩ ⇒ ⟨b,d⟩ = ⟨e,a⟩⌝);
a (PC_T "z_library_ext" z_strip_tac);
a (z_spec_nth_asm_tac 1 ⓩ(x1 ≙ 1, x2 ≙ a)⌝);
a (z_spec_nth_asm_tac 2 ⓩ(x1 ≙ 2, x2 ≙ b)⌝);
a (asm_rewrite_tac[]);
save_pop_thm "ZE3.12";
```

8.2.2 Propositional Schema Calculus

These results can be solved by stripping in a manner analogous to the analogous propositional result.

SML
```
repeat drop_main_goal;
open_theory "z_exercises_2";
set_pc "z_language";
```

We illustrate the proofs by showing one example in detail.

SML
```
setlg "*2.03" SCPM2;
```

ProofPower output
```
...
(* *** Goal "" *** *)

(* ?⊢ *)  ⓩ((Pab ⇒ ¬ Qac) ⇒ Qac ⇒ ¬ Pab)⌝
...
```

In the following proof two main things are taking place.

Firstly, the logical schema operators are being transformed into the corresponding propositional logic operators, and secondly, the stripping of these follows the normal course.

In addition there is some switching taking place between schemas-as-predicates, in which there is an implicit binding membership assertion, and explicit statements about membership of bindings.

The basic proof facilities are provided for the binding membership assertions since these are more general than the schema-as-predicate format, and also are likely to arise from the latter when substitutions take place. At present the stripping facilities will revert to the schema-as-predicate format at the top level of the conclusion or assumptions if possible.

SML
```
a z_strip_tac;
```

ProofPower output
```
..
(* ?⊢ *)  ⓩ(a ≙ a, b ≙ b, c ≙ c) ∈ (Pab ⇒ ¬ Qac)
               ⇒ (a ≙ a, b ≙ b, c ≙ c) ∈ (Qac ⇒ ¬ Pab)⌝
...
```

The first step has transformed the implication to a logical implication.

SML
```
a z_strip_tac;
```

ProofPower output
```
Tactic produced 2 subgoals:

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

(*  1 *)  ⓩ¬ Qac⌝

(* ?⊢ *)  ⓩ(Qac ⇒ ¬ Pab)⌝


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

(*  1 *)  ⓩ¬ Pab⌝

(* ?⊢ *)  ⓩ(Qac ⇒ ¬ Pab)⌝
...
```

When the implication is stripped the left hand side is completely stripped into the assumptions. This results in a case split. Note here that the negation in the assumption is now a logical negation.

SML
```
a z_strip_tac;
```

ProofPower output
```
...
(*  1 *)  ⓩ¬ Pab⌝

(* ?⊢ *)  ⓩ(a ≙ a, c ≙ c) ∈ Qac ⇒ (a ≙ a, b ≙ b) ∈ (¬ Pab)⌝
...
```

SML
```
a z_strip_tac;
```

ProofPower output
```
...
(*  2 *)  ⓩ¬ Pab⌝
(*  1 *)  ⓩQac⌝

(* ?⊢ *)  ⓩ(¬ Pab)⌝
...
```

Here we are not quite finished because the negation in the assumption is a logical negation while the one in the conclusion is a schema-negation.

SML

```
a z_strip_tac;
```

ProofPower output

```
...
(*  2 *)  ⓩ¬ Pab⌝
(*  1 *)  ⓩQac⌝

(* ?⊢ *)  ⓩ¬ (a ≙ a, b ≙ b) ∈ Pab⌝
...
```

SML

```
a z_strip_tac;
```

ProofPower output

```
...
(*  2 *)  ⓩ¬ Pab⌝
(*  1 *)  ⓩQac⌝

(* ?⊢ *)  ⓩ¬ Pab⌝
...
```

Now the conclusion really is the same as the assumption.

SML

```
a z_strip_tac;
```

ProofPower output

```
Current goal achieved, next goal is:

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

(*  1 *)  ⓩ¬ Qac⌝

(* ?⊢ *)  ⓩ(Qac ⇒ ¬ Pab)⌝
...
```

The proof of this subgoal contains nothing new so we do it in one step.

SML

```
a (REPEAT z_strip_tac);
```

ProofPower output

```
Tactic produced 0 subgoals:
Current and main goal achieved
val it = () : unit
```

An alternative approach is to eliminate the schema operations first by rewriting, and then complete the proof by stripping.

SML
```
setlg "*2.03" SCPM2;
```

ProofPower output
```
...
(* *** Goal "" *** *)

(* ?⊢ *) ⓩ((Pab ⇒ ¬ Qac) ⇒ Qac ⇒ ¬ Pab)⌝
...
```

Since all the membership conversions are built in to the proof context *z_language* rewriting with no parameters suffices to eliminate the schema operators. This would not be the case if the schema expression had not been used as a predicate, since the implicit membership statement is essential to trigger the transformations in this context.

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

This yields the syntactically similar goal in which all operators are logical operators rather than schema operators.

ProofPower output
```
...
(* ?⊢ *) ⓩ(Pab ⇒ ¬ Qac) ⇒ Qac ⇒ ¬ Pab⌝
...
```

A cleaner proof is now obtained by stripping.

SML
```
a z_strip_tac;
```

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

(*  1 *) ⓩ¬ Qac⌝

(* ?⊢ *) ⓩQac ⇒ ¬ Pab⌝


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

(*  1 *) ⓩ¬ Pab⌝

(* ?⊢ *) ⓩQac ⇒ ¬ Pab⌝
...
```

We will not complete the proof, which proceeds as the previous one but with a number of steps omitted.

SML
```
drop_main_goal();
```

Many other examples are provided for you to play through if you wish. The following script demonstrates that the system can prove them all automatically.

SML
```
map prove_and_store SCPM2;
map prove_and_store SCPM3;
map prove_and_store SCPM4;
map prove_and_store SCPM5;
```

8.2.3 Schema Calculus Quantification

SML
```
open_theory "z_exercises_2";
set_pc "z_library";
```

SML
```
map prove_and_store SCPM9;
map prove_and_store SCPM10;
```

8.3 Paragraphs

8.3.1 Axiomatic Descriptions and Generics

1.

SML
```
repeat drop_main_goal;
open_theory "z_exercises_3";
set_pc "z_library";
set_flags [("z_type_check_only", false), ("z_use_axioms", true)];
```

Z
```
fun if _ then _ else _
```

2.

Z
```
[X]
      if _ then _ else _ : (𝔹 × X × X) → X
  ─────────────────
      ∀ e1,e2:X•
      if true then e1 else e2 = e1
  ∧   if false then e1 else e2 = e2
```

3.

First specialise the specification to 𝕌 to simplify using it as a rewrite.

SML

```
val if_then_else_thm = z_gen_pred_elim [⌜Z𝕌⌝]
        (z_get_spec ⌜Z(if _ then _ else _ ) ⌝);
```

ProofPower output

```
val if_then_else_thm = ⊢ (if _ then _ else _) ∈ 𝔹 × 𝕌 × 𝕌 → 𝕌
    ∧ (∀ e1, e2 : 𝕌
      • if true then e1 else e2 = e1 ∧ if false then e1 else e2 = e2) : THM
```

The required result can then be obtained directly using *rewrite_conv*:

SML

```
rewrite_conv [if_then_else_thm] ⌜Zif 2>1 then 1 else 0⌝;
```

ProofPower output

```
val it = ⊢ if 2 > 1 then 1 else 0 = 1 : THM
```

Adding *if_then_else_thm* to any rewrite will result in elimination of conditionals.

8.3.2 Consistency Proofs

Set the flags appropriately:

SML

```
repeat drop_main_goal;
open_theory "z_exercises_3";
set_pc "z_library";
set_flags [("z_type_check_only", false), ("z_use_axioms", false)];
```

Now define the required global variable:

Z

```
num:ℕ
├──────────────────────────
 4 ≤ num ≤ 50
```

Push the consistency goal, and tidy it up:

SML

```
z_push_consistency_goal ⌜Znum⌝;
```

Supply a witness:

SML

```
a (z_∃_tac ⌜Z10⌝);
```

Then complete the proof by rewriting:

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

Then save the consistency goal.

SML
```
save_consistency_thm ⓩnum⌝ (pop_thm());
```

Now set up the required goal:

SML
```
set_goal([], ⓩ num ≥ 0⌝);
```

Strip the specification of *num* into the assumptions:

SML
```
a (strip_asm_tac (z_get_spec ⓩnum⌝));
```

ProofPower output
```
...
(* *** Goal "" *** *)

(*  3 *)  ⓩ0 ≤ num⌝
(*  2 *)  ⓩ4 ≤ num⌝
(*  1 *)  ⓩnum ≤ 50⌝

(* ?⊢ *)  ⓩnum ≥ 0⌝
..
```

Then rewrite the conclusion of the goal with the assumptions.

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

ProofPower output
```
Tactic produced 0 subgoals:
Current and main goal achieved
...
```

SML
```
save_pop_thm "ZP1";
```

8.3.3 Reasoning using Schema Definitions

SML
```
repeat drop_main_goal;
open_theory "z_exercises_3";
set_pc "z_library";
set_flags [("z_type_check_only", false), ("z_use_axioms", false)];
```

8.3.3.1 Simple Pre-conditions and Refinement

Before beginning the proofs we extract the specifications for the relevant constants and bind them to an ML name:

SML
```
val specs = (map z_get_spec [ⓩOP2⌝, ⓩOP⌝, ⓩSTATE⌝]);
```

Conjecture 1:

First set the goal:

SML
```
set_goal ([], ⓩ pre OP ⇔ i? ≥ 0⌝);
```

ProofPower output
```
...
(∗ ?⊢ ∗)  ⓩ(pre OP) ⇔ i? ≥ 0⌝
...
```

Now rewrite with the specifications of *OP* and *STATE*:

SML
```
a (rewrite_tac (map z_get_spec [ⓩOP⌝, ⓩSTATE⌝]));
```

ProofPower output
```
...
(∗ ?⊢ ∗)  ⓩ(∃ r′ : 𝕌 • true) ∧ 0 ≤ i? ⇔ 0 ≤ i?⌝
...
```

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

ProofPower output
```
...
(∗  1 ∗)  ⓩ0 ≤ i?⌝

(∗ ?⊢ ∗)  ⓩ∃ r′ : 𝕌 • true⌝
```

SML
```
a (z_∃_tac ⓩx⌝);
```

ProofPower output
```
...
(∗ ?⊢ ∗)  ⓩx ∈ 𝕌 ∧ true ∧ true⌝
...
```

SML
```
a contr_tac;
save_pop_thm "ZP2";
```

ProofPower output

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

Conjecture 2:

SML

```
save_thm ("ZP3", (prove_rule specs
        ⌜Z (pre OP ⇒ pre OP2) ∧ (pre OP ∧ OP2 ⇒ OP)⌝));
```

ProofPower output

```
val it = ⊢ ((pre OP) ⇒ (pre OP2)) ∧ ((pre OP) ∧ OP2 ⇒ OP) : THM
```

8.3.3.2 The Vending Machine

SML

```
repeat drop_main_goal;
open_theory "z_exercises_3";
set_pc "z_library";
set_flags [("z_type_check_only", false), ("z_use_axioms", true)];
```

Z

```
│       price :ℕ
```

Z

```
┌─VMSTATE──────────────────────────────
│       stock, takings :ℕ
└──────────────────────────────────────
```

Z

```
┌─VM_operation─────────────────────────
│       ΔVMSTATE;
│       cash_tendered?, cash_refunded! :ℕ;
│       bars_delivered! :ℕ
└──────────────────────────────────────
```

Z

```
┌─exact_cash───────────────────────────
│       cash_tendered? :ℕ
├──────────────────
│       cash_tendered? = price
└──────────────────────────────────────
```

Z

```
┌─insufficient_cash────────────────────
│       cash_tendered? :ℕ
├──────────────────
│       cash_tendered? < price
└──────────────────────────────────────
```

```
Z
┌─some_stock──────────────────────────
│    stock : ℕ
├────────────────
│    stock > 0
└─────────────────────────────────────
```

```
Z
┌─VM_sale─────────────────────────────
│    VM_operation
├────────────────
│    stock′ = stock − 1;
│    bars_delivered! = 1;
│    cash_refunded! = cash_tendered? − price;
│    takings′ = takings + price
└─────────────────────────────────────
```

```
Z
┌─VM_nosale───────────────────────────
│    VM_operation
├────────────────
│    stock′ = stock;
│    bars_delivered! = 0;
│    cash_refunded! = cash_tendered?;
│    takings′ = takings
└─────────────────────────────────────
```

```
Z
│    VM1 ≙ exact_cash ∧ some_stock ∧ VM_sale
```

```
Z
│    VM2 ≙ insufficient_cash ∧ VM_nosale
```

```
Z
│    VM3 ≙ VM1 ∨ VM2
```

Now for convenience we bind the various specifications to ML variables:

```
SML
│val [price, VMSTATE, VM_operation, exact_cash,
│    insufficient_cash, some_stock, VM_sale,
│    VM_nosale, VM1, VM2, VM3]
│ = map z_get_spec [⌜Z price⌝,⌜Z VMSTATE⌝,⌜Z VM_operation⌝,⌜Z exact_cash⌝,
│    ⌜Z insufficient_cash⌝,⌜Z some_stock⌝,⌜Z VM_sale⌝,
│    ⌜Z VM_nosale⌝,⌜Z VM1⌝,⌜Z VM2⌝,⌜Z VM3⌝];
```

We prove various preconditions (though these are not needed for the following correctness proofs). First the pre-condition of *VM1*.

SML

```
set_goal([],ⓩpre VM1 ⇔
        (0 < stock
        ∧ cash_tendered? = price
        ∧ 0 ≤ takings)⌝);
a (rewrite_tac [VM1, VM_sale, some_stock,
 VM_operation, VMSTATE, exact_cash]);
a (pure_rewrite_tac [z_get_spec ⓩ(_ ≤ _)⌝]);
a (rewrite_tac[]);
a (REPEAT z_strip_tac);
a (z_∃_tac ⓩ(
        bars_delivered! ≙ 1,
        cash_refunded! ≙ cash_tendered? + ~ price,
        stock' ≙ stock + ~ 1,
        takings' ≙ takings + price)⌝
   THEN rewrite_tac[]);
a (PC_T1 "z_library_ext" asm_rewrite_tac
   [rewrite_rule [] price]);
a (LEMMA_T ⓩstock + ~ 1 ≤ stock⌝ asm_tac THEN1 rewrite_tac[]);
a (all_fc_tac [z_≤_trans_thm]);
a (asm_rewrite_tac []);
a (strip_asm_tac (z_get_spec ⓩprice⌝));
a (all_fc_tac [z_ℕ_plus_thm]);
val pre_VM1_thm = save_pop_thm "pre_VM1_thm";
```

Now we establish the precondition of VM2.

SML

```
set_goal([], ⓩpre VM2 ⇔
        cash_tendered? < price
        ∧ cash_tendered? ≥ 0
        ∧ stock ≥ 0
        ∧ takings ≥ 0⌝);
a (rewrite_tac [VM2, VM_nosale, VM_operation, VMSTATE, insufficient_cash]);
a (REPEAT z_strip_tac);
a (z_∃_tac ⓩ(
        bars_delivered! ≙ 0,
        cash_refunded! ≙ cash_tendered?,
        stock' ≙ stock,
        takings' ≙ takings)⌝
   THEN PC_T1 "z_library_ext" asm_rewrite_tac[]);
val pre_VM2_thm = save_pop_thm "pre_VM2_thm";
```

We next establish the precondition of VM3. The proof is simplified if the following result is established first:

SML

```
set_goal([], ⓩpre (VM1 ∨ VM2) ⇔ pre VM1 ∨ pre VM2⌝);
a (prove_tac[]);
val VM1VM2_lemma = pop_thm();
```

SML

```
set_goal([],ⓩpre VM3 ⇔
        0 < stock ∧ cash_tendered? = price ∧ 0 ≤ takings
           ∨ cash_tendered? < price
             ∧ 0 ≤ cash_tendered?
             ∧ 0 ≤ stock
             ∧ 0 ≤ takings⌝);
a (pure_rewrite_tac [VM3, VM1VM2_lemma, pre_VM1_thm, pre_VM2_thm]);
a (z_strip_tac
        THEN z_strip_tac
        THEN z_strip_tac
        THEN strip_asm_tac price
        THEN asm_rewrite_tac[]);
val pre_VM3_thm = save_pop_thm "pre_VM3_thm";
```

Now we prove that VM3 is a correct refinement of VM1.

The results about preconditions are not particularly helpful here, since the top level structure of the specification suffices to obtain the result without detailed knowledge of the preconditions.

SML

```
set_goal([], ⓩ¬ (insufficient_cash ∧ exact_cash)⌝);
a (rewrite_tac [insufficient_cash, exact_cash]);
```

ProofPower output

```
...
(∗ ?⊢ ∗)  ⓩ¬
          ((0 ≤ cash_tendered?
             ∧ cash_tendered? < price)
             ∧ 0 ≤ cash_tendered?
             ∧ cash_tendered? = price)⌝
...
```

We eliminate the < relation by rewriting with its specification. However, the specification of < contains other facts whose inverses are in the current proof context, so a simple rewrite with the specification loops. *pure_rewrite_tac* is therefore used.

SML

```
a (pure_rewrite_tac [z_get_spec ⓩ(_<_)⌝]);
```

ProofPower output

```
(∗ ?⊢ ∗)  ⓩ¬
          ((0 ≤ cash_tendered?
             ∧ 0 ≤ price + ~ (cash_tendered? + 1))
             ∧ 0 ≤ cash_tendered?
             ∧ cash_tendered? = price)⌝
```

The remaineder of the proof is a routine arithmetic manipulation.

SML
```
a (rewrite_tac [z_minus_thm, z_plus_assoc_thm1]);
a (REPEAT_N 3 z_strip_tac THEN asm_rewrite_tac[]);
val cash_lemma = save_pop_thm "cash_lemma";
```

Proving the correctness of the refinement is now straightforward.

SML
```
set_goal([], ⓩ (pre VM1 ⇒ pre VM3) ∧ (pre VM1 ∧ VM3 ⇒ VM1)⌝);
a (rewrite_tac [VM1, VM2, VM3]);
```

ProofPower output
```
(∗ ?⊢ ∗)  ⓩ((∃ bars_delivered! : 𝕌;
                 cash_refunded! : 𝕌;
                 stock′ : 𝕌;
                 takings′ : 𝕌
               • exact_cash ∧ some_stock ∧ VM_sale)
             ⇒ (∃ bars_delivered! : 𝕌;
                 cash_refunded! : 𝕌;
                 stock′ : 𝕌;
                 takings′ : 𝕌
               • exact_cash ∧ some_stock ∧ VM_sale
                 ∨ insufficient_cash ∧ VM_nosale))
          ∧ ((∃ bars_delivered! : 𝕌;
                  cash_refunded! : 𝕌;
                  stock′ : 𝕌;
                  takings′ : 𝕌
                • exact_cash ∧ some_stock ∧ VM_sale)
              ∧ (exact_cash ∧ some_stock ∧ VM_sale
                 ∨ insufficient_cash ∧ VM_nosale)
             ⇒ exact_cash ∧ some_stock ∧ VM_sale)⌝
```

SML
```
a (strip_asm_tac cash_lemma THEN asm_rewrite_tac[]);
```

ProofPower output
```
(∗  1 ∗)  ⓩ¬ insufficient_cash⌝

(∗ ?⊢ ∗)  ⓩ(∃ bars_delivered! : 𝕌;
                 cash_refunded! : 𝕌;
                 stock′ : 𝕌;
                 takings′ : 𝕌
               • exact_cash ∧ some_stock ∧ VM_sale)
             ∧ exact_cash
             ∧ some_stock
             ∧ VM_sale
          ⇒ exact_cash ∧ some_stock ∧ VM_sale⌝
```

a (REPEAT z_strip_tac); val VM3_refines_VM1 = save_pop_thm "VM3_refines_VM1";

Next we express the requirement that a vending machine does not undercharge:

Z
```
│       VM_ok : ℙ ℙ VM_operation
├──────────────────────────────────────
│       ∀ vm : ℙ VM_operation•
│       vm ∈ VM_ok ⇔
│               (∀ VM_operation • vm ⇒
│               takings′ − takings ≥ price ∗ (stock − stock′))
```

Before using this definition we convert it into an unconditional rewrite.

SML
```
│val VM_ok = z_defn_simp_rule (z_get_spec ⓩVM_ok⌝);
```

We now prove that VM3 is a VM_ok.

SML
```
│set_goal([], ⓩVM3 ∈ VM_ok⌝);
│a (rewrite_tac [VM1,VM2,VM3,VM_ok,VM_sale,VM_nosale]);
```

ProofPower output
```
│...
│(∗ ?⊢ ∗)  ⓩ(exact_cash
│                   ∧ some_stock
│                   ∧ [VM_operation
│                     | stock′ = stock + ∼ 1
│                       ∧ bars_delivered! = 1
│                       ∧ cash_refunded! = cash_tendered? + ∼ price
│                       ∧ takings′ = takings + price]
│                 ∨ insufficient_cash
│                   ∧ [VM_operation
│                     | stock′ = stock
│                       ∧ bars_delivered! = 0
│                       ∧ cash_refunded! = cash_tendered?
│                       ∧ takings′ = takings])
│                 ⊆ VM_operation
│             ∧ (∀ VM_operation
│               • exact_cash
│                     ∧ some_stock
│                     ∧ VM_operation
│                     ∧ stock′ = stock + ∼ 1
│                     ∧ bars_delivered! = 1
│                     ∧ cash_refunded! = cash_tendered? + ∼ price
│                     ∧ takings′ = takings + price
│                   ∨ insufficient_cash
|                          ∧ VM_operation
|                          ∧ stock′ = stock
|                          ∧ bars_delivered! = 0
|                          ∧ cash_refunded! = cash_tendered?
|                          ∧ takings′ = takings
|                      ⇒ price ∗ (stock + ∼ stock′) ≤ takings′ + ∼ takings)⌝
|...
```

There are a lot of propositional logic (or related schema calculus) operators here which can be simplified by stripping. The subset sign will need to be treated extensionally, so proof context *z_library_ext* is probably appropriate. It is also clear that several equations will arise in the assumptions, and therefore likely that rewriting with the assumptions will be a good idea, so:

SML
```
|a (PC_T "z_library_ext" (REPEAT z_strip_tac) THEN asm_rewrite_tac[]);
```

Which considerably simplified the problem:

ProofPower output
```
|...
|(∗  7 ∗)  ⌜Z VM_operation⌝
|(∗  6 ∗)  ⌜Z exact_cash⌝
|(∗  5 ∗)  ⌜Z some_stock⌝
|(∗  4 ∗)  ⌜Z stock′ = stock + ∼ 1⌝
|(∗  3 ∗)  ⌜Z bars_delivered! = 1⌝
|(∗  2 ∗)  ⌜Z cash_refunded! = cash_tendered? + ∼ price⌝
|(∗  1 ∗)  ⌜Z takings′ = takings + price⌝
|
|(∗ ?⊢ ∗)  ⌜Z price ∗ (stock + ∼ (stock + ∼ 1)) ≤ (takings + price) + ∼ takings⌝
|...
```

To solve this little arithmetic problem we move ⌜Z∼ *takings*⌝ left to place it next to *takings*:

SML
```
|a (rewrite_tac [z_∀_elim ⌜Z∼ takings⌝ z_plus_order_thm]);
```

ProofPower output
```
|...
|(∗ ?⊢ ∗)  ⌜Z price ∗ (stock + ∼ (stock + ∼ 1)) ≤ ∼ takings + takings + price⌝
|...
```

Pushing in the minus sign and associating the additions to the left will result in the goal being proved using the cancellation results built into our current proof context.

SML
```
|a (rewrite_tac [z_minus_thm, z_plus_assoc_thm1]);
|val VM3_ok_thm = save_pop_thm "VM3_ok_thm";
```

8.4 The Z ToolKit

8.4.1 Sets

All of the examples in this theory can be proven automatically by the system.

First we set up an appropriate context:

SML
```
repeat drop_main_goal;
open_theory "z_exercises_4";
set_pc "z_sets_ext";
```

8.4.1.1 Results Provable by Stripping

We display one case partly expanded out:

SML
```
set_goal([],⌜Z a ∩ (b \ c) = a ∩ b \ c⌝);
a z_strip_tac;
```

ProofPower output
```
(* ?⊢ *)  ⌜Z ∀ x1 : 𝕌 • x1 ∈ a ∩ (b \ c) ⇔ x1 ∈ a ∩ b \ c⌝
```

SML
```
a z_strip_tac;
```

ProofPower output
```
(* ?⊢ *)  ⌜Z x1 ∈ 𝕌 ∧ true ⇒ (x1 ∈ a ∩ (b \ c) ⇔ x1 ∈ a ∩ b \ c)⌝
```

continuing only using *z_strip_tac* (but omitting the display of this) as follows:

ProofPower output
```
(* ?⊢ *)  ⌜Z x1 ∈ a ∩ (b \ c) ⇔ x1 ∈ a ∩ b \ c⌝
```

ProofPower output
```
(* ?⊢ *)  ⌜Z (x1 ∈ a ∩ (b \ c) ⇒ x1 ∈ a ∩ b \ c)
                ∧ (x1 ∈ a ∩ b \ c ⇒ x1 ∈ a ∩ (b \ c))⌝
```

ProofPower output
```
(* *** Goal "2" *** *)
(* ?⊢ *)  ⌜Z x1 ∈ a ∩ b \ c ⇒ x1 ∈ a ∩ (b \ c)⌝

(* *** Goal "1" *** *)
(* ?⊢ *)  ⌜Z x1 ∈ a ∩ (b \ c) ⇒ x1 ∈ a ∩ b \ c⌝
```

ProofPower output
```
(* *** Goal "1" *** *)
(*  3 *)  ⓩx1 ∈ a⌝
(*  2 *)  ⓩx1 ∈ b⌝
(*  1 *)  ⓩ¬ x1 ∈ c⌝

(* ?⊢ *)  ⓩx1 ∈ a ∩ b \ c⌝
```

ProofPower output
```
...
(* ?⊢ *)  ⓩx1 ∈ a ∩ b ∧ x1 ∉ c⌝
```

ProofPower output
```
(* *** Goal "1.2" *** *)
(*  3 *)  ⓩx1 ∈ a⌝
(*  2 *)  ⓩx1 ∈ b⌝
(*  1 *)  ⓩ¬ x1 ∈ c⌝

(* ?⊢ *)  ⓩx1 ∉ c⌝

(* *** Goal "1.1" *** *)
(*  3 *)  ⓩx1 ∈ a⌝
(*  2 *)  ⓩx1 ∈ b⌝
(*  1 *)  ⓩ¬ x1 ∈ c⌝

(* ?⊢ *)  ⓩx1 ∈ a ∩ b⌝
```

ProofPower output
```
(* *** Goal "1.1" *** *)
(*  3 *)  ⓩx1 ∈ a⌝
(*  2 *)  ⓩx1 ∈ b⌝
(*  1 *)  ⓩ¬ x1 ∈ c⌝

(* ?⊢ *)  ⓩx1 ∈ a ∧ x1 ∈ b⌝
```

ProofPower output
```
(* *** Goal "1.1.2" *** *)
(*  3 *)  ⓩx1 ∈ a⌝
(*  2 *)  ⓩx1 ∈ b⌝
(*  1 *)  ⓩ¬ x1 ∈ c⌝

(* ?⊢ *)  ⓩx1 ∈ b⌝

(* *** Goal "1.1.1" *** *)
(*  3 *)  ⓩx1 ∈ a⌝
(*  2 *) ⓩx1 ∈ b⌝
(*  1 *) ⓩ¬ x1 ∈ c⌝

(* ?⊢ *) ⓩx1 ∈ a⌝
```

ProofPower output

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

(* *** Goal "1.1.2" *** *)
...
```

ProofPower output

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

(* *** Goal "1.2" *** *)
...
```

ProofPower output

```
...
(* ?⊢ *) ⓩ¬ x1 ∈ c⌝
```

ProofPower output

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

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

(* ?⊢ *) ⓩx1 ∈ a ∩ b \ c ⇒ x1 ∈ a ∩ (b \ c)⌝
...
```

Goal 2 being similar to goal 1 we complete its proof in one step:

SML

```
a (REPEAT z_strip_tac);
```

ProofPower output

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

The following groups of exercises are provable in exactly the same manner.

SML

```
map prove_and_store Z1;
map prove_and_store Z2;
map prove_and_store Z3;
map prove_and_store Z3b;
```

8.4.2 Relations

SML
```
repeat drop_main_goal;
open_theory "z_exercises_4";
set_pc "z_rel_ext";
```

The following simple example shows how stripping followed by forward chaining often suffices for proofs in this theory.

SML
```
set_goal([], ⓩP ⨾ Q ⨾ R = (P ⨾ Q) ⨾ R⌝);
a contr_tac;
```

ProofPower output
```
(∗ ∗∗∗ Goal "2" ∗∗∗ ∗)

(∗  4 ∗)  ⓩ(x1, y′) ∈ P⌝
(∗  3 ∗)  ⓩ(y′, y) ∈ Q⌝
(∗  2 ∗)  ⓩ(y, x2) ∈ R⌝
(∗  1 ∗)  ⓩ∀ y : 𝕌 • ¬ ((x1, y) ∈ P ∧ (y, x2) ∈ Q ⨾ R)⌝

(∗ ?⊢ ∗)  ⓩfalse⌝


(∗ ∗∗∗ Goal "1" ∗∗∗ ∗)

(∗  4 ∗)  ⓩ(x1, y) ∈ P⌝
(∗  3 ∗)  ⓩ(y, y′) ∈ Q⌝
(∗  2 ∗)  ⓩ(y′, x2) ∈ R⌝
(∗  1 ∗)  ⓩ∀ y : 𝕌 • ¬ ((x1, y) ∈ P ⨾ Q ∧ (y, x2) ∈ R)⌝

(∗ ?⊢ ∗)  ⓩfalse⌝
```

The "implications" in the assumptions of these subgoals are well buried, but are nevertheless uncovered by the forward chaining facilities.

SML
```
a (all_asm_fc_tac[]);
```

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

SML
```
a (all_asm_fc_tac[]);
pop_thm();
```

ProofPower output
```
Tactic produced 0 subgoals:
Current and main goal achieved
...
val it = ⊢ P ⨾ Q ⨾ R = (P ⨾ Q) ⨾ R : THM
...
```

Many of the exercises are therefore proven automatically as follows.

SML
```
map prove_and_store Z4;
map prove_and_store Z5;
```

SML
```
map prove_and_store Z5b;
```

SML
```
(* "Z5c.1" *)
set_goal([], ⓩ[X,Y,Z] (∀x:𝕌;g:𝕌• g ∈ X ⇸ Y ∧ x ∈ dom g ⇒ (x, g x) ∈ g)⌝);
a (REPEAT z_strip_tac);
a (POP_ASM_T (PC_T1 "z_library_ext" strip_asm_tac));
a (all_fc_tac [z_fun_app_clauses]);
a (asm_rewrite_tac[]);
val Z5c1 = save_pop_thm "Z5c.1";
```

The following proof make use of the previous result (Z5c1).

SML
```
(* "Z5c.2" *)
set_goal([], ⓩ[X,Y,Z] x ∈ dom g ∧ g x ∈ dom f ∧ g ∈ X ⇸ Y
        ∧ f ∈ Y ⇸ Z ⇒ (f ∘ g)(x) = f(g(x))⌝);
set_pc "z_library";
a (REPEAT z_strip_tac);
a(z_app_eq_tac);
a (PC_T1 "z_library_ext" rewrite_tac[]);
a (REPEAT z_strip_tac);
a (lemma_tac ⓩg x = y⌝);
(* *** Goal "1.1" *** *)
a (all_fc_tac [z_fun_app_clauses]);
(* *** Goal "1.2" *** *)
a (all_fc_tac [z_fun_app_clauses]);
a (asm_rewrite_tac[]);
(* *** Goal "2" *** *)
a (z_∃_tac ⓩg x⌝);
a (REPEAT z_strip_tac);
(* *** Goal "2.1" *** *)
a (all_fc_tac [Z5c1]);
```

```
(* *** Goal "2.2" *** *)
a (all_fc_tac [Z5c1]);
save_pop_thm("Z5c.2");
```

SML

```
set_pc "z_rel_ext";
map prove_and_store Z6;
map prove_and_store Z6b;
map prove_and_store Z7;
map prove_and_store Z8;
map prove_and_store Z8b;
map prove_and_store Z9;
```

SML

```
setlg "Z9b.1" Z9b;
a(prove_tac[z_∈_first_thm]);
(* *** Goal "1" *** *)
a(contr_tac);
a(z_spec_nth_asm_tac 1 ⓩ(x ≙ (x1, y))⌝);
a(swap_nth_asm_concl_tac 1);
a(rewrite_tac[]);
(* *** Goal "2" *** *)
a(contr_tac);
a(z_spec_nth_asm_tac 1 ⓩ(y ≙ (x.2))⌝);
a(all_var_elim_asm_tac1);
a(swap_nth_asm_concl_tac 1);
a(conv_tac(ONCE_MAP_C z_tuple_intro_conv));
a(asm_rewrite_tac[]);
save_pop_thm "Z9b.1";
```

SML

```
setlg "Z9b.2" Z9b;
a(prove_tac[z_∈_second_thm]);
(* *** Goal "1" *** *)
a(contr_tac);
a(z_spec_nth_asm_tac 1 ⓩ(x ≙ (x, x1))⌝);
a(swap_nth_asm_concl_tac 1);
a(rewrite_tac[]);
(* *** Goal "2" *** *)
a(contr_tac);
a(z_spec_nth_asm_tac 1 ⓩ(x ≙ (x.1))⌝);
a(all_var_elim_asm_tac1);
a(swap_nth_asm_concl_tac 1);
a(conv_tac(ONCE_MAP_C z_tuple_intro_conv));
a (asm_rewrite_tac[]);
save_pop_thm "Z9b.2";
```

SML

```
map prove_and_store Z10;
```

SML

```
setlg "Z10b.1" Z10b;
a(contr_tac);
(* *** Goal "1" *** *)
a(z_spec_nth_asm_tac 1 ⓩ(y ≙ y)⌝);
(* *** Goal "2" *** *)
a(z_spec_nth_asm_tac 1 ⓩ(y ≙ y)⌝);
(* *** Goal "3" *** *)
a(z_spec_nth_asm_tac 1 ⓩ(y ≙ y)⌝);
a(z_spec_nth_asm_tac 3 ⓩ(y ≙ y')⌝);
(* *** Goal "4" *** *)
a(z_spec_nth_asm_tac 1 ⓩ(y ≙ y)⌝);
save_pop_thm "Z10b.1";
```

SML

```
set_pc "z_library_ext";
setlg "Z10b.2" Z10b;
a (REPEAT strip_tac);
a(z_app_eq_tac);
a(z_spec_nth_asm_tac 6 ⓩ(x1 ≙ x,x2 ≙ y)⌝);
a(REPEAT strip_tac);
(* *** Goal "1" *** *)
a(z_spec_nth_asm_tac 9 ⓩ(x1 ≙ x,x2 ≙ f_a)⌝);
a(z_app_eq_tac);
a(REPEAT strip_tac);
a(z_spec_nth_asm_tac 11 ⓩ(x1 ≙ x,x2 ≙ f_a')⌝);
a(z_spec_nth_asm_tac 11 ⓩ(x ≙ x, y1 ≙ f_a', y2 ≙ f_a)⌝);
(* *** Goal "2" *** *)
a(z_spec_nth_asm_tac 4 ⓩ(y ≙ f_a)⌝);
(* *** Goal "3" *** *)
a(z_spec_nth_asm_tac 4 ⓩ(y ≙ y')⌝);
(* *** Goal "4" *** *)
a(lemma_tac ⓩf x = y⌝);
(* *** Goal "4.1" *** *)
a(z_app_eq_tac);
a(REPEAT strip_tac);
a(z_spec_nth_asm_tac 9 ⓩ(x ≙ x, y1 ≙ f_a, y2 ≙ y)⌝);
a(z_spec_nth_asm_tac 11 ⓩ(x1 ≙ x,x2 ≙ f_a)⌝);
(* *** Goal "4.2" *** *)
a(swap_nth_asm_concl_tac 2);
a(asm_rewrite_tac[]);
save_pop_thm "Z10b.2";
```

SML

```
setlg "Z10b.3" Z10b;
a(contr_tac);
a(swap_nth_asm_concl_tac 1);
a(z_app_eq_tac);
a(contr_tac);
(* *** Goal "1" *** *)
a(z_spec_nth_asm_tac 3 ⓩ(y ≙ y)⌝);
(* *** Goal "2" *** *)
a(swap_nth_asm_concl_tac 1);
a(z_app_eq_tac);
a(contr_tac);
a(z_spec_nth_asm_tac 6 ⓩ(x1 ≙ x,x2 ≙ f_a)⌝);
a(z_spec_nth_asm_tac 8 ⓩ(x1 ≙ x,x2 ≙ f_a')⌝);
a(z_spec_nth_asm_tac 8 ⓩ(x ≙ x, y1 ≙ f_a', y2 ≙ f_a)⌝);
(* *** Goal "3" *** *)
```

```
a(z_spec_nth_asm_tac 5 ⌜(x1 ≙ x,x2 ≙ y)⌝);
a(lemma_tac ⌜g x = y⌝);
(* *** Goal "3.1" *** *)
a(z_app_eq_tac);
a(contr_tac);
a(z_spec_nth_asm_tac 9 ⌜(x1 ≙ x,x2 ≙ f_a)⌝);
a(z_spec_nth_asm_tac 9 ⌜(x ≙ x, y1 ≙ f_a, y2 ≙ y)⌝);
(* *** Goal "3.2" *** *)
a(swap_nth_asm_concl_tac 4);
a(asm_rewrite_tac[]);
(* *** Goal "4" *** *)
a(z_spec_nth_asm_tac 5 ⌜(x1 ≙ x,x2 ≙ y)⌝);
a(lemma_tac ⌜g x = y⌝);
(* *** Goal "4.1" *** *)
a(z_app_eq_tac);
a(contr_tac);
a(z_spec_nth_asm_tac 9 ⌜(x1 ≙ x,x2 ≙ f_a)⌝);
a(z_spec_nth_asm_tac 9 ⌜(x ≙ x, y1 ≙ f_a, y2 ≙ y)⌝);
(* *** Goal "4.2" *** *)
a(swap_nth_asm_concl_tac 4);
a(asm_rewrite_tac[]);
save_pop_thm "Z10b.3";
```

SML

```
map prove_and_store Z11;
```

SML

```
setlg "Z11b.1" Z11b;
a contr_tac;
a(z_spec_nth_asm_tac 2 ⌜(x1 ≙ x1, x2 ≙ x2)⌝);
a(z_spec_nth_asm_tac 6 ⌜(S ≙ S)⌝);
(* *** Goal "1" *** *)
a(asm_fc_tac[]);
(* *** Goal "2" *** *)
a(z_spec_nth_asm_tac 6 ⌜(x1 ≙ x1', x2 ≙ x2')⌝);
a(z_spec_nth_asm_tac 1 ⌜(y ≙ y')⌝);
(* *** Goal "3" *** *)
a(z_spec_nth_asm_tac 6 ⌜(S ≙ S)⌝);
(* *** Goal "3.1" *** *)
a(asm_fc_tac[]);
(* *** Goal "3.2" *** *)
a(z_spec_nth_asm_tac 7 ⌜(x1 ≙ x1', x2 ≙ x2')⌝);
a(z_spec_nth_asm_tac 1 ⌜(y ≙ y')⌝);
(* *** Goal "3.3" *** *)
a(z_spec_nth_asm_tac 3 ⌜(y ≙ y)⌝);
save_pop_thm "Z11b.1";
```

8.4.3 Functions

SML
```
repeat drop_main_goal;
open_theory "z_exercises_4";
set_pc "z_fun_ext";
```

SML
```
setlg "Z12.1" Z12;
a (rewrite_tac[] THEN REPEAT strip_tac);
a (z_app_eq_tac THEN REPEAT strip_tac);
a (all_asm_fc_tac[]);
a (all_asm_fc_tac[]);
save_pop_thm "Z12.1";
```

SML
```
setlg "Z12.2" Z12;
a (rewrite_tac[] THEN REPEAT strip_tac);
a (z_app_eq_tac THEN REPEAT strip_tac);
a (all_asm_fc_tac[]);
a (all_asm_fc_tac[]);
save_pop_thm "Z12.2";
```

SML
```
setlg "Z12.3" Z12;
a (rewrite_tac[] THEN REPEAT strip_tac);
a (z_app_eq_tac THEN REPEAT strip_tac);
a (all_asm_fc_tac[]);
a (all_asm_fc_tac[]);
save_pop_thm "Z12.3";
```

SML
```
setlg "Z12.4" Z12;
a (rewrite_tac[] THEN REPEAT strip_tac);
a (z_app_eq_tac THEN REPEAT strip_tac);
a (all_asm_fc_tac[]);
a (all_asm_fc_tac[]);
save_pop_thm "Z12.4";
```

SML
```
setlg "Z12.5" Z12;
a (rewrite_tac[] THEN REPEAT strip_tac);
a (z_app_eq_tac THEN REPEAT strip_tac);
a (all_asm_fc_tac[]);
a (all_asm_fc_tac[]);
save_pop_thm "Z12.5";
```

SML
```
setlg "Z12.6" Z12;
a (rewrite_tac[] THEN REPEAT strip_tac);
a (z_app_eq_tac THEN REPEAT strip_tac);
a (all_asm_fc_tac[]);
a (all_asm_fc_tac[]);
save_pop_thm "Z12.6";
```

SML
```
setlg "Z12a.1" Z12a;
a (EVERY[
        rewrite_tac[],
        REPEAT strip_tac,
        z_app_eq_tac,
        REPEAT strip_tac,
        all_asm_fc_tac[],
        all_asm_fc_tac[]]);
save_pop_thm "Z12a.1";
```

SML
```
setlg "Z12b.1" Z12b;
a(contr_tac THEN all_asm_fc_tac[]);
a(all_asm_fc_tac[]);
a(z_spec_nth_asm_tac 3 ⓩ(y ≙ x)⌝);
a(swap_nth_asm_concl_tac 6);
a(asm_rewrite_tac[]);
save_pop_thm "z12b.1";
```

SML
```
setlg "Z12b.2" Z12b;
a (contr_tac THEN all_asm_fc_tac[]);
(* *** Goal "1" *** *)
a(z_spec_nth_asm_tac 7 ⓩ(x1 ≙ y1, x2 ≙ y2)⌝);
(* *** Goal "1.1" *** *)
a(z_spec_nth_asm_tac 1 ⓩ(y ≙ x)⌝);
(* *** Goal "1.2" *** *)
a(z_spec_nth_asm_tac 1 ⓩ(y ≙ x)⌝);
(* *** Goal "1.3" *** *)
a(swap_nth_asm_concl_tac 1);
a(ALL_ASM_FC_T rewrite_tac [get_thm "−" "Z12a.1"]);
(* *** Goal "2" *** *)
a(lemma_tac ⓩy = f x1⌝);
(* *** Goal "2.1" *** *)
a(z_app_eq_tac);
a(contr_tac);
```

```
a(z_spec_nth_asm_tac 14 ⌜Z(x1 ≘ x1, x2 ≘ f_a)⌝);
a(z_spec_nth_asm_tac 14 ⌜Z(x ≘ x1, y1 ≘ f_a, y2 ≘ y)⌝);
(* *** Goal "2.2" *** *)
a(swap_nth_asm_concl_tac 9);
a(asm_rewrite_tac[]);
a(contr_tac);
a(z_spec_nth_asm_tac 13 ⌜Z(x1 ≘ x1, x2 ≘ f x2)⌝);
a(z_spec_nth_asm_tac 11 ⌜Z(x ≘ ⌜Zf x2⌝, y1 ≘ x1, y2 ≘ x2)⌝);
a(lemma_tac ⌜Zy' = f x2⌝);
(* *** Goal "2.2.1" *** *)
a(z_app_eq_tac);
a(contr_tac);
a(z_spec_nth_asm_tac 17 ⌜Z(x1 ≘ x2, x2 ≘ f_a)⌝);
a(z_spec_nth_asm_tac 17 ⌜Z(x ≘ x2, y1 ≘ f_a, y2 ≘ y')⌝);
(* *** Goal "2.2.2" *** *)
a(swap_nth_asm_concl_tac 12);
a(asm_rewrite_tac[]);
save_pop_thm "Z12b.2";
```

No solutions for Z12.b3 to Z12b.7.

8.4.4 Numbers and Finiteness

SML
```
repeat drop_main_goal;
open_theory "z_exercises_4";
set_pc "z_library";
set_flags [("z_type_check_only", false), ("z_use_axioms", true)];
```

Z
```
│    Σ : ℕ → ℕ
├──────────────────────────────
│    ∀n:ℕ•
│    Σ 0 = 0
│∧   Σ (n+1) = (n + 1) + Σ n
```

The second problem:

SML
```
set_goal([],⌜Z ∀n: ℕ• (Σ n) * 2 = n * (n+1)⌝);
```

ProofPower output
```
...
(* *** Goal "" *** *)

(* ?⊢ *)  ⌜Z∀ n : ℕ • Σ n * 2 = n * (n + 1)⌝
...
```

SML
```
a (strip_asm_tac (z_get_spec ⓩ(Σ)⌝));
```

ProofPower output
```
...
(*  2 *)  ⓩΣ ∈ ℕ → ℕ⌝
(*  1 *)  ⓩ∀ n : ℕ • Σ 0 = 0 ∧ Σ (n + 1) = (n + 1) + Σ n⌝

(* ?⊢ *)  ⓩ∀ n : ℕ • Σ n * 2 = n * (n + 1)⌝
...
```

SML
```
a (z_strip_tac THEN PC_T1 "z_language" rewrite_tac[]);
```

ProofPower output
```
...
(*  2 *)  ⓩΣ ∈ ℕ → ℕ⌝
(*  1 *)  ⓩ∀ n : ℕ • Σ 0 = 0 ∧ Σ (n + 1) = (n + 1) + Σ n⌝

(* ?⊢ *)  ⓩn ∈ ℕ ⇒ Σ n * 2 = n * (n + 1)⌝
...
```

SML
```
a z_ℕ_induction_tac;
```

ProofPower output
```
...
(* *** Goal "2" *** *)

(*  4 *)  ⓩΣ ∈ ℕ → ℕ⌝
(*  3 *)  ⓩ∀ n : ℕ • Σ 0 = 0 ∧ Σ (n + 1) = (n + 1) + Σ n⌝
(*  2 *)  ⓩ0 ≤ i⌝
(*  1 *)  ⓩΣ i * 2 = i * (i + 1)⌝

(* ?⊢ *)  ⓩΣ (i + 1) * 2 = (i + 1) * ((i + 1) + 1)⌝


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

(*  2 *)  ⓩΣ ∈ ℕ → ℕ⌝
(*  1 *)  ⓩ∀ n : ℕ • Σ 0 = 0 ∧ Σ (n + 1) = (n + 1) + Σ n⌝

(* ?⊢ *)  ⓩΣ 0 * 2 = 0 * (0 + 1)⌝
```

SML

```
(* *** Goal "1" *** *)
a (z_spec_nth_asm_tac 1 ⓩ0⌝
        THEN asm_rewrite_tac[]);
```

ProofPower output

```
...
Current goal achieved, next goal is:
...
```

SML

```
(* *** Goal "2" *** *)
a (all_asm_fc_tac[]);
```

ProofPower output

```
...
(*  6 *)  ⓩΣ ∈ ℕ → ℕ⌝
(*  5 *)  ⓩ∀ n : ℕ • Σ 0 = 0 ∧ Σ (n + 1) = (n + 1) + Σ n⌝
(*  4 *)  ⓩ0 ≤ i⌝
(*  3 *)  ⓩΣ i * 2 = i * (i + 1)⌝
(*  2 *)  ⓩΣ 0 = 0⌝
(*  1 *)  ⓩΣ (i + 1) = (i + 1) + Σ i⌝

(* ?⊢ *)  ⓩΣ (i + 1) * 2 = (i + 1) * ((i + 1) + 1)⌝
```

SML

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

ProofPower output

```
...
(* ?⊢ *)  ⓩ((i + 1) + Σ i) * 2 = (i + 1) * ((i + 1) + 1)⌝
...
```

SML

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

ProofPower output

```
...
(* ?⊢ *)  ⓩ(i * 2 + 2) + i * i + i = ((i * i + i) + i + 1) + i + 1⌝
...
```

SML

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

ProofPower output
```
...
(* ?⊢ *) ⓩ(i * 2 + 2) + i = (i + i + 1) + i + 1⌝
...
```

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

ProofPower output
```
...
(* ?⊢ *) ⓩi * 2 + 2 = i + i + 2⌝
...
```

SML
```
a (rewrite_tac[z_plus_assoc_thm1]);
```

ProofPower output
```
...
(* ?⊢ *) ⓩi * 2 = i + i⌝
```

SML
```
a (pure_rewrite_tac
        [prove_rule []ⓩ2 = 1 + 1⌝,
        z_times_plus_distrib_thm]);
```

ProofPower output
```
...
(* ?⊢ *) ⓩi * 1 + i * 1 = i + i⌝
```

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

ProofPower output
```
...
Current and main goal achieved
```

SML
```
save_pop_thm "ZNum.1";
```

The solution to the third problem of this section is:

SML
```
repeat drop_main_goal;
open_theory "z_exercises_4";
set_pc "z_library_ext";
set_flags [("z_type_check_only", false), ("z_use_axioms", true)];

setlg "ZNum.2" ZNum;
```

ProofPower output

```
(∗ ?⊢ ∗)  ⓩ∀ x, y : ℤ • x ≤ y ⇒ 0 .. x ⊆ 0 .. y⌝
```

First expand ...

SML

```
a(rewrite_tac[z_get_spec ⓩ(_.._)⌝] THEN REPEAT strip_tac);
```

ProofPower output

```
...
(∗  3 ∗)  ⓩx ≤ y⌝
(∗  2 ∗)  ⓩ0 ≤ x1⌝
(∗  1 ∗)  ⓩx1 ≤ x⌝

(∗ ?⊢ ∗)  ⓩx1 ≤ y⌝
...
```

Then forward chain using transitivity of ≤.

SML

```
a(all_fc_tac[z_≤_trans_thm]);
```

ProofPower output

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

SML

```
save_pop_thm "ZNum.2";
```

The solution to the fourth problem of this section is:

SML

```
setlg "ZNum.3" ZNum;
```

ProofPower output

```
...
(∗ ?⊢ ∗)  ⓩ∀ x, y : ℤ • ¬ x ≤ y ⇒ 0 .. y ⊆ 0 .. x − 1⌝
...
```

First expand the definition of ...

SML

```
a(rewrite_tac[z_get_spec ⓩ(_.._)⌝] THEN REPEAT strip_tac);
```

ProofPower output

```
...
(∗  3 ∗)  ⓩy < x⌝
(∗  2 ∗)  ⓩ0 ≤ x1⌝
(∗  1 ∗)  ⓩx1 ≤ y⌝

(∗ ?⊢ ∗)  ⓩx1 ≤ x + ∼ 1⌝
...
```

Now forward chain on the assumptions using *z_≤_less_trans_thm* to obtain *x1* < *x*.

SML
```
a(all_fc_tac[z_≤_less_trans_thm]);
```

ProofPower output
```
...
(*  1 *)  ⌜x1 < x⌝

(* ?⊢ *)  ⌜x1 ≤ x + ~ 1⌝
...
```

Now it is necessary to expand the definition of < in the last assumption. *POP_ASM_T* takes out the last assumption and feeds it into the *THM_TACTIC* supplied to it. In this case we rewrite the assumption with the specification of < before passing it to *ante_tac*, which inserts in into the conclusion of the goal as the *ante*cedent of a new implication.

SML
```
a(POP_ASM_T (ante_tac o pure_once_rewrite_rule[z_get_spec⌜(_<_)⌝]));
```

ProofPower output
```
...
(* ?⊢ *)  ⌜x1 + 1 ≤ x ⇒ x1 ≤ x + ~ 1⌝
...
```

In the absence of support for linear arithmetic this obvious result must be proven by transforming the conclusion of the goal until the various built in cancellation laws apply. First we move everything to the left hand side of the inequalities using *z_≤_≤_0_thm*.

SML
```
a(once_rewrite_tac[z_≤_≤_0_thm]);
```

ProofPower output
```
...
(* ?⊢ *)  ⌜(x1 + 1) + ~ x ≤ 0 ⇒ x1 + ~ (x + ~ 1) ≤ 0⌝
...
```

Now we use *z_plus_order_thm* to reorder the arithmetic expressions and *z_minus_thm* to provide some cancellation results which have been omitted from the proof context.

SML
```
a(rewrite_tac[z_∀_elim ⌜~ x⌝ z_plus_order_thm, z_minus_thm]);
```

ProofPower output
```
...
Current and main goal achieved
...
```

SML
```
save_pop_thm "ZNum.3";
```

REFERENCES

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

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

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

[4] J.M. Spivey. *The Z Notation: A Reference Manual, Second Edition.* Prentice-Hall, 1992.

[5] DS/FMU/IED/USR001. *ProofPower Document Preparation.* Lemma 1 Ltd.

[6] DS/FMU/IED/USR004. *ProofPower Tutorial Manual.* Lemma 1 Ltd., `http://www.lemma-one.com`.

[7] DS/FMU/IED/USR005. *ProofPower Description Manual.* Lemma 1 Ltd., `http://www.lemma-one.com`.

[8] DS/FMU/IED/USR013. *ProofPower HOL Tutorial Notes.* Lemma 1 Ltd., `http://www.lemma-one.com`.

[9] LEMMA1/HOL/USR029. *ProofPower HOL Reference Manual.* Lemma 1 Ltd., `rda@lemma-one.com`.

[10] ZIP/PRG/92/121. *Z Base Standard (version 1.0).* Z Standards Change Group, Oxford University Computing Laboratory, 30th November 1992.

INDEX