The Z/EVES Reference Manual (for Version 1.5)

TR-97-5493-03d

Irwin Meisels and Mark Saaltink

Release date: December 1995
Latest revision date: September 1997

ORA Canada
P.O. Box 46005, 2339 Ogilvie Rd.
Ottawa, Ontario K1J 9M7
CANADA

Contents

1 Introduction **1**

2 Basic Concepts **3**
- 2.1 The Z/EVES User Interface . . . 3
  - 2.1.1 Entering Specifications . . . 3
  - 2.1.2 Entering Commands . . . 3
  - 2.1.3 Recovery From Errors . . . 3
- 2.2 Proving . . . 4
  - 2.2.1 Implicit Declarations . . . 4
  - 2.2.2 Type Information . . . 4
- 2.3 Domain Checking . . . 4

3 The Z/LaTeX Language **7**
- 3.1 Lexical Structure . . . 7
  - 3.1.1 Words . . . 7
  - 3.1.2 Numerals . . . 8
  - 3.1.3 Strings . . . 8
  - 3.1.4 Newlines, Spaces, and Ignored Text . . . 8
  - 3.1.5 Equivalents . . . 8
  - 3.1.6 *fuzz* Directives . . . 9
- 3.2 Syntax Notation and Lexical Classes . . . 9
- 3.3 Specifications . . . 10
- 3.4 Z Sections . . . 10
  - 3.4.1 Creating Z Sections . . . 11
  - 3.4.2 Z Section Parents . . . 11
    - 3.4.2.1 Consistency Checks . . . 12
- 3.5 Paragraphs . . . 12
  - 3.5.1 Zed Boxes . . . 13
  - 3.5.2 Given Set Definitions . . . 13
  - 3.5.3 Schema Definitions . . . 13
  - 3.5.4 Abbreviation Definitions . . . 14
  - 3.5.5 Free Type Definitions . . . 15
  - 3.5.6 Labelled Predicates . . . 16
  - 3.5.7 Schema Boxes . . . 16
  - 3.5.8 Axiomatic Boxes . . . 17
  - 3.5.9 Generic Boxes . . . 18
  - 3.5.10 Theorems . . . 18
    - 3.5.10.1 Facts . . . 19
    - 3.5.10.2 Rewrite Rules . . . 19

3.5.10.3 Forward Rules . . . 19
3.5.10.4 Assumption Rules . . . 20
3.5.11 Syntax Declarations . . . 20
3.5.12 Input Commands . . . 21
3.6 Declarations and Scope . . . 21
3.7 Schema Expressions . . . 22
3.7.1 Schema Quantifications . . . 23
3.7.2 Schema Texts . . . 23
3.7.3 Schema References . . . 23
3.7.4 Schema Negation . . . 24
3.7.5 Schema Precondition . . . 24
3.7.6 Binary Logical Schema Operators . . . 25
3.7.7 Schema Projection and Hiding . . . 25
3.7.8 Schema Compositions . . . 25
3.8 Predicates . . . 26
3.8.1 Quantifications . . . 26
3.8.2 Local Variable Binding . . . 27
3.8.3 Conditional Predicates . . . 27
3.8.4 Binary Relations . . . 27
3.8.5 Unary Relations . . . 28
3.8.6 Schema References . . . 28
3.8.7 Logical Connectives . . . 28
3.8.8 Constants . . . 28
3.9 Expressions . . . 29
3.9.1 Lambda Expressions . . . 30
3.9.2 Definite Descriptions . . . 30
3.9.3 Local Variable Binding . . . 30
3.9.4 Conditional Expressions . . . 31
3.9.5 Cross Products . . . 31
3.9.6 Function Applications . . . 31
3.9.7 Variable References . . . 32
3.9.8 Constants . . . 32
3.9.9 Schema References . . . 32
3.9.10 Set Comprehensions . . . 33
3.9.11 Set Displays . . . 33
3.9.12 Sequence Displays . . . 33
3.9.13 Bag Displays . . . 34
3.9.14 Binding Set Displays . . . 34
3.9.15 Tuples . . . 34
3.9.16 Binding Formation . . . 35
3.9.17 Component Selection . . . 35
3.10 Identifiers . . . 35
3.10.1 Variable and Declaration Names . . . 35
3.10.2 Operator Names . . . 36
3.10.3 Identifiers . . . 36

4 Z/EVES Commands 37
4.1 Z Section Proofs . . . 37
4.1.1 Declare . . . 38
4.1.2 Declare To . . . 38
4.1.3 Declare Through . . . 38

4.1.4 Parent 38
4.2 Printing Commands 38
4.2.1 Help 38
4.2.2 Print Declaration 39
4.2.3 Print Formula 39
4.2.4 Print History 39
4.2.5 Print Proof 39
4.2.6 Print Status 39
4.2.7 Print Syntax 39
4.2.8 Searching for Theorems 39
4.3 Undoing Declarations 40
4.3.1 Reset 40
4.3.2 Undo 40
4.3.3 Undo Back To 40
4.3.4 Undo Back Through 40
4.4 Interface Commands 40
4.4.1 Check 40
4.4.2 Quit 40
4.4.3 Read 41
4.4.4 Ztags 41

5 Proof Commands **43**
5.1 Starting Proofs 43
5.1.1 Try 43
5.1.2 Try Lemma 43
5.2 Undoing Proof Steps 44
5.2.1 Back 44
5.2.2 Retry 44
5.3 Case Splitting 44
5.3.1 Cases 44
5.3.2 Next 44
5.3.3 Split 45
5.3.4 Conjunctive 45
5.3.5 Disjunctive 45
5.4 Using Theorems and Definitions 45
5.4.1 Apply 45
5.4.2 Invoke 46
5.4.3 Invoke Predicate 46
5.4.4 Use 46
5.5 Using Equalities 47
5.5.1 Equality Substitute 47
5.6 Quantifiers 47
5.6.1 Instantiate 47
5.6.2 Prenex 47
5.7 Reduction 47
5.7.1 Simplify 48
5.7.2 Rewrite 48
5.7.3 Reduce 48
5.7.4 Trivial Simplify 48
5.7.5 Trivial Rewrite 49
5.7.6 Rearrange 49

5.7.7 Prove By Reduce . . . 49
5.7.8 Prove By Rewrite . . . 49
5.8 Modifiers . . . 49
5.8.1 With Disabled . . . 49
5.8.2 With Enabled . . . 50
5.8.3 With Expression . . . 50
5.8.4 With Predicate . . . 50
5.8.5 With Normalization . . . 50

A Collected Syntax **53**

B Collected Domain Checks **61**

C Differences Between Z/EVES and ***fuzz*** **67**

Chapter 1

Introduction

Z/EVES is a tool for analysing Z specifications. It can be used for parsing, type checking, domain checking, schema expansion, precondition calculation, refinement proofs, and proving theorems.

The language accepted by Z/EVES, Z/LaTeX, is a LaTeX [4] markup form that is compatible with the `zed`, `zed-csp`, and `fuzz` LaTeX styles and with Spivey's *fuzz* type checker, but with several extensions that are useful in stating theorems and their proofs. Appendix C summarizes these extensions. Z/LaTeX is described in full in Chapter 3.

Z/EVES is based on the EVES system [2, 3], and uses the EVES prover to carry out its proof steps. However, it should not be necessary to have any knowledge of EVES or its language (Verdi) in order to use Z/EVES.

This reference manual describes the language accepted by Z/EVES; the system and prover commands and their effects; and the different types of theorems that can be expressed. Where necessary, the actual Z/LaTeX form for samples of Z/LaTeX text is given. However, Z/LaTeX as typeset by LaTeX is much more readable, and this form is used where possible.

Other documents useful for understanding the Z/EVES system are

- a description of the Z/EVES version of the Mathematical Toolkit [7],
- a user's guide to Z/EVES [8], and
- installation guides and release notes for particular platforms (e.g., [5, 6]).

Chapter 2

Basic Concepts

2.1 The Z/EVES User Interface

Z/EVES is an interactive command-line system. This section describes the basic interactions with the system.

When running Z/EVES on a Unix system, it is possible to use the Emacs editor to run Z/EVES; this provides convenient facilities for editing input, retrieving past inputs, saving output to a file, and so on. See the Unix release notes [5] for details. When running Z/EVES under Windows (3.1, '95, or NT), a simple GUI allows for similar capabilities; details are in the Windows release notes [6].

Z/EVES inputs are entered at the prompt (`=>`). Z/EVES accepts Z paragraphs or Z/EVES commands.

2.1.1 Entering Specifications

Z/EVES maintains a *history*, representing a specification. Specifications can be entered interactively, paragraph by paragraph, or can be read from files. There are commands for deleting parts of a specification (see Section 4.3) or an entire specification (see Section 4.3.1).

Z/EVES can read files containing LaTeX markup using the `read` command (see Section 4.4.3). The paragraphs in the file are type checked and added to the history. It is also possible to type check a specification without adding it to the history; see Section 4.4.1.

The complete list of commands for maintaining a specification appear in Chapter 4.

2.1.2 Entering Commands

Z/EVES commands are sequences of words, terminated by a semicolon. The Z/EVES commands are described in detail in Chapters 4 and 5.

Some proof commands can take a long time to execute; these can be interrupted in a system-dependent way (with a control-C on Unix or the break key on Windows). See the release notes for details.

2.1.3 Recovery From Errors

In interactive use, after a syntax error Z/EVES may not display its prompt, because it is expecting more input. Usually, entering a semicolon will end the input; otherwise, interrupting the system (with a control-C on Unix or the break key on Windows) will return to the prompt.

2.2 Proving

Z/EVES allows for the statement and proof of theorems within a specification. Chapter 5 describes various proof step commands. Proofs work on a predicate called the *goal*; eack step transforms the goal into a new goal that is equivalent. Transforming a goal to *true* thus completes a proof.

Z/EVES proofs are based on a translation to first-order predicate calculus. This translation is not visible to the user, but it has ramifications, explained below, that affect the user.

2.2.1 Implicit Declarations

The Z notation allows for some predicates and expressions that cannot be directly translated into predicate calculus (e.g., set comprehensions and "let" forms). When such predicates or expressions are encountered, Z/EVES defines a function or predicate name, and replaces the predicate or expression by a function application. For example, the expression $\{ x : \mathbb{N} \mid x < 9 \}$ might get translated into an expression like $f0(\mathbb{N}, 9)$, where function $f0$ has been declared with the defining axiom $x \in f0(X, y) \Leftrightarrow x \in X \land x < y$. Normally, these implicit declarations are invisible to the user. However, the need to make these declarations imposes a constraint: EVES does not allow a declaration to be made in the middle of a proof. Therefore, in a proof step, it is not possible to use a predicate or expression that requires an implicit definition, and an error is generated if a user attempts to do so.

Implicitly declared functions are reused when possible, so after the above function $f0$ is defined, if the expression $\{ y : \mathbb{N}_1 \mid y < 100 \}$ is encountered, it can be replaced by $f0(\mathbb{N}_1, 100)$ and no new implicit declaration is required. In particular, therefore, any expression or predicate that has already appeared in a specification can be used in a proof step.

2.2.2 Type Information

The translation is to an *untyped* first-order logic, and type information does not appear in the translation. Thus, facts about "types" (e.g., $x \in \mathbb{N} \Rightarrow x \in \mathbb{Z}$ must be deduced. The standard Toolkit contains many rewriting rules that facilitate these deductions.

2.3 Domain Checking

The Z notation allows one to write expressions that are not meaningful. There are two ways to do so. First, a function can be applied outside its domain, as in $1 \mathbin{\mathsf{div}} 0$, $max\, \mathbb{Z}$, or $\# \mathbb{N}$. Second, a definite description (μ-term) is not meaningful if there is not, in fact, a single value satisfying the predicate. Examples are $\mu\, x : \mathbb{Z} \mid x \neq x$, for which there is no possible value of x satisfying the predicate, and $\mu\, x : \mathbb{Z} \mid x > 0$, for which there are many possible values.

Logicians have studied many ways of dealing with such meaningless expressions, and several different ways have been proposed for use in Z. There is, as yet, no consensus on the proper approach.

The approach in Z/EVES is to show that expressions are always meaningful. The type system of Z is not strong enough to guarantee this, so another kind of analysis is used. Z/EVES examines each paragraph as it is entered, and checks each function application and definite description for meaningfulness. Many applications are easily seen to be meaningful because the function applied is total over its type. For example, the application of $_ + _$ to two arguments is always defined, because type checking guarantees that the arguments are integers, and the domain of addition is all pairs of integers. In contrast, an application of $_ \mathbin{\mathsf{div}} _$ is not necessarily defined; type checking does not show that the second argument is non-zero.

The meaningfulness conditions for Z constructs are presented in "raw" form. Z/EVES applies trivial propositional simplification (e.g., $true \lor P$ is replaced by $true$) to these conditions before displaying them.

In the descriptions of the domain checking conditions in this document, we use variables (possibly decorated) to represent phrases of different grammatical types as follows:

Symbol	**Grammatical type**
e	expression
P, Q	predicate
ST	schema-text
SE	schema-exp
D	decl-part
n, v, N, X	name
PR	prefix relation symbol
IR	infix relation symbol
PG	prefix generic symbol
IG	infix generic symbol
F	infix function symbol

The descriptions define a function DC from Z phrases to Z predicates, which gives the domain checking condition for a phrase. The domain checking rules appear throughout the language description, and are also summarized in Appendix B.

Chapter 3

The Z/LaTeX Language

The Z/LaTeX language accepted by Z/EVES is almost a superset of the language accepted by the *fuzz* [10] tool; a few features of the *fuzz* language are not supported, and a number of extensions have been added. The differences between Z/LaTeX and *fuzz* are described in Appendix C.

3.1 Lexical Structure

Z/EVES operates in one of two modes. In interactive mode, comments are ignored, and all other text is significant. Z/EVES commands are recognized only in interactive mode. In batch mode, Z paragraphs, theorems, syntax declarations, input commands, and supported *fuzz* directives are significant, and all other text is ignored.

Interactive mode is for interactive use of Z/EVES, and batch mode is for reading LaTeX documents with embedded Z specification text. When Z/EVES starts, it is in interactive mode, and can be directed to read files in either mode.

3.1.1 Words

A word is an identifier, an "extended" identifier, a "quoted" identifier, a LaTeX command, or a symbol. Unlike *fuzz*, a word used as a schema name is not lexically different from any other word. Like *fuzz*, however, once a word has been declared to be part of a schema name, it may not be used in the declaration name of any other object, except another schema with a different name prefix.

An identifier is a nonempty sequence of decimal digits, letters, and escaped underscores (`_`) that begins with a letter. Case is significant; the identifer `foo` is different from the identifier `FOO`.

An extended identifier is an identifier which begins with a letter and contains escaped dollar signs (`\$`). Extended identifiers appear in names generated by Z/EVES; to avoid name clashes, names declared by the user (e.g., variable names) may not contain extended identifiers.

A quoted identifier is a backquote (`‘`) followed by a nonempty sequence of characters that does not contain backquote or newline, followed by a backquote. Like extended identifiers, quoted identifiers appear in names generated by Z/EVES. Quoted identifiers are not permitted in user-declared names, unless the contents of the quoted identifier are a valid (ordinary) identifier.

A LaTeX command consists of a backslash followed by either a nonempty sequence of letters, or by any single character. (The character ␣ in a syntax description or an example means a space.) Some LaTeX commands may have an argument, which is either a single decimal digit, or a left brace, a sequence of decimal digits, and a right brace.

A symbol is a nonempty sequence of the characters `*+-.<=>`. Symbols must be defined (Section 3.5.11) before they are used. Z/EVES always tries to scan the longest known symbol. If, for

example, the symbols `+` and `++` are both defined, the input `+++` will result in the two tokens `++` and `+`.

3.1.2 Numerals

A numeral is a nonempty sequence of decimal digits, optionally preceded with a negation sign (`\neg`, which prints as a raised minus sign). Thus, -3 is an expression (the application of function $(-)$ to the literal 3), whereas $\overline{\ }3$ is a negative literal.

3.1.3 Strings

A string is a double quote (`"`) followed by a possibly empty sequence of characters that does not contain double quote or newline, followed by a double quote.

3.1.4 Newlines, Spaces, and Ignored Text

A comment consists of the percent character (`%`), followed by a possibly empty sequence of any characters other than newline, and terminated by a newline character or by the end of the file. All comments, except for those that are recognized as supported *fuzz* directives (Section 3.1.6), are ignored by Z/EVES.

The LaTeX commands `\\` and `\also` denote newlines; the `\also` command adds some extra vertical space in LaTeX output. Ordinary newline characters and whitespace appearing in the text are ignored. A sequence of newlines is equivalent to one newline, and newlines appearing adjacent to one of the following tokens are ignored:

any operator symbol	`;`
`:`	`,`
`|`	`\mid`
`@`	`\spot`
`==`	`\defs`
`::=`	`\in`
`\THEN`	`\ELSE`

The following LaTeX commands may be used in the text for adjusting LaTeX spacing. They are ignored by Z/EVES.

`\,`	thin space
`\!`	negative thin space
`\:`	medium space
`\;`	thick space
`\@`	inter-sentence space
`\␣`	inter-word space
`\tn`	tab command, single digit argument
`\t{n}`	tab command, multi-digit argument
`~`	non-breakable inter-word space

The character sequence `{}` and the character `&` are ignored.

A comma or period appearing before the `\end` command of a paragraph is ignored.

3.1.5 Equivalents

The following pairs of tokens are considered equivalent:

```
@   \spot
|   \mid
```

The name `\empty` is a LaTeX command, so the Toolkit defines the equivalent name `\emptyset`.

3.1.6 *fuzz* Directives

A *fuzz* directive consists of two percent signs at the beginning of a line, followed by a space or by a recognized word, followed by more text, and terminated by a newline. A *fuzz* directive is lexically a comment, but it is recognized by Z/EVES. The following directives are recognized and supported:

`%%inop` symbols n	defines infix function symbols with precedence n
`%%postop` symbols	defines postop function symbols
`%%inrel` symbols	defines infix relation symbols
`%%prerel` symbols	defines prefix relation symbols
`%%ingen` symbols	defines infix generic symbols
`%%pregen` symbols	defines prefix generic symbols
`%%unchecked`	causes the next paragraph to be ignored by Z/EVES
`%%␣`text	the text is processed by Z/EVES
`%%ignore` symbols	the specified symbols are ignored by Z/EVES

Syntax declarations (Section 3.5.11) are used in the Z/EVES Toolkit rather than *fuzz* directives, because they allow an "interchange name" to be specified for the a declared symbol. Future work on Z/EVES may include an interchange format printer, and this will allow the Toolkit to be easily translated into that format. Specifications containing symbols which are defined with syntax declarations will also enjoy this benefit. (The interchange format is part of the proposed ISO standard for Z.)

3.2 Syntax Notation and Lexical Classes

The following conventions are used to describe the syntax:

- A definition is sometimes broken into a number of lines for readability; extra newlines and spaces in a definition are not significant.
- Nonterminals are written in roman font.
- Terminals are written in `typewriter` font.
- Lexical classes are written in *emphasized* font.
- The notation a ::= b means "a is defined to be b."
- The symbol | introduces alternation; a | b | c means "choose one of a, b, or c."
- The notation x . . . x means a nonempty sequence of x.
- The notation x y . . . y x means a nonempty sequence of x, separated by y.
- The notation [x] means x is optional.

A lexical class denotes a set of terminals. The following lexical classes appear in the syntax:

word	identifier, extended or quoted identifier, LaTeX command, or symbol
string	string
number	number
pre-gen	word defined as a prefix generic operator
in-gen	word defined as a infix generic operator
pre-rel	word defined as a prefix relational operator
in-rel	word defined as a infix relational operator
in-fun	word defined as a infix function operator
post-fun	word defined as a postfix function operator

3.3 Specifications

specification	::=	spec-item ... spec-item
spec-item	::=	z-section
	|	z-paragraph
	|	syntax-decl
	|	input-cmd
interaction	::=	decl-or-cmd ... decl-or-cmd
decl-or-cmd	::=	spec-item
	|	z-section-proof
	|	command `;`

In batch mode, a specification consists of a sequence of Z sections, Z section proofs, Z paragraphs, syntax declarations, and LaTeX-style file input commands. In interactive mode, Z/EVES commands are also allowed; Z/EVES commands are described in Chapter 4.

Note that in interactive mode, all (complete) commands must be terminated with a semicolon (`;`). In interactive or batch mode, commands in the proof part of a theorem must be separated with `\\`, `;`, or `\also`.

3.4 Z Sections

The Z/EVES Z section facility provides a way of dividing a specification into component modules. A Z section is a set of paragraphs that has been type checked and saved in a file (a "section file"); it can then be used in the definition of other Z sections. If a Z section `A` depends on (i.e., uses declarations from) a Z section `B`, then `B` is said to be a *parent* of `A`.

When a section is created, Z/EVES saves information about its parents as well as the declarations. A section is not permitted to depend on itself, or on a section whose parents have been modified and is possibly obsolete (Section 3.4.2.1).

Once a section has been created, the declarations in it can be proved. In the proof of the section, the order of declarations may differ from that of the section being proved, additional declarations may be made, and additional parents ("proof parents") may be specified for the section. If all the proof obligations for the section are discharged, Z/EVES records this fact, and any proof parents, in a file (a "section proof" file). See Section 4.1 for a description of Z section proofs.

The Z/EVES Z section facility has been designed to stay out of the way of users who do not wish to use it. If a user starts entering declarations and commands without first starting a section, the Toolkit section is first loaded, if it is not already present, before the first declaration or command is processed.

The Z/EVES Z section facility is based on the proposed Z standard for sections [1], and on ideas from the GNU Ada library facility.

3.4.1 Creating Z Sections

Z-section	::=	`\begin{zsection}{` ident `}{` [ident `,` ... `,` ident] `}`
		section-item ... section-item
		`\end{zsection}`
section-item	::=	z-paragraph
	|	syntax-decl
	|	input-cmd
command	::=	`zsection path` [*string* `,` ... `,` *string*]

A Z section looks like a declaration, but Z/EVES does not read the whole section before processing it. Rather, the `\begin{zsection}` and `\end{zsection}` lines act as commands that begin and end processing of the section, and each declaration in the section is processed as it is read. Thus, a section can be defined in both the batch and interactive input modes.

The `\begin{zsection}` line specifies the name of the section, and the names of its parents, if any. If a section is begun, and there are any declarations in the history, the user is given the option of saving them in a section whose name is `main` and whose (only) parent is the section `toolkit`. Z/EVES is then reset, the parent sections, if any, are loaded, and the system is then ready to accept the declarations in the section. Note that a declaration in a section can contain references only to previous declarations in the section, or to declarations in parent sections. There are no "default" parents. If you want the Toolkit section, you have to specify it as a parent.

Declarations in loaded sections cannot be printed with the `print declaration` command, except for theorem declarations.

The `\end{zsection}` line ends the definition of the current section. Information about the parents of the section and the declarations in the current development are saved in a file in the current directory; if the name of the section is `S`, the name of the file will be `S.sct`. Information about proofs in the current development is discarded, and so are implicit declarations added by Z/EVES. The latter will be recreated if and when the section is loaded as the parent of some other section. Finally, Z/EVES is reset.

Since proofs are discarded anyway, the `check` command (Section 4.4.1), which only does type checking, can be used to create a section file by checking a file which contains a section declaration. This is much faster than the usual Z/EVES processing.

3.4.2 Z Section Parents

When a section is begun, all its parents must exist. This enforces a "bottom-up" order of section creation. Z/EVES searches for the section file for a parent in the following places, in the order specified:

1. the current directory,
2. directories specified with the `zsection path` command, if any, and in
3. a system-dependent list of directories.

The system-dependent list of directories usually includes the directory containing the Z/EVES image or a sibling directory; the release notes for a particular version of Z/EVES will specify what directories are in the list for that version.

The syntax of the `zsection path` command is given above; each string in the command is a directory to search for section files. If no strings are specified, the system-dependent and user-specified search paths for sections are printed.

Although sections are type-checked before they are created, it is possible to get type errors when loading parents. Usually, this is because of name conflicts between two independent parents that are loaded into the current section. There is no qualification of names of objects declared in parents, so all global names in all parents of a section must be distinct.

3.4.2.1 Consistency Checks

Z/EVES checks that section files for parent sections are up-to-date, and that no section depends on itself, either directly or through some chain of ancestors. A section file for a section `S` is up-to-date if for every parent `P` of `S`:

1. a section file for `P` exists, and
2. the section file for `P` has not been modified since the section file for `S` was created, and
3. the section file for `P` is itself up-to-date.

For example, suppose section `A` has a parent `B`, and section `B` has no parents. After creating section files for `B` and `A`, in that order, we start a new section `C`, whose parent is `A`. If the section file for `B` was modified after the section file for `A` was created, then Z/EVES considers the section file for `A` to be out-of-date.

Continuing the example, suppose we have section files for `A` and `B` as described above. We then add `A` to `B`'s parent list and try to recreate `B`. Z/EVES will detect the attempt at circular dependency — `A` depends on `B`, and we are now trying to say that `B` depends on `A`.

3.5 Paragraphs

z-paragraph	::=	zed-box
	|	schema-box
	|	axiomatic-box
	|	generic-box
	|	theorem
para-opt	::=	`[` ability `]`
ability	::=	`enabled` | `disabled`
sep	::=	`\\` | `;` | `\also`

Paragraphs declare types, constants, global variables, schemas, and theorems. The following information is provided for each type of declaration:

- the syntax of the declaration,
- rules for determining whether the declaration is well-formed (type rules),
- the semantics of the declaration,
- proof obligations generated by Z/EVES that, when proven, ensure that functions are applied only to arguments in their domain (domain checking), and
- "implicit" declarations added by Z/EVES for the declaration.

A list of options may be specified for a paragraph. Currently, the only option recognized is an "ability;" if the `disabled` ability is specified, some declarations in the paragraph are not used automatically in proofs. The default ability is `enabled`.

Items in a paragraph must be separated by one of the above separators. To Z/EVES, they are equivalent, but LaTeX formats `\\` and `\also` as newlines, and adds extra vertical space for `\also`.

3.5.1 Zed Boxes

zed-box	::=	`\begin`[para-opt]`{zed}`
		zed-box-item sep ... sep zed-box-item
		`\end{zed}`
	|	`\begin`[para-opt]`{syntax}`
		zed-box-item sep ... sep zed-box-item
		`\end{syntax}`
zed-box-item	::=	given-set-definition
	|	schema-definition
	|	abbreviation-definition
	|	free-type-definition
	|	labelled-predicate

Each item in a zed box is added to the vocabulary as soon as it is processed, so later items can refer to earlier ones. To Z/EVES, a syntax box is the same as a zed box, but to LaTeX, the inside of a syntax box is an `eqnarray` environment, which can be useful for large free type definitions. Z/EVES ignores the LaTeX field-separator character `&`.

If a paragraph option is specified, it applies to all definitions and labelled predicates in the zed box. However, an ability may also be specified for a labelled predicate; if it is, this ability overrides the ability specified for the zed box.

Each zed box item is described in its own section, below.

3.5.2 Given Set Definitions

Syntax

given-set-definition	::=	`[` ident-list `]`

Domain Checking

$DC([N, \ldots]) = true$

Implicit Declarations

None.

3.5.3 Schema Definitions

Syntax

schema-definition	::=	schema-name [gen-formals] `\defs` schema-exp

Domain Checking

$DC(S \mathrel{\widehat=} SE) = DC(SE)$

Implicit Declarations

Schema definitions are treated as much like schema boxes as possible; see Section 3.5.7 for a description of the implicit declarations added for schema boxes.

The determination of the declaration part of a defined schema (for use in the $*declarationPart*$ theorem) is not straightforward in general, and Z/EVES determines it as follows: if the schema expression is a schema reference, this is the declaration part; if it is a schema text of the form $[D \mid P]$,

the declaration part is derived from D as explained in Section 3.5.7; if the schema expression is a conjunction, the declaration part is the conjunction of the declaration parts of the conjuncts. In any other case, the declaration part is simply *true*. In particular, then, a definition $S \mathrel{\widehat{=}} [D \mid P]$ is treated in the same way as the equivalent schema box (Section 3.5.7). The rules for θ-terms and binding sets are always added.

For defined schemas with definitions that are not conjunctions of schema references and schema texts, it is advisable for the user to add a forward rule giving suitable information about the declaration part; see the discussion in Section 3.5.7. For example, given the definition

$$
\begin{array}{l}
\mathit{AddItem} \mathrel{\widehat{=}} [\Delta \mathit{State} \mid \ldots] \\
\mathit{Success} \mathrel{\widehat{=}} [\mathit{report}! : \mathit{Report} \mid \mathit{report}! = \mathit{OK}] \\
\ldots \mathit{Op} \mathrel{\widehat{=}} (\mathit{AddItem} \land \mathit{Success}) \lor (\Xi \mathit{State} \land \mathit{NoRoom}),
\end{array}
$$

where Z/EVES is unable to determine a suitable declaration part for Op, we might add

theorem frule OpDeclarationPart
$\quad Op \Rightarrow \Delta State \land report! \in Report.$

The \$*declarationPart* theorem is useful in proofs where the *Op* schema is referred to, and we want to carry out proof steps without invoking it (and thus increasing the size of the formula). The forward rule allows Z/EVES to use some of information derived from the schema's definition without expanding it.

Proof Considerations

A schema determines an alphabet (that is, the set of components of the schema) and a predicate. This predicate can be explored in Z/EVES while working on a goal containing a schema-ref (by invoking the schema name).

The predicate associated with a schema defined by a schema definition has the form $x_1 \in T_1 \land \ldots \land x_k \in T_k \land P$, where $x_1, \ldots, x_k$ are the components of the schema, the T_i are expressions denoting the types of these components (as inferred by the type checker), and predicate P is derived from the schema expression as explained in Section 3.7. Under some circumstances, some of the component constraints are recognized as redundant, and Z/EVES omits them from the predicate. In particular, if the schema expression is a conjunction of schema references or schema texts, none of the component constraints are needed and the predicate is simply P.

3.5.4 Abbreviation Definitions

Syntax

abbreviation-definition	::=	def-lhs == expression
def-lhs	::=	var-name [gen-formals]
	|	*pre-gen* decoration ident
	|	ident *in-gen* decoration ident

Domain Checking

$DC(n[X] == e) = DC(e)$

Implicit Declarations

No theorems are added for definitions. The defined name can, however, be "invoked" (Section 5.4.2) in a proof step.

It is usually advisable to add a "typing" theorem for a name introduced in a definition. For example, given the definition

$$id[X] == \{x : X \bullet (x, x)\}$$

an assumption rule (Section 3.5.10.4) asserting something like $id[X] \in X \leftrightarrow X$ is useful. In fact, it is useful to make these typing rules as strong as possible, to provide the most information. In this case, the identity relation is in fact a bijection, and the typing theorem could be written as

theorem grule id_type $[X]$
  $\mathrm{id}\, X \in X \rightarrowtail\!\!\!\!\twoheadrightarrow X.$

3.5.5 Free Type Definitions

Syntax

free-type-definition	::=	ident `::=` branch `|` ... `|` branch
branch	::=	ident
	|	var-name `\ldata` expression `\rdata`

Domain Checking

$DC(N ::= a \mid b\langle\!\langle e\rangle\!\rangle) = DC(e)$

Implicit Declarations

For a free type $N ::= a \mid b\langle\!\langle e\rangle\!\rangle$, the following theorems are added:

- grule $a\$declaration : a \in N$,
- grule $b\$declaration : b \in e \rightarrow N$,
- rule $b\$injective : \forall x\$, x\$' : e \bullet b\,(x\$) = b\,(x\$') \Leftrightarrow x\$ = x\$'$,
- internal theorems which allow Z/EVES to infer that any two values on different branches are distinct,
- theorem $N\$expansion : N = a \cup \{\, x : e \bullet b(x)\,\}$,
- disabled rule $N\$member : x\$ \in N \Leftrightarrow x\$ = a \lor (\exists\, v0 : e \bullet x\$ = b(v0))$, and
- theorem $N\$induction : x \in N \land X \in \mathbb{P}\,N \land a \in X \land (\forall\, v0 : e \bullet b(v0) \in X) \Rightarrow x \in X$.

The induction property can be used as follows. To show $\forall\, n : N \bullet P(n)$:

1. define $inductionSet == \{\, n : N \mid P(n)\,\}$,
2. *try* $\forall\, n : N \bullet P(n)$; Z/EVES will strip the quantifier, leaving the goal $n \in N \Rightarrow P(n)$,
3. *use* $N\$induction[X := inductionSet, x := n]$,
4. *invoke* $inductionSet$, and
5. *prove*.

Eventually, this gets Z/EVES working on the subgoals $P(a)$ and $\forall\, v : N \mid P(v) \bullet P(b(v))$, which are the induction base and step cases, respectively.

(It is necessary to define *inductionSet* in this sequence of commands, because Z/EVES will not necessarily allow a set comprehension to be introduced in a proof step; see Section 2.2.1. Thus, we introduce the comprehension in a declaration, then refer to that definition in the proof step.)

3.5.6 Labelled Predicates

Syntax

labelled-predicate	::=	[label] predicate
label	::=	`\Label{` [ability] [usage] theorem-name `}`
usage	::=	`axiom` | `rule` | `grule` | `frule`

Domain Checking

$DC($[ability usage name] $P) = DC(P)$

Implicit Declarations

A labelled predicate is added as a theorem, with the name and usage specified in its label, if any. See Section 3.5.10 for an explanation of the meaning of the usage specifications, and Section 5.7 for an explanation of how the various kinds of theorems are used in reduction commands.

3.5.7 Schema Boxes

Syntax

schema-box	::=	`\begin`[para-opt]`{schema}{` schema-name `}` [gen-formals] decl-part [`\where` axiom-part] `\end{schema}`
decl-part	::=	basic-decl sep ... sep basic-decl
axiom-part	::=	predicate sep ... sep predicate

Type rules

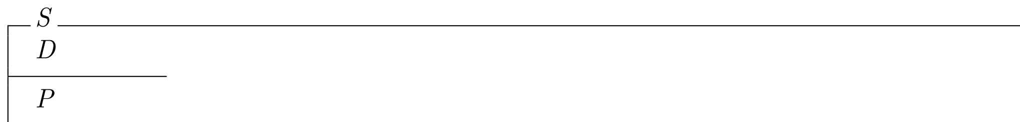
Declarations in the decl-part may not refer to previous declarations, but predicates in the axiom-part may refer to any declaration in the decl-part.

Domain Checking

$$DC\left(\begin{array}{|l} S \\ \hline D \\ \hline P \\ \hline \end{array}\right) = DC(D) \wedge (\forall D \bullet DC(P))$$

Implicit Declarations

For a schema box



the following theorem is added:

- frule $S\$declarationPart : S \Rightarrow D'$

  where D' is derived from the declaration part D by replacing declarations of the form $n_1, n_2 \ldots :$ T by predicates $n_1 \in T \wedge n_2 \in T \wedge \ldots$.

The following theorems are added for the schema used as a set:

- disabled rule $S\$member : x\$ \in S \Leftrightarrow (\exists S \bullet x\$ = \theta S)$,
- rule $S\$thetaMember : \theta S \in S \Leftrightarrow S$ (a special case of $S\$member$), and
- frule $S\$declaration : S \in \mathbb{P} \langle\!| n_1 : T_1, \ldots |\!\rangle$, where the T_i are determined by the type checker.

If the schema has an alphabet $\{n_1, n_2, \ldots\}$ that has not been encountered before, the following theorems are also added:

- rule $S\$select\$n_i : (\theta S).n_i = n_i$ for each i,
- rule $S\$thetasEqual : \theta S = \theta S' \Leftrightarrow n_1 = n_1' \wedge \cdots$ (alternatively, $\theta S_0 = \theta S_1 \Leftrightarrow \ldots$ if there are i and j for which $n_i = n_j'$),
- disabled rule $S\$inSet : x\$ \in \langle\!| n_1 : n_1', \ldots |\!\rangle \Leftrightarrow (\exists n_1 : n_1', \ldots \bullet x\$ = \theta S)$,
- rule $S\$thetaInSet : \theta S \in \langle\!| n_1 : n_1', \ldots |\!\rangle \Leftrightarrow n_1 \in n_1' \wedge \cdots$
  (a special case of $S\$inSet$. As above, we may have $n_{i0} : n_{i1}$ if necessary), and
- rule $S\$setInPowerSet : \langle\!| n_1 : n_1, \ldots |\!\rangle \in \mathbb{P} \langle\!| n_1 : n_1', \ldots |\!\rangle \Leftrightarrow n_1 = \{\} \vee \ldots \vee (n_1 \in \mathbb{P}\, n_1' \wedge \ldots)$.
  (As above, we may have n_{i0} and n_{i1} if necessary).

These rules for schemas used as sets are generated lazily, that is, only when a schema is in fact used as a set. Many schemas (e.g., schemas defining operations) are never used in this way and so these rules need never be generated.

3.5.8 Axiomatic Boxes

Syntax

axiomatic-box	::=	`\begin`[para-opt]`{axdef}`
		decl-part
		[`\where`
		labelled-axiom-part]
		`\end{axdef}`
labelled-axiom-part	::=	labelled-predicate sep ... sep

Type rules

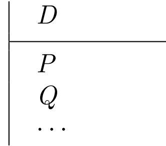
Declarations in the decl-part are added to the global vocabulary only after the entire decl-part is processed. Thus, a declaration in the decl-part may not refer to previous declarations in the decl-part, but the predicates in the labelled-axiom-part may refer to any declaration in the decl-part.

Domain Checking

$$DC\left(\begin{array}{|l} D \\ \hline P \\ Q \\ \cdots \end{array}\right) = DC(D) \wedge (\forall D \bullet DC(P) \wedge DC(Q) \wedge \cdots)$$

Implicit Declarations

For an axiomatic box



the following theorems are added:

- frule $n\$declaration : n \in S$
  for each declared name $n : S$ in the declaration part. If the name is declared more than once, the theorems derived from subsequent declarations have names $n\$declaration2$, $n\$declaration3$, and so on.
- Each predicate in the predicate part is also added as a theorem, using the name and usage provided in its label, if given; otherwise, it is added as an axiom with an internally generated name.

It is possible to write axiomatic boxes with schema references in their declaration parts. This seems to be rare in practice, and the Z/EVES support for this style is weak. In particular, no declaration theorems are added.

3.5.9 Generic Boxes

Generic boxes are similar to axiomatic boxes, except that they have a list of generic formals. The type rules, domain checks, and implicit declarations are all the same as those for an axiomatic box.

Syntax

generic-box	::=	`\begin`[para-opt]`{gendef}` [gen-formals]
		  decl-part
		[`\where`
		  labelled-axiom-part]
		`\end{gendef}`

3.5.10 Theorems

Theorem paragraphs allow for the expression of facts—or of properties that one hopes are facts. These facts can be proved using Z/EVES proof commands. In addition, theorems can be used in the proofs of other theorems. Theorems can be used at the user's request (with the `apply` or `use` commands), or may be used automatically by the `prove`, `reduce`, `rewrite`, or `simplify` commands (depending on the *usage* given in the theorem declaration).

WARNING: The syntactic restrictions on rewrite, forward, and assumption rules, described below, are not checked by the current version of Z/EVES. Expect strange error messages when the syntax rules are violated. We plan to implement these checks in a future release.

Syntax

theorem	::=	`\begin`[para-opt]`{theorem}{`[usage] theorem-name`}`[gen-formals]
		  predicate
		[`\proof`
		  command sep . . . sep command]
		`\end{theorem}`

Type rules

Theorems are type checked, but, unlike other paragraphs, may contain undeclared (free) variables.

Domain Checking

No domain checking condition is generated. (There is a technical difficulty in doing so, as the necessary type information is not always available.)

Implicit Declarations

None.

3.5.10.1 Facts

The usage `axiom` specifies that a theorem is to be used as a fact. Facts are not considered during reduction; they must be explicitly added to the current goal with the `use` command (Section 5.4.4).

3.5.10.2 Rewrite Rules

The usage `rule` specifies that a theorem is to be used as a rewrite rule.

A theorem can be a rewrite rule if its predicate is a rewrite-rule according to the following syntax:

rewrite-rule	::=	`\forall` schema-text `@` rewrite-rule
	|	predicate `\implies` rewrite-rule
	|	lhs-e `=` expression
	|	lhs-p `\iff` predicate
	|	lhs-p (equivalent to lhs-p `iff true`)
	|	`\lnot` lhs-p (equivalent to lhs-p `iff false`)

where lhs-e is an expression that does not contain quantifications, and lhs-p is a predicate that does not contain quantification or logical connectives.

The expression or predicate on the left side of the equality or equivalence is referred to as the *pattern*, and the expression or predicate on the right side is referred to as the *replacement*. A rewrite rule which is a quantification or an implication is said to be a *conditional* rewrite rule; the condition for the former is that the variables bound by the schema-text are in their types and that the schema-text predicate (if given) is true, and the condition for the latter is the hypothesis. The condition must be shown to be true (the way in which the condition is shown to be true depends on the proof command used) before the rewrite rule can be applied. See Section 5.7 for a description of how rewrite rules are applied during reduction commands.

Examples of rewrite rules can be found throughout the Toolkit.

3.5.10.3 Forward Rules

The usage `frule` specifies that a theorem is to be used as a forward rule.

A theorem can be a forward rule if its predicate is a forward-rule according to the following syntax:

forward-rule	::=	schema-ref `\implies` predicate `\land` ... `\land` predicate

where the schema-ref (the hypothesis) has no replacements, and the predicate(s) in the conclusion contain no quantifiers or logical connectives, except possibly a leading `\lnot`.

See Section 5.7 for a description of how forward rules are applied during reduction commands. Forward rules are usually used to introduce inequalities, which are handled specially by the Z/EVES simplifier.

3.5.10.4 Assumption Rules

The usage `grule` specifies that a theorem is an assumption rule.

A theorem can be an assumption rule if its predicate is an assumption-rule according to the following syntax:

assumption-rule	::=	simple-predicate
	|	assumption-hyp `\implies` simple-predicate
assumption-hyp	::=	simple-predicate [`\land`] . . . [`\land`] simple-predicate
simple-predicate	::=	[`\lnot`] simple-predicate-2
simple-predicate-2	::=	term `\in` term
	|	term `=` term
	|	simple-schema-ref

where a term is an expression not containing a local variable binding, a conditional expression, a definite description, or a set comprehension, and a simple-schema-ref is a schema reference in which generic actuals and replacement expressions are terms.

An additional restriction on assumption rules is that the conclusion, or at least one subexpression of the conclusion, must be a *trigger*. A trigger is:

- a simple-schema-ref with no replacements, or,
- a global constant (e.g., `\nat` or `succ`); if the constant is generic, generic actuals must be supplied, and must be distinct variable names (e.g., `myConstant[X,Y]`), or
- an application of a prefix or infix generic (e.g., `\seq X`, `X \fun Y`); the generic actual(s) must be distinct variable names.

Furthermore, all free variables in the assumption rule must appear in the trigger. If more than one expression can be a trigger, the first one (reading left-to-right) becomes the trigger.

There are some other special cases of assumption rules and triggers to be found in the Toolkit. The rules for determining whether a theorem predicate is a valid assumption rule are actually rather involved and depend on details of the translation from Z to Verdi. However, the above restrictions will always result in valid assumption rules, and should be sufficient for user-developed theories.

See Section 5.7 for a description of how assumption rules are applied during reduction commands. Usually, assumption rules are used to express type information in Z/EVES. Whenever a constant is declared in an axiomatic box (Section 3.5.8) or generic box (Section 3.5.9), Z/EVES automatically adds an assumption rule expressing its type. The assumptions added by these rules are used together with the rules in the Toolkit to enable certain obvious inferences to be drawn.

3.5.11 Syntax Declarations

syntax-decl	::=	`\syndef{`string-or-word`}{`opclass`}{`[string-or-word]`}`
string-or-word	::=	*string* | *word*
opclass	::=	`infun1` | `infun2` | `infun3` | `infun4` | `infun5` | `infun6`
	|	`ingen` | `pregen`
	|	`inrel` | `prerel`
	|	`postfun`
	|	`word`
	|	`ignore`

A syntax declaration allows the declaration of the following:

- A prefix, infix, or postfix operator symbol. Z/EVES can then use the operator symbol as an abbreviation in parsing and printing.
- An interchange name for a symbol. The symbol has no special syntactic significance.

- A symbol which is to be ignored. Z/EVES considers this symbol to be whitespace.

The first string-or-word specifies the symbol to be declared. If the symbol is an identifier or LaTeX command, it is given as a word. Otherwise, the operator symbol is a sequence of symbol characters, and is given as a string.

The opclass specifies the syntactic category of the operator symbol, i.e., what kind of operator it is, or whether an interchange name is being specified for the symbol, or whether the symbol is to be ignored.

The second string-or-word specifies the "internal name" of the operator symbol. This is usually the interchange name, but may be any name that does not conflict with an existing internal name. If the name is a valid identifier or LaTeX command, it is given as a word. Otherwise, it is given as a string. If the symbol is to be ignored, this argument should be specified as `{}`, since the symbol will have no internal name.

3.5.12 Input Commands

input-cmd	::=	`\input{`text`}`
	|	`\include{`text`}`

Unlike the `read` and `read script` commands, input commands are LaTeX commands, and are recognized in both interactive and batch modes.

The text argument in an input command can be any sequence of characters excluding braces (`{}`) and newline, and is the name of the file to read. If the file name does not include an extension, `.tex` is appended to it. The file is read in batch mode. The LaTeX `\includeonly` command is not supported.

3.6 Declarations and Scope

The *scope* of an identifier is the set of textual regions in a specification in which the identifier is known. Identifiers are introduced by paragraphs and syntax declarations; identifiers introduced by paragraphs may be *global*, e.g., a theorem name, or *local*, e.g., a schema component. Variables local to a part of a predicate or expression may also be introduced by, e.g., quantifications and `\LET` expressions. It is possible for the scope of an identifier to be non-contiguous; an example of this is when a global identifier is declared to be a local variable in some scope. Before this scope, the identifier has its global meaning, in the scope, it has its local meaning, and after the scope, the meaning is again global.

Declarations introduce named variables. The scope of these variables may be local or global, depending on the context in which a declaration appears. For example, the scope of a variable declared in an axiomatic box is global: the scope begins after the end of the box, and extends to the end of the specification. (The variable is also accessible, as a local variable, in the predicate part of the box.) In contrast, the scope of a variable declared in the declaration part of a schema is local: it is the axiom part of the schema.

A declaration that is a schema reference introduces the variables that are the components of the schema.

Syntax

declaration	::=	basic-decl `;` ... `;` basic-decl
basic-decl	::=	decl-name-list `:` expression
	|	schema-ref
decl-name-list	::=	decl-name `,` ... `,` decl-name

Domain Checking

$DC(n, \ldots : e) = DC(e)$
$DC(S[e, \ldots]) = DC(e) \land \cdots$
$DC(D;\ D') = DC(D) \land DC(D')$

Proof Considerations

The meaning of a declaration $x : S$ is the same as the predicate $x \in S$, but there is a significant difference in the applicability of proof steps to declarations and predicates. Predicates can be rewritten or (if they are schema references) invoked; declarations cannot (although any expressions appearing in them may be). For example, the Toolkit defines rewrite rules that turn $x \in A \cup B$ into $x \in A \lor x \in B$, but there are no corresponding rules for the declaration $x : A \cup B$ (and, indeed, $x : A \lor x : B$ is not valid syntax).

The `prenex` command (see Section 5.6.2) eliminates quantifiers whenever possible, converting declarations into predicates. It is usually beneficial to prenex whenever possible.

Z/EVES generates internal forward rules, so that inside the scope of a declaration, the corresponding predicate is assumed. Thus, if a user-defined forward rule applies to a schema reference (e.g., $S \Rightarrow x > 0$), the rule is applied when S appears as a declaration. In this case, the internal forward rule asserts S as a predicate; which then triggers the user's forward rule.

3.7 Schema Expressions

```
schema-exp      ::= \forall schema-text @ schema-exp
                  | \exists schema-text @ schema-exp
                  | \exists_1 schema-text @ schema-exp
                  | schema-exp-1
schema-exp-1    ::= [ schema-text ]
                  | schema-ref
                  | \lnot schema-exp-1
                  | \pre schema-exp-1
                  | schema-exp-1 \land schema-exp-1
                  | schema-exp-1 \lor schema-exp-1
                  | schema-exp-1 \implies schema-exp-1
                  | schema-exp-1 \iff schema-exp-1
                  | schema-exp-1 \project schema-exp-1
                  | schema-exp-1 \hide ( decl-name-list )
                  | schema-exp-1 \semi schema-exp-1
                  | schema-exp-1 \pipe schema-exp-1
                  | ( schema-exp-1 )
```

In the above syntax, operators defined in later productions have higher precedence than operators defined in earlier productions. Within a production, operators defined earlier in the production have higher precedence than operators defined later in the production. All binary operators are left-associative, except `\implies`, which associates to the right.

Proof Considerations

In proof, Z/EVES treats schemas as predicates. Most of the schema operators correspond to obvious predicate operators (e.g., $\land$); we will explain those cases where there is a non-obvious correspondence.

3.7.1 Schema Quantifications

Syntax

schema-exp	::=	`\forall` schema-text `@` schema-exp
	|	`\exists` schema-text `@` schema-exp
	|	`\exists_1` schema-text `@` schema-exp

Type rules

The scope rules for schema quantification and predicate quantification are subtly different: the quantifier in a schema quantification captures names declared in the schema expression but not names used there. For example, in $\forall X : \mathbb{P}\mathbb{N} \bullet S[X] \land [a : X] \land [X : \mathbb{P}\mathbb{Z} \mid \ldots]$, only the rightmost X is captured by the quantifier, the other two refer to X in an outer scope.

Domain Checking

$DC(\forall D \mid P \bullet SE) = DC(D) \land (\forall D \bullet DC(P)) \land DC(SE)$
$DC(\exists D \mid P \bullet SE) =$ as above
$DC(\exists_1 D \mid P \bullet SE) =$ as above

The domain checking conditions are slightly weaker than necessary; it would be possible to have

$$DC(\forall D \mid P \bullet SE) = DC(D) \land (\forall D \bullet DC(P)) \land ((\exists D \bullet P) \Rightarrow DC(SE))$$

instead. We decided to use the simpler form above, as it seems unlikely that $\exists D \mid P$ will be false in many cases.

Proof Considerations

The unique existence quantifier for schemas is not yet fully supported; however, it is possible to type check specifications using it (with the `check` command).

Z/EVES renames the bound variable if necessary in the predicate associated with the schema quantification, to account for the different scope rules in schema expressions and predicates.

3.7.2 Schema Texts

Syntax

schema-exp-1	::=	`[` schema-text `]`
schema-text	::=	declaration [`|` predicate]

Domain Checking

$DC([D \mid P]) = DC(D) \land (\forall D \bullet DC(P))$

See Section 3.6 for a description of the domain checking for declarations.

3.7.3 Schema References

Syntax

schema-ref	::=	schema-name decoration [gen-actuals] [replacements]
schema-name	::=	[prefix] *word*
prefix	::=	`\Delta` | `\Xi`

replacements	::=	`[` rename-or-repl `,` ... `,` rename-or-repl `]`
rename-or-repl	::=	decl-name `/` decl-name
	|	decl-name `:=` expression

Type rules

In a replacement, the expression must have the same type as the decl-name component of the schema.

Z/EVES will automatically declare a Δ or Ξ schema if it appears in a schema reference without having been declared; this follows the common convention. The default declarations are $\Delta S \mathrel{\widehat=} [S;\ S']$ and $\Xi S \mathrel{\widehat=} [S;\ S' \mid \theta S = \theta S']$.

It is possible for the default definition of a Δ or Ξ schema to be ill-typed, in which case the schema reference containing the Δ or Ξ is itself in error.

Semantics

Replacements specify values for components. Any replaced component is not in the alphabet of the schema reference. For example, given the schema $S \mathrel{\widehat=} [x, y : \mathbb{Z} \mid x < y]$, the reference $S[x := 0]$ is equivalent to a reference to the schema $[y : \mathbb{Z} \mid 0 < y]$.

Domain Checking

$DC(S[X, Y][x/y, z := e]) = DC(X) \land DC(Y) \land DC(e)$

3.7.4 Schema Negation

Syntax

schema-exp-1	::=	`\lnot` schema-exp-1

Domain Checking

$DC(\lnot\ SE) = DC(SE)$

3.7.5 Schema Precondition

Syntax

schema-exp-1	::=	`\pre` schema-exp-1

Domain Checking

$DC(\text{pre } SE) = DC(SE)$

Proof Considerations

The predicate associated with a schema precondition is explicitly quantified; thus, for $S \mathrel{\widehat=} [\Delta T;\ r! : RT \mid \ldots]$, the predicate for pre S is $\exists x' : X, \ldots R! : RT \bullet S$, where x' is declared in T' and X is determined by the type checker.

3.7.6 Binary Logical Schema Operators

Syntax

schema-exp-1	::=	schema-exp-1 `\land` schema-exp-1
	|	schema-exp-1 `\lor` schema-exp-1
	|	schema-exp-1 `\implies` schema-exp-1
	|	schema-exp-1 `\iff` schema-exp-1

Domain Checking

$DC(SE \land SE') = DC(SE) \land DC(SE')$
$DC(SE \lor SE') = DC(SE) \land DC(SE')$
$DC(SE \Rightarrow SE') = DC(SE) \land DC(SE')$
$DC(SE \Leftrightarrow SE') = DC(SE) \land DC(SE')$

3.7.7 Schema Projection and Hiding

Syntax

schema-exp-1	::=	schema-exp-1 `\project` schema-exp-1
	|	schema-exp-1 `\hide` `(` decl-name-list `)`

Domain Checking

$DC(SE \upharpoonright SE') = DC(SE) \land DC(SE')$
$DC(SE \setminus (n, \ldots)) = DC(SE)$

Proof Considerations

The predicate associated with a projection or hiding is existentially quantified, as for a schema precondition.

3.7.8 Schema Compositions

Syntax

schema-exp-1	::=	schema-exp-1 `\semi` schema-exp-1
	|	schema-exp-1 `\pipe` schema-exp-1

Domain Checking

$DC(SE \fatsemi SE') = DC(SE) \land DC(SE')$
$DC(SE \gg SE') = DC(SE) \land DC(SE')$

Proof Considerations

The predicates associated with these schema operations, like those for preconditions and hiding, are explicitly quantified.

3.8 Predicates

predicate	::=	`\forall` schema-text `@` predicate
	|	`\exists` schema-text `@` predicate
	|	`\exists_1` schema-text `@` predicate
	|	`\LET` let-def `;` ... `;` let-def `@` predicate
	|	`\IF` predicate `\THEN` predicate `\ELSE` predicate
	|	predicate-1
predicate-1	::=	expression rel expression rel ... rel expression
	|	*pre-rel* decoration [gen-actuals] expression
	|	schema-ref
	|	`\pre` schema-ref
	|	`\lnot` predicate-1
	|	predicate-1 `\land` predicate-1
	|	predicate-1 `\lor` predicate-1
	|	predicate-1 `\implies` predicate-1
	|	predicate-1 `\iff` predicate-1
	|	`true`
	|	`false`
	|	`(` predicate `)`

In the above syntax, operators defined in later productions have higher precedence than operators defined in earlier productions. Within a production, operators defined earlier in the production have higher precedence than operators defined later in the production. All binary operators are left-associative, except `\implies`, which associates to the right.

The domain checking conditions for predicates do not respect logical equivalences. For example, although $P \Rightarrow Q$ is logically equivalent to $(\neg\ Q) \Rightarrow (\neg\ P)$, the conditions generated for the two predicates are different. The conditions use a left to right reading of the predicate, so, for example, for $P \Rightarrow Q$ to be meaningful, first P must be meaningful, and then Q must be meaningful whenever P holds. This allows predicates to be guarded, as in the formula $x \neq 0 \Rightarrow x \mathsf{div}\ x = 1$.

3.8.1 Quantifications

Syntax

predicate	::=	`\forall` schema-text `@` predicate
	|	`\exists` schema-text `@` predicate
	|	`\exists_1` schema-text `@` predicate

Domain Checking

$DC(\forall D \mid P \bullet Q) = DC(D) \land (\forall D \bullet DC(P) \land (P \Rightarrow DC(Q)))$
$DC(\exists D \mid P \bullet Q) =$ as above
$DC(\exists_1 D \mid P \bullet Q) =$ as above

Proof Considerations

Unique existence is handled rather strangely in the prover; a predicate $\exists_1 D \mid P \bullet Q$ has an internal form using the set comprehension $\{D \mid P \land Q\}$ (which is asserted to have a single element). An internal rewrite rule converts this to a form involving the standard quantifiers.

Because the internal form uses a set comprehension, it may not be possible to use a unique existence quantifier in a predicate appearing in a proof step (unless the quantification has already appeared in some formula). See Section 2.2.1.

3.8.2 Local Variable Binding

Syntax

predicate	::=	`\LET` let-def `;` ... `;` let-def `@` predicate
let-def	::=	var-name `==` expression

Domain Checking

$DC(\textbf{let}\ x == e;\ \ldots \bullet P) = DC(e) \land \cdots \land (\textbf{let}\ x == e;\ \ldots \bullet DC(P))$

Proof Considerations

Let predicates are not represented directly in the prover; in particular, any part of the body of a let predicate that contains a reference to one of the local variables cannot be directly manipulated or changed in a proof step.

The commands `invoke` and `reduce` will apply a β-reduction, that is, the let form will be replaced by an expanded body. For example, the predicate $\textbf{let}\ x == 1, y == 2 + 3 \bullet x < y$ will be changed to $1 < 2 + 3$.

It is not possible to use a let predicate in a proof step, unless that predicate (or one like it) has already appeared in some paragraph of a specification; see Section 2.2.1.

3.8.3 Conditional Predicates

Syntax

predicate	::=	`\IF` predicate `\THEN` predicate `\ELSE` predicate

Domain Checking

$DC(\textbf{if}\ P\ \textbf{then}\ Q\ \textbf{else}\ R) = DC(P) \land (\textbf{if}\ P\ \textbf{then}\ DC(Q)\ \textbf{else}\ DC(R))$

Proof Considerations

If the "test" predicate (P in the above example) is a conditional predicate or an application of a Boolean connective, it may be necessary to use normalization in a proof; see Section 5.8.5.

3.8.4 Binary Relations

Syntax

predicate-1	::=	expression rel expression rel ... rel expression
rel	::=	`=` | `\in` | *in-rel* decoration [gen-actuals] | `\inrel{` ident `}`

Domain Checking

$DC(e_1\ IR\ e_2 \ldots) = DC(e_1) \land DC(IR) \land DC(e_2) \cdots$

Proof Considerations

A predicate $x\ R\ y$ is represented internally as $(x, y) \in (_R_)$, except when R is one of the predefined arithmetic comparisons.

3.8.5 Unary Relations

Syntax

predicate-1	::=	*pre-rel* decoration [gen-actuals] expression

Domain Checking

$DC(PR\ e) = DC(e) \land DC(PR)$

Proof Considerations

A predicate $PR\ e$ is represented internally as $e \in (PR_)$.

3.8.6 Schema References

Syntax

predicate-1	::=	schema-ref
	|	`\pre` schema-ref

Domain Checking

See Section 3.7.3.

Proof Considerations

A schema precondition is represented by an existential quantification, as described in Section 3.7.5.

3.8.7 Logical Connectives

Syntax

predicate-1	::=	`\lnot` predicate-1
	|	predicate-1 `\land` predicate-1
	|	predicate-1 `\lor` predicate-1
	|	predicate-1 `\implies` predicate-1
	|	predicate-1 `\iff` predicate-1

Domain Checking

$DC(\neg\ P) = DC(P)$
$DC(P \land Q) = DC(P) \land (P \Rightarrow DC(Q))$
$DC(P \lor Q) = DC(P) \land (P \lor DC(Q))$
$DC(P \Rightarrow Q) = DC(P) \land (P \Rightarrow DC(Q))$
$DC(P \Leftrightarrow Q) = DC(P) \land DC(Q)$

3.8.8 Constants

Syntax

predicate-1	::=	`true`
	|	`false`

Domain Checking

$DC(true) = true$
$DC(false) = true$

3.9 Expressions

```
expression-0   ::=  \lambda schema-text @ expression
                |   \mu schema-text [@ expression]
                |   \LET let-def ; ... ; let-def @ expression
                |   expression
expression     ::=  \IF predicate \THEN expression \ELSE expression
                |   expression-1
expression-1   ::=  expression-1 in-gen decoration expression-1
                |   expression-2 \cross ... \cross expression-2
                |   expression-2
expression-2   ::=  expression-2 in-fun decoration [gen-actuals] expression-2
                |   pre-gen decoration expression-4
                |   - decoration [gen-actuals] expression-4
                |   expression-4 \limg expression-0 \rimg decoration [gen-actuals]
                |   expression-3
expression-3   ::=  expression-3 expression-4
                |   expression-4
expression-4   ::=  [local-or-global] var-name [gen-actuals]
                |   number
                |   schema-ref
                |   \{ schema-text [@ expression] \}
                |   \{ [expression-list] \}
                |   \langle expression-list \rangle
                |   \lbag expression-list \rbag
                |   \lblot binding ; ... ; binding \rblot
                |   ( expression-list )
                |   \theta schema-name decoration [replacements]
                |   expression-4 . var-name
                |   expression-4 . number
                |   expression-4 post-fun decoration [gen-actuals]
                |   expression-4 \bsup expression \esup
                |   expression-4 ^{ expression }
                |   ( expression-0 )
```

In the above syntax, operators defined in later productions have higher precedence than operators defined in earlier productions. Within a production, operators defined earlier in the production have higher precedence than operators defined later in the production. All binary operators are left-associative, except infix generic (*in-gen*) operators, which associate to the right. Note that $\times$ is not an associative operator; $A \times B \times C$ means neither $(A \times B) \times C$ nor $A \times (B \times C)$. Also, note that the `\power` operator is considered to be an ordinary prefix generic operator.

A word defined as an *in-fun* operator is given a "precedence" between one and six. The higher the precedence, the higher the binding power.

3.9.1 Lambda Expressions

Syntax

expression-0	::=	`\lambda` schema-text `@` expression

Domain Checking

$DC(\lambda D \mid P \bullet e) = DC(D) \wedge (\forall D \bullet DC(P) \wedge (P \Rightarrow DC(e)))$

Proof Considerations

A lambda expression $\lambda D \mid P \bullet e$ is equivalent to the set comprehension $\{ D \mid P \bullet (T, e) \}$, where T is the characteristic tuple of the declaration D. No special rules are generated beyond those for the comprehension; in particular, Z/EVES has no special knowledge that the lambda expression determines a (partial) function. It will probably be necessary, therefore, for the user to state and prove such theorems.

As explained in Section 2.2.1, it may not be possible to use a lambda expression in a proof command.

3.9.2 Definite Descriptions

Syntax

expression-0	::=	`\mu` schema-text [`@` expression]

Domain Checking

$DC(\mu D \mid P \bullet e) = DC(D) \wedge (\forall D \bullet DC(P) \wedge (P \Rightarrow DC(e))) \wedge (\exists_1 D \bullet P)$

Proof Considerations

Definite description terms are converted to a standard form $\mu x : S$ (if they are not already in that form); in general $\mu D \mid P \bullet e$ is converted to $\mu x : \{D \mid P \bullet e\}$. Internally, this form is represented as a function of the set S. The Toolkit gives several rules for working with this form.

As explained in Section 2.2.1, it may not be possible to use a definite description in a proof command.

3.9.3 Local Variable Binding

Syntax

expression-0	::=	`\LET` let-def `;` ... `;` let-def `@` expression

Domain Checking

$DC(\mathbf{let}\ x == e;\ \ldots \bullet e') = DC(e) \wedge \cdots \wedge (\mathbf{let}\ x == e;\ \ldots \bullet DC(e'))$

Proof Considerations

See the comments on local variable bindings in predicates in Section 3.8.2.

3.9.4 Conditional Expressions

Syntax

expression	::=	`\IF` predicate `\THEN` expression `\ELSE` expression

Domain Checking

$DC(\textbf{if } P \textbf{ then } e \textbf{ else } e') = DC(P) \wedge (\textbf{if } P \textbf{ then } DC(e) \textbf{ else } DC(e'))$

3.9.5 Cross Products

Syntax

expression-1	::=	expression-2 `\cross` ... `\cross` expression-2

Domain Checking

$DC(e \times e' \times \cdots) = DC(e) \wedge DC(e') \wedge \cdots$

Proof Considerations

There are no built-in facts about cross products and tuples; suitable rules can be added manually if need be. The Toolkit gives rules for two and three-element cross products.

3.9.6 Function Applications

Syntax

expression-2	::=	expression-2 *in-fun* decoration [gen-actuals] expression-2
	|	`-` decoration [gen-actuals] expression-4
	|	expression-4 `\limg` expression-0 `\rimg` decoration [gen-actuals]
expression-3	::=	expression-3 expression-4
expression-4	::=	expression-4 *post-fun* decoration [gen-actuals]
	|	expression-4 `\bsup` expression `\esup`
	|	expression-4 `^{` expression `}`

Domain Checking

$DC(e\,F\,e') = DC((_F_)(e, e'))$

$$DC(f(a)) = \begin{cases} \mathit{true} & \text{if f is a total function} \\ a \in \operatorname{dom} f & \text{if f is a partial function} \\ f\ \mathit{applies}\$\mathit{to}\ a & \text{otherwise} \end{cases}$$

For the analysis of function applications, Z/EVES considers the declaration of the function (when the function is a variable or constant) to determine if it is total, partial, or unknown.

Proof Considerations

Except in the case of a predefined arithmetic function, an infix function application is represented internally as a normal function application. For example, $1 \mathrel{..} 9$ is represented as $(_ \mathrel{..} _)(1, 9)$.

3.9.7 Variable References

Syntax

expression-1	::=	expression-1 *in-gen* decoration expression-1
	|	expression-2
expression-2	::=	*pre-gen* decoration expression-4
expression-4	::=	[local-or-global] var-name [gen-actuals]
gen-actuals	::=	`[` expression-list `]`
local-or-global	::=	`\Local` | `\Global`

Semantics

A name preceded by `\Local` is interpreted as a variable name, even if there is a constant of the same name. Conversely, a name preceded by `\Global` is interpreted as a reference to a global constant, even if there is a variable of that name in scope.

Domain Checking

$DC(e\ IG\ e') = DC(e) \land DC(e')$
$DC(PG\ e) = DC(e)$
$DC(v[e_1, \ldots]) = DC(e_1) \land \ldots$

Proof Considerations

The `\Local` and `\Global` tags can occur in Z/EVES output as a result of a prenex or substitution operation.

3.9.8 Constants

Syntax

expression-4	::=	*number*

Domain Checking

$DC(number) = true$

3.9.9 Schema References

Syntax

expression-4	::=	schema-ref

Domain Checking

See Section 3.7.3.

Proof Considerations

Rules pertaining to schema-refs are declared when schemas are declared; see Section 3.5.7.

3.9.10 Set Comprehensions

Syntax

expression-4 ::= `\{` schema-text [`@` expression] `\}`

The syntax for set expression/comprehension is ambiguous; the expression `\{ S \}` may be either a singleton set, or, if `S` is a schema reference, a set comprehension. If `S` is a schema, the expression is assumed to be a comprehension. For a singleton set containing a schema reference, parenthesize the reference: `\{ (S) \}`.

Domain Checking

$DC(\{ D \mid P \bullet e \}) = DC(D) \land (\forall D \bullet DC(P) \land (P \Rightarrow DC(e)))$

Proof Considerations

Set comprehensions are not represented directly in the prover; in particular, any part of the comprehension that contains a reference to one of the local variables cannot be directly manipulated or changed in a proof step.

Two internal rewriting rules apply to comprehensions. For the comprehension $\{ D \mid P \bullet e \}$, they are

- $x \in \{ D \mid P \bullet e \} \Leftrightarrow (\exists D \mid P \bullet x = e)$, and
- $\{ D \mid P \bullet e \} \in \mathbb{P} X \Leftrightarrow (\forall D \mid P \bullet e \in X)$.

Any command that applies rewrite rules can apply these rules. The second fact is somewhat weak, in that it does not help in showing that a comprehension defines a relation or partial function; such rules must be added by the user if needed.

It is not possible to use a comprehension in a proof step, unless the comprehension (or one like it) has already appeared in some paragraph of a specification; see Section 2.2.1.

3.9.11 Set Displays

Syntax

expression-4 ::= `\{` [expression-list] `\}`
expression-list ::= expression `,` . . . `,` expression

Domain Checking

$DC(\{e, e', \ldots\}) = DC(e) \land DC(e') \land \cdots$

Proof Considerations

The empty set is represented internally as a constant; unit sets are represented as functions of their element; displays with two or more elements are represented as unions of unit sets.

3.9.12 Sequence Displays

Syntax

expression-4 ::= `\langle` expression-list `\rangle`

Domain Checking

$DC(\langle e, e', \ldots \rangle) = DC(e) \land DC(e') \land \cdots$

Proof Considerations

The empty sequence is represented internally as a constant; unit sequences are represented as functions of their element; displays with two or more elements are represented as concatenations (i.e., applications of function $_ \frown _$) of unit sequences.

3.9.13 Bag Displays

Syntax

expression-4	::=	`\lbag` expression-list `\rbag`

Domain Checking

$DC([\![e, e', \ldots]\!]) = DC(e) \land DC(e') \land \cdots$

Proof Considerations

The empty bag is represented internally as a constant. Unit bags are represented as functions of their element. Displays with two or more elements are represented as bag unions of unit bags.

3.9.14 Binding Set Displays

Binding set displays denote sets of bindings (as a schema does when used as an expression). These terms usually arise as inferred generic actuals; for example, given $S \mathrel{\widehat=} [x, y : \mathbb{N}]$, the generic actual inferred for $\cup$ in the expression $S \cup S$ is $\lblot x, y : \mathbb{Z} \rblot$.

Syntax

expression-4	::=	`\lblot` binding `;` ... `;` binding `\rblot`
binding	::=	decl-name-list `:` expression

Domain Checking

$DC(\lblot n : e, \ldots \rblot) = DC(e) \land \cdots$

Proof Considerations

The alphabet of a binding set display must be the alphabet of some schema appearing in the specification. The significant facts about the binding set are derived when the schema is declared; see Section 3.5.7.

3.9.15 Tuples

Syntax

expression-4	::=	`(` expression-list `)`

Domain Checking

$DC((e, e', \ldots)) = DC(e) \wedge DC(e') \wedge \cdots$

Proof Considerations

There are no built-in facts about cross products and tuples; suitable rules can be added manually if need be. The Toolkit gives rules for two and three-element tuples.

3.9.16 Binding Formation

Syntax

expression-4	::=	`\theta` schema-name decoration [replacements]

Domain Checking

$DC(\theta S) = \mathit{true}$
$DC(\theta S[n := e, \ldots]) = DC(e) \wedge \cdots$

Proof Considerations

Several facts about theta terms are introduced when a schema is defined; see Section 3.5.7.

3.9.17 Component Selection

Syntax

expression-4	::=	expression-4 **.** var-name
	|	expression-4 **.** *number*

Domain Checking

$DC(e.n) = DC(e)$

Proof Considerations

Several facts about named component selections are introduced when a schema is defined; see Section 3.5.7.

There are no built-in facts about numeric selections; suitable rules can be added manually if need be. The Toolkit gives rules for selections from two and three-element tuples.

3.10 Identifiers

This section gives the syntax of the various kinds of identifiers, simple and compound, used in Z specifications, and the syntax of the various kinds of identifier lists.

3.10.1 Variable and Declaration Names

var-name	::=	ident | **(** op-name **)**
decl-name	::=	ident | op-name
decl-name-list	::=	decl-name **,** . . . **,** decl-name

3.10.2 Operator Names

op-name	::=	`_` in-sym decoration `_`
	|	pre-sym decoration `_`
	|	`_` post-sym decoration
	|	`_ \limg _ \rimg`
	|	`-` decoration
in-sym	::=	*in-fun* | *in-gen* | *in-rel*
pre-sym	::=	*pre-gen* | *pre-rel*
post-sym	::=	*post-fun*

The above syntax gives the rules for constructing operator names from operator symbols; in most cases, an escaped underscore (`_`) replaces the operands.

For op-names, if the symbol is an in-sym or pre-sym, there must be a space between the symbol and the trailing escaped underscore, if the symbol is an identifier and there is no decoration. This is because Z/EVES tries to scan as long a token as possible, and an escaped underscore is a valid character in an identifier.

3.10.3 Identifiers

ident	::=	*word* decoration
event-name	::=	schema-name | ident
theorem-name	::=	*word*
decoration	:=	[stroke ... stroke]
stroke	:=	`'` | `!` | `?`
	|	`_0` | `_1` | `_2` | `_3` | `_4` | `_5` | `_6` | `_7` | `_8` | `_9`
ident-list	::=	ident `,` ... `,` ident
gen-formals	::=	`[` ident-list `]`

An identifier consists of a word, optionally followed by a sequence of strokes (a decoration). Note that it would be valid LaTeX to write something like `x_{10}`, but not valid Z/LaTeX.

LaTeX gives an error on a double subscript, e.g., `x_1_2`, and typesets mixed subscripts and superscripts above one another, so that `x_1'` and `x'_1` both appear as x_1'. Thus, it is advisable to use `{}` between adjacent sub- or superscripts, e.g. `x_2{}_1{}'`. Z/EVES ignores the braces, and the typeset form (${x_{21}}'$) clearly indicates the order of strokes.

Event names are used in Z/EVES commands (Chapter 4) to denote names of declarations.

Chapter 4

Z/EVES Commands

Z/EVES is an interactive system. This chapter describes the Z/EVES interactive commands and their effects.

Both Z paragraphs and Z/EVES commands can be entered at the Z/EVES prompt. A paragraph is type-checked and, if well-typed, is added to the history. If the paragraph has a non-trivial domain checking condition, that condition is added as a proof goal; its name will end with $*domainCheck*.

4.1 Z Section Proofs

Z-section-proof	::=	`\begin{zsectionproof}{` name `}{` [name `,` ... `,` name] `}`
		decl-or-cmd ... decl-or-cmd
		`\end{zsectionproof}`

command	::=	`declare` event-name `,` ... `,` event-name
	|	`declare to` event-name
	|	`declare through` event-name
	|	`parent` ident

Once a section has been created (Section 3.4), the declarations in the section can be proved. This is done in a Z section proof. As with Z sections, the syntax looks like a declaration, but the begin and end lines are commands that begin and end the section proof, and items in the section proof are processed as they are read.

The `\begin{zsectionproof}` line specifies the name of the section to prove, and any additional parent sections (proof parents). It is not necessary to specify the parents of the section that were specified when the section was created.

When a section proof is begun, Z/EVES is reset, and the parents and proof parents of the specified section are loaded. Section files for parents and proof parents must exist, but section proof files are not required. If a section proof file exists for a parent, it is used along with the section file, and both the parent's parents and the parent's proof parents are checked for consistency and loaded, as described in Section 3.4. Otherwise, only the parent's parents are checked. Thus, section proofs, unlike sections, need not be done in bottom-up order. Section proof files are searched for in the same places as section files, as described in Section 3.4.

After the parents and proof parents of a section are loaded, the declarations are made available for processing. They are not actually declared, but are placed in a "pending" list. Declarations are moved from the pending list to the current development with one of the `declare` commands; these commands type check the specified declarations and generate proof obligations for them. If a declaration added by a `declare` command is undone, it is placed back on the pending list. The pending list may be viewed with the `print status` command.

At any time, new declarations may be made, and additional proof parents may be added with the `parent` command. Thus, the order of declarations in a section proof, except for the requirement that a name be declared before it is used, need have no relation to the order of declarations in the section being proved. Also, additional declarations required for a proof may be added before the proof where they are required.

When a section proof is ended, if there are no pending declarations or undischarged proof obligations, a section proof file is created in the current directory. If the name of the section is `S`, the name of the file will be `S.pct`. This file records the proof parents specified in the section proof, both in the `\begin{zsectionproof}` command and in `parent` commands. If the section proof is not finished, nothing happens. The `reset` command may be used if you want to discard the section proof.

4.1.1 Declare

command ::= `declare` event-name `,` `...` `,` event-name

The `declare` command declares the specified declarations, which must be on the pending list. The declarations are processed in the specified order, and removed from the pending list.

4.1.2 Declare To

command ::= `declare to` event-name

The `declare to` command declares all declarations on the pending list up to but not including the specified declaration, which must be on the pending list. The declarations are processed in the order in which they appear in the pending list, and are then removed from the list.

4.1.3 Declare Through

command ::= `declare through` event-name

The `declare through` command declares all declarations on the pending list up to and including the specified declaration, which must be on the pending list. The declarations are processed in the order in which they appear in the pending list, and are then removed from the list.

4.1.4 Parent

command ::= `parent` ident

The `parent` command declares an additional proof parent for the current Z section proof.

4.2 Printing Commands

Printing commands are used to display information.

4.2.1 Help

command ::= `help` ident

Error messages in Z/EVES are named, and the names are printed with the messages; the `help` command prints an explanation for the specified message.

4.2.2 Print Declaration

command ::= `print declaration` event-name

The `print declaration` command displays the declaration of a particular name. The paragraph containing the declaration is printed. The specified name may be the name of a global constant, theorem, or schema. The specified name may also be the name of a declaration on the pending list, if a Z section proof is currently in progress.

With the exception of theorems, it is not possible to print declarations from loaded sections, including the Toolkit.

4.2.3 Print Formula

command ::= `print formula`

The `print formula` command displays the current formula, if one exists.

4.2.4 Print History

command ::= `print history` [`summary`] [*number*]

The `print history` command displays the specified number of the most recently entered paragraphs. If the number is omitted, all entered paragraphs are printed. If the `summary` option is given, a short summary of the paragraphs is printed, rather than the paragraphs themselves.

4.2.5 Print Proof

command ::= `print proof` [`summary`]

The `print proof` command displays the proof steps used in the current proof attempt, along with the results of those steps if the `summary` option is omitted.

4.2.6 Print Status

command ::= `print status`

The `print status` command displays the names of all established proof goals. The goals are presented in three lists: the proved goals, the untried goals, and the unfinished goals (which have been tried, but not yet proved). During a Z section proof (Section 4.1), the command also displays the names of pending declarations.

4.2.7 Print Syntax

command ::= `print syntax` *word*

The `print syntax` command displays the syntax declaration used to define the specified *word*. Note that if the word was defined to have an operator class of `ignore`, the syntax declaration for it cannot be printed, since Z/EVES can no longer see it, and the command to print it cannot be syntactically correct. Operator classes are described in Section 3.5.11.

4.2.8 Searching for Theorems

command ::= [`print`] `theorems about expression` expression
| [`print`] `theorems about predicate` predicate

These commands search for theorems in the history that contain references to the specified expressions or predicates. For example, the command `theorems about expression X \fun Y;` prints the names of any theorems mentioning total functions; here X and Y are variables that can match any expression in the theorem. The `print` forms of the commands print the declarations of the theorems; if the word `print` is omitted, only the names of the theorems are printed.

4.3 Undoing Declarations

The undoing commands cause some or all of the most recently added declarations to be removed. Except for the `reset` command, an undoing command prints a short description of the removed declaration(s).

For the `undo` commands, if the undone declarations were declared with one of the `declare` commands (Section 4.1), they are placed back on the pending list. Otherwise, undone declarations are lost; the only way to get them back is to present the declarations again to Z/EVES.

4.3.1 Reset

command ::= `reset`

The `reset` command returns Z/EVES to its initial state. If the Toolkit section was present before the command was entered, it will still be available afterwards.

4.3.2 Undo

command ::= `undo` [*number*]

The `undo` command removes the specified number of most recently added declarations. If no number is given, the most recent declaration is removed.

4.3.3 Undo Back To

command ::= `undo back to` event-name

The `undo back to` command removes all declarations up to but not including the declaration with the specified name.

4.3.4 Undo Back Through

command ::= `undo back through` event-name

The `undo back to` command removes all declarations up to and including the declaration with the specified name.

4.4 Interface Commands

4.4.1 Check

command ::= `check` *string*

The `check` command reads input from the specified file in batch mode, and performs type checking only. No declarations are sent to EVES, and no proof obligations are generated. If the file contains a Z section declaration (Section 3.4), a section file for the Z section is created. If the file contains `\input` or `\include` commands (Section 3.5.12), the specified files are also read and checked.

The `check` command resets Z/EVES before and after checking the file.

4.4.2 Quit

command ::= `quit`

The `quit` command terminates Z/EVES.

4.4.3 Read

command ::= `read` [`script`] *string*

The `read` command reads input from the specified file, in batch mode. If the `script` option is given, the file is read in interactive mode.

4.4.4 Ztags

command ::= `ztags` *string* , ... , *string*

The `ztags` command generates a "tags" file for the specified files. When editing Z specification files with the GNU Emacs editor and the Emacs Z mode provided with the Z/EVES system, the tags file provides information to the editor that allows searching for definitions in the specified files, and a number of other useful features.

The specified file names may contain system-specific "wild card" characters, and the tags file produced is named `TAGS`, in the current directory.

Chapter 5

Proof Commands

In interactive use, the Z/EVES system maintains a *current goal*. The current goal is set when a Z paragraph with a non-trivial domain condition or a theorem paragraph is entered, or when one of the `try` commands is entered. The proof commands perform various transformations on the current goal, and the current goal may be examined with the `print formula` command. A proof is complete when the current goal is the predicate *true*.

Several proof commands contain predicates or expressions. Any such formulas are type checked in the context of the goal, so that the types of the variables appearing in the goal are known, and generic actuals can usually be inferred by the type checker. However, there is a slight weakness in the type checker's handling of a theorem's generic formals, and any types referring to these formals will not be inferred.

5.1 Starting Proofs

Named proof goals are established by `theorem` declarations or by domain checking conditions for paragraphs. The `try` command establishes an unnamed proof goal.

When the current goal is named, entering a declaration or one of the `try` commands causes Z/EVES to save the current goal and current proof, i.e., the goal is deferred. If the new input results in a new goal, that becomes the current goal. Otherwise, there is no current goal. There is also no current goal after one of the `undo` commands is entered.

A deferred goal may be returned to, if the associated declaration has not been undone, and the `print status` command can be used to display the status of all named goals.

If the current goal is unnamed, a declaration or `try` or `undo` command causes the current goal to be lost.

5.1.1 Try

command ::= `try` predicate

The `try` command establishes a new, unnamed, proof goal. The given predicate is type checked, but references to free variables are allowed. However, if there is not enough information in the predicate to determine the types of these variables (e.g., membership in some defined set), it is unlikely that a proof of the predicate will be successful.

5.1.2 Try Lemma

command ::= `try lemma` event-name

The `try lemma` command begins the proof, or returns to an unfinished proof, of a named proof goal.

5.2 Undoing Proof Steps

Proof undo commands are used to return to a previous state in a proof attempt. Proof steps that are undone are lost; the only way to return to the state before the proof undo command was entered is to redo the proof commands. Do not confuse the `back` command, which discards proof steps, with the `undo` command, which discards paragraphs!

5.2.1 Back

command ::= `back` [*number*]

The `back` command discards the specified number (or one, if no number is given) of proof steps.

5.2.2 Retry

command ::= `retry`

The `retry` command discards the current proof attempt, and restores the proof goal to its original form.

5.3 Case Splitting

Complex goals can often be split into independent cases, and those cases proved separately. The `cases` command is used to split a goal into subgoals. The `next` command is used to move from one subgoal to the next. The `split` command splits the current goal into two cases: one in which a specified predicate is true, and one in which it is false. The `conjunctive` and `disjunctive` commands can sometimes be used to transform the current goal into a form more amenable to case splitting.

5.3.1 Cases

command ::= `cases`

The `cases` command splits the goal into separate cases, if possible. A case split is possible if the goal is a conjunction, disjunction, a conditional predicate (of the form **if** ... **then** ... **else** ...), or an implication whose conclusion is one of these.

Each case is numbered, and the highest numbered case becomes a new proof goal. Different cases can be considered by using the `next` command.

5.3.2 Next

command ::= `next`

The `next` command moves to the next case, if a case split has been performed. If there are no further cases, the final form of each of the goal cases are collected. It is not necessary to complete the proof of a case before moving on to the next.

5.3.3 Split

command ::= `split` predicate

The `split` command is used to consider two cases. The command `split` P transforms a goal G to the new goal **if** P **then** G **else** G. (The `cases` command can be applied to this new goal to break it into two cases: $P \Rightarrow G$ and $\neg P \Rightarrow G$.)

The specified predicate is type checked with respect to the current goal.

5.3.4 Conjunctive

command ::= `conjunctive`

The `conjunctive` command performs the following transformations on the current goal:

- Implication, logical equivalence, and conditional predicates are replaced by $\wedge$ and $\vee$, (e.g., $a \Rightarrow b$ becomes $(\neg a) \vee b$).
- The scope of negations is minimized (e.g., $\neg (\forall D \bullet P)$ becomes $\exists D \bullet \neg P$).
- $\vee$ is distributed across $\wedge$ (e.g., $a \vee (b \wedge c)$ becomes $(a \vee b) \wedge (a \vee c)$).

5.3.5 Disjunctive

command ::= `disjunctive`

The `disjunctive` command performs the following transformations on the current goal:

- Implication, logical equivalence, and conditional predicates are replaced by $\wedge$ and $\vee$, (e.g., $a \Rightarrow b$ becomes $(\neg a) \vee b$).
- The scope of negations is minimized (e.g., $\neg (\forall D \bullet P)$ becomes $\exists D \bullet \neg P$).
- $\wedge$ is distributed across $\vee$ (e.g., $a \wedge (b \vee c)$ becomes $(a \wedge b) \vee (a \wedge c)$).

5.4 Using Theorems and Definitions

Theorems and definitions can be used manually, in the `apply`, `use`, and `invoke` commands, or can be used automatically by the reduction commands (Section 5.7). In order to use a theorem automatically, it must be declared with a usage as described in Section 3.5.10. Note that labelled predicates (Section 3.5.6) are added to the system as theorems.

5.4.1 Apply

command ::= `apply` theorem-name
| `apply` theorem-name `to expression` expression
| `apply` theorem-name `to predicate` predicate

The specified name must be the name of a theorem that was declared as a rewrite rule. If no expression or predicate is specified, the rule is applied wherever possible in the current goal. If an expression or predicate is specified, the rule is applied, if possible, only to occurrences of the expression or predicate in the current goal.

5.4.2 Invoke

command	::=	`invoke` [event-name]

If the specified name is the name of a schema or of a name introduced in a definition, all occurrences of the name in the current goal are replaced by its definition. If no name is specified, all schema and definition names in the current goal are invoked.

The specified name must be either

- a schema name, which may have a prefix but must not have decorations, generic actuals, or replacements, or
- a global reference, which may have generic actuals.

5.4.3 Invoke Predicate

command	::=	`invoke predicate` predicate

All occurrences of the specified predicate in the current goal are invoked. The difference between this command and the `invoke` command can be illustrated with the following example. Suppose a schema `S` with a predicate has been declared, and the current goal contains references to both `S` and `S'`. The command `invoke S` will replace the occurrences of both `S` and `S'` with the schema's predicate; in the replacement for `S'`, the variables in the predicate will be decorated. The command `invoke predicate S` will invoke only the occurrence of `S` in the goal, and the the command `invoke predicate S'` will invoke only the occurrence of `S'`. Note that the command `invoke S'` would be incorrect.

5.4.4 Use

command	::=	`use` theorem-ref
theorem-ref	::=	theorem-name decoration [gen-actuals] [replacements]
theorem-name	::=	*word*

The specified name must be the name of a theorem, and generic actuals must be supplied for all generic formals of the theorem. The generic actuals are type checked with respect to the current goal.

Leading universal quantifiers in the theorem's predicate are stripped, and the variables bound by these quantifiers become free in the theorem, together with variables in the theorem that were already free. Each free variable in the theorem must be instantiated, i.e., given a value, in one of two ways:

- The value e of a free variable x is specified as a rename or replacement: e/x or $x := e$. The syntax of renames and replacements requires that the instantiation be a replacement if e is not a variable.
- A free variable x is also free in the current goal or is globally defined; this is equivalent to the replacement $x := x$.

If a decoration is specified in the theorem reference, the decoration is appended to all free variables of the theorem. The resulting variables must then be instantiated as described above.

Z/EVES then adds the theorem to the current goal, with free variables in the theorem replaced by their values, and type checks the result.

5.5 Using Equalities

5.5.1 Equality Substitute

command	::=	`equality substitute` expression

If an expression e is specified, and there is an hypothesis in the goal that is an equality between e and some other expression e', then any subsequent occurrences of e in the goal are replaced by e'.

If no expression is given, equality hypotheses in the goal are heuristically chosen (if possible) and replacements are performed as described above.

5.6 Quantifiers

5.6.1 Instantiate

command	::=	`instantiate` instantiations
instantiations	::=	let-def `,` ... `,` let-def
let-def	::=	var-name `==` expression

The `instantiate` command is used to instantiate one or more quantified variables. The variables instantiated must appear together in a quantified predicate (that is, one cannot instantiate both x and y in $\forall x : \mathbb{Z} \mid x > 1 \bullet \forall y : \mathbb{Z} \bullet \ldots$, but can do so in $\forall x, y : \mathbb{Z} \bullet \ldots$).

The expressions in the command are type checked with respect to the current goal.

5.6.2 Prenex

command	::=	`prenex`

The `prenex` command removes any quantifiers that can be made into leading universal quantifiers. For example, the predicate

$$(\exists x : \mathbb{N} \mid x \in S) \Rightarrow (\forall y : S' \bullet y < x)$$

is equivalent to the predicate

$$\forall x : \mathbb{N};\ y : S' \bullet x \in S \Rightarrow y < x.$$

When the quantifiers are removed, the declarations are converted to predicates (thus, colons are converted to epsilons). Thus, applying `prenex` to the original predicate results in the goal

$$x \in \mathbb{N} \land x \in S \land y \in S' \Rightarrow y < x.$$

Note: Prenexing is highly recommended whenever possible.

5.7 Reduction

The reduction commands traverse the current goal, accumulating assumptions and performing reduction on predicates and expressions in the goal. In a traversal, each formula and subformula of the goal is examined, and may be replaced by a logically equivalent formula which Z/EVES considers "simpler." Other replacements may occur, depending on the type of reduction being performed.

Z/EVES can perform three types of reduction: *simplification*, *rewriting*, and *reducing*. The reduction commands come in two flavours: the "regular" reduction commands, and the "trivial" reduction commands, which are less powerful, but faster. There is no command for "trivial" reducing; this would simply invoke all definitions in the current goal, which can be done with the `invoke` command (Section 5.4.2).

The reduction commands use only theorems and definitions that are *enabled*. The declaration of a name or theorem may specify that the name should normally be disabled. A command modifier can be used to enable or disable a theorem or definition for a single reduction command; see Section 5.8.

The effectiveness of the Z/EVES reduction commands depends to some extent on the order of hypotheses in the current goal. The `rearrange` command may be used to rearrange the hypotheses to improve the performance of the reduction commands.

The `prove by reduce` and `prove by rewrite` commands repeatedly reduce or rewrite the current goal, until the goal is unchanged.

5.7.1 Simplify

command ::= `simplify`

The `simplify` command performs simplification on the current goal.

In simplification, Z/EVES performs equality and integer reasoning, propositional reasoning and tautology checking, and applies forward rules and assumption rules where possible. A forward rule can be applied if its hypothesis matches a subformula; its conclusion is then assumed in the proof. An assumption rule can be applied if its trigger matches a subformula and its condition, if any, can be proved with simplification; its conclusion is then assumed in the proof. See Section 3.5.10.3 for a description of forward rules, and Section 3.5.10.4 for a description of assumption rules.

5.7.2 Rewrite

command ::= `rewrite`

The `rewrite` command performs rewriting on the current goal.

In rewriting, Z/EVES performs simplification, and also applies rewrite rules where possible. A rewrite rule can be applied if its pattern matches a subformula and its condition, if any, can be proved with rewriting; the subformula is replaced with the replacement of the rule and the result will again be rewritten. See Section 3.5.10.2 for a description of rewrite rules.

For example, suppose the following rule has been declared:

theorem rule crossNull
$\{\} \times Y = \{\}$

and the current goal is $S = \{\} \times T$. A `rewrite` command would apply this rule, and the current goal would become $S = \{\}$.

5.7.3 Reduce

command ::= `reduce`

The `reduce` command performs reducing on the current goal.

In reducing, Z/EVES performs simplification and rewriting, and if a subformula is a schema reference or generic instance, the subformula will be replaced by the definition of the schema or generic symbol (Section 5.4.2) and the result will again be reduced. Also, conditional rewrite rules apply if their conditions can be shown to be true by reducing.

5.7.4 Trivial Simplify

command ::= `trivial simplify`

The `trivial simplify` command performs a limited form of simplification on the current goal.

In trivial simplification, only propositional reasoning and tautology checking is performed. Equality reasoning and the application of forward and assumption rules are not done.

5.7.5 Trivial Rewrite

command ::= `trivial rewrite`

The `trivial rewrite` command performs a limited form of rewriting on the current goal.

In trivial rewriting, only rewriting with unconditional rewrite rules is performed. Simplification is not done.

5.7.6 Rearrange

command ::= `rearrange`

If the current goal is of the form $H_1 \wedge H_2 \wedge \ldots \Rightarrow C$, the `rearrange` command rearranges the hypotheses so that those that are "simpler," in some sense, appear before more complicated ones. This often improves the effectiveness of reduction commands.

5.7.7 Prove By Reduce

command ::= `prove by reduce`

The `prove by reduce` command performs repeated reducing on the current goal, until reduction has no effect. In each iteration, this command does the following:

1. The current goal is prenexed (Section 5.6.2).
2. The current goal is rearranged.
3. Equality substitution is performed (Section 5.5.1).
4. The current goal is reduced.

5.7.8 Prove By Rewrite

command ::= `prove by rewrite`

The `prove by rewrite` command performs repeated rewriting on the current goal, until rewriting has no effect. In each iteration, this command does the following:

1. The current goal is prenexed (Section 5.6.2).
2. The current goal is rearranged.
3. Equality substitution is performed (Section 5.5.1).
4. The current goal is rewritten.

5.8 Modifiers

Modifiers temporarily affect the behaviour of a reduction command.

5.8.1 With Disabled

command ::= `with disabled (` event-name-list `)` command

The declarations specified by the list of names are temporarily disabled, and the proof command is performed. Disabled declarations are effectively ignored during proof; disabled forward, assumption, and rewrite rules are not applied during reduction, and disabled definitions are not invoked.

Disabling an already-disabled declaration has no effect.

5.8.2 With Enabled

command ::= `with enabled (` event-name-list `)` command

The declarations specified by the list of names are temporarily enabled, and the proof command is performed.

Enabling an already-enabled declaration has no effect.

5.8.3 With Expression

command ::= `with expression (` expression `)` command

The proof command is performed on the specified expression, which must appear in the current goal.

5.8.4 With Predicate

command ::= `with predicate (` predicate `)` command

The proof command is performed on the specified predicate, which must appear in the current goal.

5.8.5 With Normalization

command ::= `with normalization` command

Internally, Z/EVES represents propositions in “if-form,” in which all logical connectives are represented as conditional expressions. For example, $a \wedge b$ is represented as **if** a **then** b **else** *false*. During reduction, if a conditional expression whose test is itself a conditional expression is encountered, Z/EVES can “normalize” the outer conditional by pushing the inner conditional into the arms of the outer conditional. For example, the expression

$$\textbf{if } (\textbf{if } a \textbf{ then } b \textbf{ else } c) \textbf{ then } d \textbf{ else } e$$

would be normalized to

$$\textbf{if } a \textbf{ then } (\textbf{if } b \textbf{ then } d \textbf{ else } e) \textbf{ else } (\textbf{if } c \textbf{ then } d \textbf{ else } e).$$

This normalization increases the power of the prover, but since it involves repeating parts of a formula, the current goal can become extremely large, so normalization is disabled by default. The `with normalization` command enables normalization during performance of the specified command.

The extra power that normalization gives Z/EVES is especially noticeable when, for example, the conclusion of the current goal is a disjunction. A good strategy in this case is to do what reduction you can without normalization (the default), and then do a final simplification step with normalization.

Bibliography

[1] R. D. Arthan. *Modules for Z.* Proposal to ISO JTC1/SC22/WG19, 11 May 1995.

[2] D. Craigen, S. Kromodimoeljo, I. Meisels, W. Pase and M. Saaltink. EVES: An Overview. In *Proceedings of VDM '91 (Formal Software Development Methods)*, Noordwijkerhout, The Netherlands (October 1991), Lecture Notes in Computer Science 551, Springer-Verlag, Berlin, 1991.

[3] Sentot Kromodimoeljo, Bill Pase, Mark Saaltink, Dan Craigen and Irwin Meisels. The EVES System. In *Functional Programming, Concurrency, Simulation and Automated Reasoning*, Lecture Notes in Computer Science 693, Springer-Verlag, Berlin, 1993.

[4] Leslie Lamport. *LaTeX: A Document Preparation System, Second Edition.* Addison-Wesley, 1986.

[5] Irwin Meisels. *Software Manual for Unix Z/EVES Version 1.5.* ORA Canada TR-97-6028-01c, September 1997.

[6] Irwin Meisels. *Software Manual for Windows Z/EVES Version 1.5 and the Z Browser.* ORA Canada TR-97-5505-04e, September 1997.

[7] Mark Saaltink. *The Z/EVES Mathematical Toolkit Version 2.2.* ORA Canada Technical Report TR-97-5493-05a, September 1997.

[8] Mark Saaltink. *The Z/EVES User's Guide.* ORA Canada Technical Report TR-97-5493-06, September 1997.

[9] J.M. Spivey. *The Z Notation: A Reference Manual, Second Edition.* Prentice Hall, 1992.

[10] J.M. Spivey. *The fuzz Manual, Second Edition.* J.M. Spivey Computing Science Consultancy, May 1993.

Appendix A

Collected Syntax

Specifications

<table>
<tr><td>specification</td><td>::=</td><td>spec-item . . . spec-item</td></tr>
<tr><td>spec-item</td><td>::=</td><td>z-section</td></tr>
<tr><td></td><td>|</td><td>z-paragraph</td></tr>
<tr><td></td><td>|</td><td>syntax-decl</td></tr>
<tr><td></td><td>|</td><td>input-cmd</td></tr>
<tr><td>interaction</td><td>::=</td><td>decl-or-cmd . . . decl-or-cmd</td></tr>
<tr><td>decl-or-cmd</td><td>::=</td><td>spec-item</td></tr>
<tr><td></td><td>|</td><td>z-section-proof</td></tr>
<tr><td></td><td>|</td><td>command ;</td></tr>
</table>

Z Sections

<table>
<tr><td>Z-section</td><td>::=</td><td>\begin{zsection}{ ident }{ [ident , . . . , ident] }
section-item . . . section-item
\end{zsection}</td></tr>
<tr><td>section-item</td><td>::=</td><td>z-paragraph</td></tr>
<tr><td></td><td>|</td><td>syntax-decl</td></tr>
<tr><td></td><td>|</td><td>input-cmd</td></tr>
</table>

Paragraphs

<table>
<tr><td>z-paragraph</td><td>::=</td><td>zed-box</td></tr>
<tr><td></td><td>|</td><td>schema-box</td></tr>
<tr><td></td><td>|</td><td>axiomatic-box</td></tr>
<tr><td></td><td>|</td><td>generic-box</td></tr>
<tr><td></td><td>|</td><td>theorem</td></tr>
<tr><td>zed-box</td><td>::=</td><td>\begin[para-opt]{zed}
zed-box-item sep . . . sep zed-box-item
\end{zed}</td></tr>
<tr><td></td><td>|</td><td>\begin[para-opt]{syntax}
zed-box-item sep . . . sep zed-box-item
\end{syntax}</td></tr>
</table>

```
schema-box               ::= \begin[para-opt]{schema}{ schema-name } [gen-formals]
                               decl-part
                             [\where
                               axiom-part]
                             \end{schema}

axiomatic-box            ::= \begin[para-opt]{axdef}
                               decl-part
                             [\where
                               labelled-axiom-part]
                             \end{axdef}

generic-box              ::= \begin[para-opt]{gendef} [gen-formals]
                               decl-part
                             [\where
                               labelled-axiom-part]
                             \end{gendef}

theorem                  ::= \begin[para-opt]{theorem}{[usage] theorem-name}[gen-formals]
                               predicate
                             [ \proof
                               command sep ... sep command ]
                             \end{theorem}

zed-box-item             ::= given-set-definition
                          |  schema-definition
                          |  abbreviation-definition
                          |  free-type-definition
                          |  labelled-predicate

given-set-definition     ::= [ ident-list ]

schema-definition        ::= schema-name [gen-formals] \defs schema-exp

abbreviation-definition  ::= def-lhs == expression

free-type-definition     ::= ident ::= branch | ... | branch

labelled-predicate       ::= [label] predicate

def-lhs                  ::= var-name [gen-formals]
                          |  pre-gen decoration ident
                          |  ident in-gen decoration ident

branch                   ::= ident
                          |  var-name \ldata expression \rdata

decl-part                ::= basic-decl sep ... sep basic-decl

axiom-part               ::= predicate sep ... sep predicate

labelled-axiom-part      ::= labelled-predicate sep ... sep

para-opt                 ::= [ ability ]

label                    ::= \Label{ [ability] [usage] theorem-name }

ability                  ::= enabled | disabled

usage                    ::= axiom | rule | grule | frule

sep                      ::= \\ | ; | \also
```

Syntax Declarations

syntax-decl	::=	`\syndef{`string-or-word`}{`opclass`}{`[string-or-word]`}`
string-or-word	::=	*string* | *word*
opclass	::=	`infun1` | `infun2` | `infun3` | `infun4` | `infun5` | `infun6`
	|	`ingen` | `pregen`
	|	`inrel` | `prerel`
	|	`postfun`
	|	`word`
	|	`ignore`

Input Commands

input-cmd	::=	`\input{`text`}`
	|	`\include{`text`}`

Declarations

declaration	::=	basic-decl `;` ... `;` basic-decl
basic-decl	::=	decl-name-list `:` expression
	|	schema-ref
decl-name-list	::=	decl-name `,` ... `,` decl-name

Schema Expressions

schema-exp	::=	`\forall` schema-text `@` schema-exp
	|	`\exists` schema-text `@` schema-exp
	|	`\exists_1` schema-text `@` schema-exp
	|	schema-exp-1
schema-exp-1	::=	`[` schema-text `]`
	|	schema-ref
	|	`\lnot` schema-exp-1
	|	`\pre` schema-exp-1
	|	schema-exp-1 `\land` schema-exp-1
	|	schema-exp-1 `\lor` schema-exp-1
	|	schema-exp-1 `\implies` schema-exp-1
	|	schema-exp-1 `\iff` schema-exp-1
	|	schema-exp-1 `\project` schema-exp-1
	|	schema-exp-1 `\hide` `(` decl-name-list `)`
	|	schema-exp-1 `\semi` schema-exp-1
	|	schema-exp-1 `\pipe` schema-exp-1
	|	`(` schema-exp-1 `)`
schema-text	::=	declaration [`|` predicate]
schema-ref	::=	schema-name decoration [gen-actuals] [replacements]
schema-name	::=	[prefix] *word*
prefix	::=	`\Delta` | `\Xi`
replacements	::=	`[` rename-or-repl `,` ... `,` rename-or-repl `]`
rename-or-repl	::=	decl-name `/` decl-name
	|	decl-name `:=` expression

Predicates

```
predicate        ::=  \forall schema-text @ predicate
                   |  \exists schema-text @ predicate
                   |  \exists_1 schema-text @ predicate
                   |  \LET let-def ; ... ; let-def @ predicate
                   |  \IF predicate \THEN predicate \ELSE predicate
                   |  predicate-1

predicate-1      ::=  expression rel expression rel ... rel expression
                   |  pre-rel decoration [gen-actuals] expression
                   |  schema-ref
                   |  \pre schema-ref
                   |  \lnot predicate-1
                   |  predicate-1 \land predicate-1
                   |  predicate-1 \lor predicate-1
                   |  predicate-1 \implies predicate-1
                   |  predicate-1 \iff predicate-1
                   |  true
                   |  false
                   |  ( predicate )

let-def          ::=  var-name == expression

rel              ::=  = | \in | in-rel decoration [gen-actuals] | \inrel{ ident }
```

Expressions

```
expression-0     ::=  \lambda schema-text @ expression
                   |  \mu schema-text [@ expression]
                   |  \LET let-def ; ... ; let-def @ expression
                   |  expression

expression       ::=  \IF predicate \THEN expression \ELSE expression
                   |  expression-1

expression-1     ::=  expression-1 in-gen decoration expression-1
                   |  expression-2 \cross ... \cross expression-2
                   |  expression-2

expression-2     ::=  expression-2 in-fun decoration [gen-actuals] expression-2
                   |  pre-gen decoration expression-4
                   |  - decoration [gen-actuals] expression-4
                   |  expression-4 \limg expression-0 \rimg decoration [gen-actuals]
                   |  expression-3

expression-3     ::=  expression-3 expression-4
                   |  expression-4
```

expression-4	::=	[local-or-global] var-name [gen-actuals]
	|	*number*
	|	schema-ref
	|	`\{` schema-text [`@` expression] `\}`
	|	`\{` [expression-list] `\}`
	|	`\langle` expression-list `\rangle`
	|	`\lbag` expression-list `\rbag`
	|	`\lblot` binding `;` ... `;` binding `\rblot`
	|	`(` expression-list `)`
	|	`\theta` schema-name decoration [replacements]
	|	expression-4 `.` var-name
	|	expression-4 `.` *number*
	|	expression-4 *post-fun* decoration [gen-actuals]
	|	expression-4 `\bsup` expression `\esup`
	|	expression-4 `^{` expression `}`
	|	`(` expression-0 `)`
local-or-global	::=	`\Local` | `\Global`
gen-actuals	::=	`[` expression-list `]`
expression-list	::=	expression `,` ... `,` expression
binding	::=	decl-name-list `:` expression

Names

var-name	::=	ident | `(` op-name `)`
decl-name	::=	ident | op-name
decl-name-list	::=	decl-name `,` ... `,` decl-name
op-name	::=	`_` in-sym decoration `_`
	|	pre-sym decoration `_`
	|	`_` post-sym decoration
	|	`_` `\limg` `_` `\rimg`
	|	`-` decoration
in-sym	::=	*in-fun* | *in-gen* | *in-rel*
pre-sym	::=	*pre-gen* | *pre-rel*
post-sym	::=	*post-fun*
ident	::=	*word* decoration
event-name	::=	schema-name | ident
theorem-name	::=	*word*
decoration	:=	[stroke ... stroke]
stroke	:=	`'` | `!` | `?`
	|	`_0` | `_1` | `_2` | `_3` | `_4` | `_5` | `_6` | `_7` | `_8` | `_9`
ident-list	::=	ident `,` ... `,` ident
event-name-list	::=	event-name `,` ... `,` event-name
gen-formals	::=	`[` ident-list `]`

Z Section Proofs

Z-section-proof	::=	`\begin{zsectionproof}{` ident `}{` [ident `,` ... `,` ident] `}`
		decl-or-cmd ... decl-or-cmd
		`\end{zsectionproof}`

command	::=	`declare` event-name , . . . , event-name
	|	`declare to` event-name
	|	`declare through` event-name
	|	`parent` ident
	|	`zsection path` [*string* , . . . , *string*]

Printing Commands

command	::=	`help` ident
	|	`print declaration` event-name
	|	`print formula`
	|	`print history` [`summary`] [*number*]
	|	`print proof` [`summary`]
	|	`print status`
	|	`print syntax` *word*
	|	[`print`] `theorems about expression` expression
	|	[`print`] `theorems about predicate` predicate

Undoing Commands

command	::=	`reset`
	|	`undo` [*number*]
	|	`undo back to` event-name
	|	`undo back through` event-name

Interface Commands

command	::=	`check` *string*
	|	`quit`
	|	`read` [`script`] *string*
	|	`ztags` *string* , . . . , *string*

Proof Commands

command	::=	`apply` theorem-name
	|	`apply` theorem-name `to expression` expression
	|	`apply` theorem-name `to predicate` predicate
	|	`back` [*number*]
	|	`cases`
	|	`conjunctive`
	|	`disjunctive`
	|	`equality substitute` expression
	|	`instantiate` instantiations
	|	`invoke` [event-name]
	|	`invoke predicate` predicate
	|	`next`
	|	`prenex`
	|	`prove by reduce`
	|	`prove by rewrite`
	|	`rearrange`
	|	`reduce`
	|	`retry`
	|	`rewrite`
	|	`simplify`
	|	`split` predicate
	|	`trivial rewrite`
	|	`trivial simplify`
	|	`try` predicate
	|	`try lemma` event-name
	|	`use` theorem-ref
	|	`with disabled (` event-name-list `)` command
	|	`with enabled (` event-name-list `)` command
	|	`with expression (` expression `)` command
	|	`with normalization` command
	|	`with predicate (` predicate `)` command
instantiations	::=	let-def `,` . . . `,` let-def
theorem-ref	::=	*word* decoration [gen-actuals] [replacements]

Appendix B

Collected Domain Checks

Notation

Symbol	Grammatical type
e	expression
P, Q	predicate
ST	schema-text
SE	schema-exp
D	decl-part
n, v, N, X	name
PR	prefix relation symbol
IR	infix relation symbol
PG	infix generic symbol
IG	infix generic symbol
F	infix function symbol

The descriptions use tables to define a function DC from Z phrases to Z predicates, which gives the domain checking condition for a phrase.

Paragraphs

Paragraph	**Domain check**
$[N, \ldots]$	*true*
$N ::= a \mid b\langle\!\langle e \rangle\!\rangle$	$DC(e)$
$n[X] == e$	$DC(e)$
D ——— P Q $\ldots$	$DC(D) \land (\forall D \bullet DC(P) \land DC(Q) \land \cdots)$
$[X]$ D ——— P Q $\ldots$	$DC(D) \land (\forall D \bullet DC(P) \land DC(Q) \land \cdots)$
S D ——— P	$DC(D) \land (\forall D \bullet DC(P))$
$S \mathrel{\widehat=} SE$	$DC(SE)$
theorem n $\ldots$	no condition

Schema Expressions

Schema expression	**Domain check**
$\forall D \mid P \bullet SE$	$DC(D) \land (\forall D \bullet DC(P)) \land DC(SE)$
$\exists D \mid P \bullet SE$	$DC(D) \land (\forall D \bullet DC(P)) \land DC(SE)$
$\exists_1 D \mid P \bullet SE$	$DC(D) \land (\forall D \bullet DC(P)) \land DC(SE)$
$[D \mid P]$	$DC(D) \land (\forall D \bullet DC(P))$
$\lnot SE$	$DC(SE)$
$\mathrm{pre}\ SE$	$DC(SE)$
$SE \land SE'$	$DC(SE) \land DC(SE')$
$SE \lor SE'$	$DC(SE) \land DC(SE')$
$SE \Rightarrow SE'$	$DC(SE) \land DC(SE')$
$SE \Leftrightarrow SE'$	$DC(SE) \land DC(SE')$
$SE \upharpoonright SE'$	$DC(SE) \land DC(SE')$
$SE \setminus (n, \ldots)$	$DC(SE)$
$SE \mathbin{⨾} SE'$	$DC(SE) \land DC(SE')$
$SE \gg SE'$	$DC(SE) \land DC(SE')$
(SE)	$DC(SE)$

Declarations

Declaration	**Domain check**
$n, \ldots : e$	$DC(e)$
$S[e, \ldots]$	$DC(e) \land \cdots$
$D;\ D'$	$DC(D) \land DC(D')$

Predicates

Predicate	**Domain check**
$\forall D \mid P \bullet Q$	$DC(D) \land (\forall D \bullet DC(P) \land (P \Rightarrow DC(Q)))$
$\exists D \mid P \bullet Q$	$DC(D) \land (\forall D \bullet DC(P) \land (P \Rightarrow DC(Q)))$
$\exists_1 D \mid P \bullet Q$	$DC(D) \land (\forall D \bullet DC(P) \land (P \Rightarrow DC(Q)))$
$\mathbf{let}\ x == e;\ \ldots \bullet P$	$DC(e) \land \cdots \land (\mathbf{let}\ x == e;\ \ldots \bullet DC(P))$
$\mathbf{if}\ P\ \mathbf{then}\ Q\ \mathbf{else}\ R$	$DC(P) \land (\mathbf{if}\ P\ \mathbf{then}\ DC(Q)\ \mathbf{else}\ DC(R))$
$e_1\ IR\ e_2 \ldots$	$DC(e_1) \land DC(IR) \land DC(e_2) \cdots$
$PR\ e$	$DC(e) \land DC(PR)$
$true$	$true$
$false$	$true$
$\neg P$	$DC(P)$
$P \land Q$	$DC(P) \land (P \Rightarrow DC(Q))$
$P \lor Q$	$DC(P) \land (P \lor DC(Q))$
$P \Rightarrow Q$	$DC(P) \land (P \Rightarrow DC(Q))$
$P \Leftrightarrow Q$	$DC(P) \land DC(Q)$
(P)	$DC(P)$

Expressions

Expression	**Domain check**
$\lambda D \mid P \bullet e$	$DC(D) \land (\forall D \bullet DC(P) \land (P \Rightarrow DC(e)))$
$\mu D \mid P \bullet e$	$DC(D) \land (\forall D \bullet DC(P) \land (P \Rightarrow DC(e))) \land (\exists_1 D \bullet P)$
$\mathbf{let}\ x == e;\ \ldots \bullet e'$	$DC(e) \land \cdots \land (\mathbf{let}\ x == e;\ \ldots \bullet DC(e'))$
$\mathbf{if}\ P\ \mathbf{then}\ e\ \mathbf{else}\ e'$	$DC(P) \land (\mathbf{if}\ P\ \mathbf{then}\ DC(e)\ \mathbf{else}\ DC(e'))$
$e\ IG\ e'$	$DC(e) \land DC(e')$
$PG\ e$	$DC(e)$
$e \times e' \times \cdots$	$DC(e) \land DC(e') \land \cdots$
$f(a)$	$\begin{cases} \mathit{true} & \text{if } f \text{ is a total function} \\ a \in \operatorname{dom} f & \text{if } f \text{ is a partial function} \\ f\ \mathit{applies\$to}\ a & \text{otherwise} \end{cases}$
$e\ F\ e'$	$DC((_F_)(e, e'))$
$\{e, e', \ldots\}$	$DC(e) \land DC(e') \land \cdots$
$\{\, D \mid P \bullet e \,\}$	$DC(D) \land (\forall D \bullet DC(P) \land (P \Rightarrow DC(e)))$
$\langle e, e', \ldots \rangle$	$DC(e) \land DC(e') \land \cdots$
$[\![e, e', \ldots]\!]$	$DC(e) \land DC(e') \land \cdots$
$(e, e', \ldots)$	$DC(e) \land DC(e') \land \cdots$
$\langle\!\vert n : e, \ldots \vert\!\rangle$	$DC(e) \land \cdots$
θS	true
$\theta S[n := e, \ldots]$	$DC(e) \land \cdots$
$e.n$	$DC(e)$
$v[e_1, \ldots]$	$DC(e_1) \land \ldots$
(e)	$DC(e)$

Appendix C

Differences Between Z/EVES and *fuzz*

The language accepted by Z/EVES is almost a superset of the language accepted by the *fuzz* [10] tool; a few features of the *fuzz* language are not supported, and a number of extensions have been added.

The following features of the *fuzz* language are not supported:

- The *fuzz* `%%type` and `%%tame` directives are not currently supported. The other directives are supported, but the `\syndef` declaration should be used instead. *fuzz* also accepts some undocumented directives (e.g., `%%token`) which are not supported.
- Quoted identifiers (e.g., `"FOO_BAR"`) are not supported; this would not allow the implementation of strings.

The following extensions have been added:

- There is a `theorem` paragraph for stating propositions and their proofs.
- A number of proof commands and other interactive commands are supported.
- Support for Z sections and Z section proofs has been added.
- A `\syndef` declaration can be used to define operator symbols.
- An "ability" attribute may be specified for declaration boxes, to indicate to Z/EVES that the results of processing the declaration (subsidiary declarations, etc.) should not automatically be used in proofs.
- In some contexts, a predicate may have an optional label, to indicate to Z/EVES that the predicate is a named axiom or rule.
- A binding set ($\lblot \ldots \rblot$) expression has been added.
- Generic actuals may be specified for operator symbols.
- Renaming in schema references has been generalized to allow an expression to be substituted for a variable, rather than just name substitution.
- An `IF` predicate, analogous to the `IF` expression, has been added.

- A `\Local` or `\Global` indication may be added to variable names in expressions to distinguish between references to local and global names.
- The selection operator "`.`" has been extended to allow selection from a tuple; a number may appear on the right as well as a var-name.
- Support for "quoted" identifiers has been added, to allow Z/EVES to generate names for subsidiary declarations without ugly name mapping. Identifiers may also contain escaped dollar signs as well as underscores.
- The LaTeX `\read` and `\include` commands are supported, and cause the specified file to be read in batch mode (Section 3.1).