The *f*UZZ Manual

Mike Spivey

Second Edition

The Spivey Partnership

10 Warneford Road
Oxford OX4 1LU
England

Contents

1 Introduction

*f*UZZ is a collection of tools that help you to format and print Z specifications and check them for compliance with the Z scope and type rules. One part of the package is a style option for the LaTeX type-setting system that defines extra LaTeX commands for laying out Z specifications, and a font that contains Z's special symbols. The other part is a program for analysing and checking specifications that are written using these commands.

Chapter 2 of this document is a quick introduction to *f*UZZ and its features, based on a small example specification. Chapter 3 describes in more detail how to enter your specification as an input file for LaTeX using the `fuzz` style option. Chapters 4 to 6 describe the *f*UZZ type checker and the messages and reports it can produce. Chapter 7 is a syntax summary for the Z language as recognized by the type checker.

In writing this document, I have assumed that you have some basic knowledge about LaTeX and Z. For information about LaTeX and the underlying TeX program, you should consult the manuals

Leslie Lamport, LaTeX: *A Document Preparation System*, Addison Wesley, 1985.

Donald E. Knuth, *The TeXbook*, Addison Wesley, 1984.

For information about Z, and a description of the scope and type rules used by the *f*UZZ type checker, you should consult

J. M. Spivey, *The Z Notation: A Reference Manual*, Second edition, Prentice-Hall International, 1992,

which is referred to in this document by the abbreviation ZRM.

This manual describes versions of the *f*uzz package numbered 2.0 or higher; the language supported is that described in the second edition of the ZRM. The `fuzz` style option is compatible with version 2.99 of TeX and version 2.09 of LaTeX and SliTeX.

Release 2 of *f*uzz differs from release 1 in supporting the additional language constructs added in the second edition of the ZRM, including

- renaming of schema components.
- a **let** construct for local definitions in expressions and predicates.
- conditional **if then else** expressions.
- piping of operation schemas ($\gg$).

In addition, there is an extended mathematical tool-kit and font of special symbols, and it is now possible to use SliTeX to make slides of Z specifications. Finally, there are improvements in the type inference algorithm that the *f*uzz type-checker uses to calculate the types it displays in error messages and reports.

2 First Steps

The *f*uzz distribution includes a small example specification in the file `example.tex`. Here it is, with the lines numbered for easy reference:

```
 1 \documentstyle[fuzz]{article}
 2 \begin{document}
 3 \noindent Let $PERSON$ be the set of all people:
 4 \begin{zed}
 5         [PERSON].
 6 \end{zed}
 7 A ‘club’ has a set of members and a president, who is
 8 one of the members:
 9 \begin{schema}{Club}
10         members: \power PERSON \\
11         president: PERSON
12 \where
13         president \subseteq members
14 \end{schema}
15 To enroll somebody in the club, we just add them
16 to the set of members:
17 \begin{schema}{Enrol}
18         \Delta Club \\
19         new?: PERSON
20 \where
21         members’ = members \cup new? \\
22         president’ = president
23 \end{schema}
24 The president doesn’t change when a new member
25 is enrolled.
26 \end{document}
```

This file looks like any other LaTeX manuscript, except that it includes the style option `fuzz` (line 1), and it contains three embedded pieces of Z text, written using the style option's environments and macros.

The first piece of Z text (lines 4–6) is the declaration of a basic type *PERSON*. The second and third pieces (lines 9–14 and 17–23) are two schemas: the state space of a very simple membership register for a club, and an operation that enrolls a new member.

You might like to make a copy of the file and format it with LaTeX by issuing the command:

```
latex example
```

Now send the results to a printer. The output should look something like this:

Let *PERSON* be the set of all people:

$$[\mathit{PERSON}].$$

A 'club' has a set of members and a president, who is one of the members:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Club}} \\ \hline
\mathit{members} : \mathbb{P}\,\mathit{PERSON} \\
\mathit{president} : \mathit{PERSON} \\ \hline
\mathit{president} \subseteq \mathit{members} \\ \hline
\end{array}$$

To enroll somebody in the club, we just add them to the set of members:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Enrol}} \\ \hline
\Delta \mathit{Club} \\
\mathit{new}? : \mathit{PERSON} \\ \hline
\mathit{members}' = \mathit{members} \cup \mathit{new}? \\
\mathit{president}' = \mathit{president} \\ \hline
\end{array}$$

The president doesn't change when a new member is enrolled.

As you can see, the `zed` environment produces an ordinary mathematical display, and the schema environment produces a schema box.

The specification contains a couple of mistakes that can be found using the *f*uzz type checker. So let's run the type checker on the specification and see what error messages it gives. You can run the type checker like this:

```
fuzz example.tex
```

The input to the type checker is exactly the same file that you used as input to LaTeX. But instead of printing the document, *f*uzz scans it, extracts the pieces of text that make up the formal specification, and checks that they conform to the rules of the Z language. Here are the error messages it produces:

```
"example.tex", line 13: Type mismatch in left argument
        of infix relation
> Predicate: president \subseteq members
> Arg type:  PERSON
> Expected:  P ?

"example.tex", line 21: Right argument of operator \cup
        has wrong type
> Expression: members \cup new?
> Arg type:   PERSON
> Expected:   P PERSON
```

The first error is in the predicate

$$\mathit{president} \subseteq \mathit{members}.$$

It is supposed to say that the president of the club must be a member, but the subset relation $\subseteq$ has been used instead of the membership sign $\in$. This is wrong because *president* is not a set but a simple element of *PERSON*. The type checker says that the left argument of $\subseteq$ must be a set (with type $\mathbb{P}\,?$ for some type ?), but it has found an expression of type *PERSON* instead. The correct thing would be to use the membership sign $\in$ instead of $\subseteq$, so use your favourite editor to change line 13 of the file so that it reads

```
president \in members
```

The second error is in the expression

$members \cup new?$.

This is supposed to be the set *members* extended with an additional member *new*?, but it is wrong because *new*? is just a *PERSON*, and the union operator ∪ expects a *set* of elements of *PERSON*. The type checker knows from the standard library that ∪ expects sets as its arguments, and it has determined by looking at the left argument *members* that this use of ∪ is for sets drawn from *PERSON*, but it finds as the right argument a simple element of *PERSON*. You should change line 21 so that it reads

```
members' = members \cup \{new?\} \\
```

In the corrected expression, *new*? is made into a singleton set before taking the union with *members*, like this: $members \cup \{new?\}$. After you have made these changes, the type checker should report no errors when it checks the document.

In addition to checking for errors, the type checker can produce a report listing the names defined in your specification and their types. You can get this list by giving the command

```
fuzz -t example.tex
```

Here is the output:

```
Given PERSON

Schema Club
    members: P PERSON
    president: PERSON
End

Schema \Delta Club
    members: P PERSON
    president: PERSON
    members': P PERSON
    president': PERSON
End
```

```
Schema Enrol
    members: P PERSON
    president: PERSON
    members': P PERSON
    president': PERSON
    new?: PERSON
End
```

This output shows that *PERSON* has been declared as a basic type, and that three schemas *Club*, Δ*Club* and *Enrol* have been defined. The definition of Δ*Club* was inserted by the type checker because Δ*Club* is used in *Enrol* but was given no explicit definition in the specification. For each schema, the signature is shown after all the inclusions have been expanded, so you can see that *Enrol* has five variables in all.

There is another, much longer, example specification included in the distribution under the name `tut.tex`; it is essentially the same as the first chapter of the ZRM. This file contains examples of many of the Z constructs, and you may like to look at it as you read the rest of this manual. The file also shows how the *f*UZZ style option can be used to print small pieces of program text.

3 Printing Specifications

Every LaTeX document begins with a `\documentstyle` command. If the document contains a Z specification, this command should include the style option `fuzz`, like this:

```
\documentstyle[12pt,fuzz]{article}
```

The `fuzz` style option can be used with any of the standard LaTeX styles, and it can appear either before or after the type-size option if one is used. It can be combined with most of the standard style options, but it should not be combined with `fleqn`, because it already makes provision for setting mathematics flush left. The `fuzz` style option can be used with SliTeX to make slides, but only in black and white at present.

The `fuzz` style option provides a number of LaTeX environments for making Z boxes of various kinds, and for other unboxed elements of Z like basic type definitions. It also provides many commands that print the fancy symbols of Z using the special font supplied in the package. Finally, it provides some environments that print things, like equational proofs, that are not properly part of Z but often appear in specification documents.

Section 3.1 describes the environments that are used for printing the various boxes and unboxed elements used in Z specifications; these environments are the ones recognized and analysed by the *f*uzz type checker. Section 3.2 explains how to obtain the symbols that go inside the boxes, and Section 3.3 gives some hints on getting the best-looking layout from LaTeX and the `fuzz` style option. Section 3.4 explains the environments for printing miscellaneous displayed formulas. The chapter closes with a list (Section 3.5) of the parameters you may adjust to customize the layout of your specification.

If you want it, the fancy logo '*f*uzz' is made by the command `\fuzz`, which works like the `\LaTeX` command for printing LaTeX's own logo.

3.1 Making boxes

The Z reference manual lists in Sections 3.2 and 3.9 the various kinds of 'paragraph' that may appear as one of the top-level units of a Z specification. The `fuzz` style option provides a LaTeX environment for each kind of paragraph that comes in a box, and another environment called `zed` for the unboxed paragraphs.

Schema definitions (ZRM Section 3.2.4)
To print a schema, just use the `schema` environment. Here is an example, showing first the input, then the output from LaTeX:

```
\begin{schema}{BirthdayBook}
    known: \power NAME \\
    birthday: NAME \pfun DATE
\where
    known = \dom birthday
\end{schema}
```

BirthdayBook

$known : \mathbb{P}\ NAME$

$birthday : NAME \pfun DATE$

$known = \operatorname{dom} birthday$

The name of the schema appears as an argument to the environment, and (as in all kinds of box) the horizontal dividing line between declarations and predicates is indicated by `\where`. Successive lines in the declaration and predicate parts are separated by the command `\\`. As usual in LaTeX, the actual layout of the input is ignored, and it is only the explicit command `\\` that marks the end of a line in the output. The Z symbols '$\mathbb{P}$', '$\pfun$' and 'dom' have been entered as the commands `\power`, `\pfun` and `\dom`: for a complete list of these commands, see Section 3.2.

Like the `displaymath` environment of LaTeX, the `schema` environment (and the others we shall come to in a moment) can appear in the middle of a paragraph of text, and ordinarily should have no blank lines either before or after it. Blank lines before the environment are ignored, but blank lines afterwards cause the following text to begin a new paragraph.

For a schema without a predicate part, the command `\where` is simply omitted, as in the following example:

```
\begin{schema}{Document}[CHAR]
    left, right: \seq CHAR
\end{schema}
```

$$\begin{array}{|l}\hline \mathit{Document}[\mathit{CHAR}] \\ \mathit{left}, \mathit{right} : \mathrm{seq}\ \mathit{CHAR} \\ \hline \end{array}$$

This example also shows how to print the generic schemas described in Section 3.9.1 of the ZRM. The formal generic parameters are an optional argument of the `schema` environment. For compatibility with previous versions of *fuzz*, you can also say `\begin{schema}{Document[CHAR]}`, making the formal parameters part of the `schema` environment's first argument. The printed effect is the same, and the type checker recognizes both forms.

If a schema or other box contains more than one predicate below the line, it often looks better to add a small vertical space between them. This can be done with the command `\also`:

```
\begin{schema}{AddPhone}
    \Delta PhoneDB \\
    name?: NAME \\
    number?: PHONE
\where
    name? \notin known
\also
    phone' = phone \oplus \{name? \mapsto number?\}
\end{schema}
```

$$\begin{array}{|l}\hline \mathit{AddPhone} \\ \Delta \mathit{PhoneDB} \\ \mathit{name}? : \mathit{NAME} \\ \mathit{number}? : \mathit{PHONE} \\ \hline \mathit{name}? \notin \mathit{known} \\[4pt] \mathit{phone}' = \mathit{phone} \oplus \{\mathit{name}? \mapsto \mathit{number}?\} \\ \hline \end{array}$$

Axiomatic descriptions (ZRM Section 3.2.2)
These are printed with the `axdef` environment[1]. Here is an example:

```
\begin{axdef}
    square: \nat \fun \nat
\where
    \forall n: \nat @ \\
\t1     square(n) = n * n
\end{axdef}
```

$$\begin{array}{|l} \mathit{square} : \mathbb{N} \longrightarrow \mathbb{N} \\ \hline \forall n : \mathbb{N} \bullet \\ \qquad \mathit{square}(n) = n * n \end{array}$$

Incidentally, this example illustrates that predicates and declarations can be split between lines before or after any infix symbol (such as $\bullet$ here). The strange hint `\t1` makes the corresponding line in the output have one helping of indentation. As things get more nested, you can say `\t2`, `\t3`, and so on; these commands are ignored by the type checker, so you are free to use them as you like to improve the look of your specification. The `\t` command usurps the name used by LaTeX for a tie-after accent, as in 'o͡o', so the `fuzz` style option renames the accent as `\tie`.

[1]The environment should really be called `axdesc` and not `axdef`, but that's too hard to pronounce.

Like schemas, axiomatic descriptions may omit the horizontal dividing line and the predicates below it, leaving only the declaration of some variables, as in the following example:

```
\begin{axdef}
    n\_disks: \nat
\end{axdef}
```

$$\mathit{n\_disks} : \mathbb{N}$$

Generic definitions (ZRM Section 3.9.2)
These are printed with the `gendef` environment:

```
\begin{gendef}[X,Y]
    fst: X \cross Y \fun X
\where
    \forall x: X; y: Y @ \\
\t1     fst(x,y) = x
\end{gendef}
```

$$[X, Y]$$

$$\mathit{fst} : X \times Y \longrightarrow X$$

$$\forall x : X; y : Y \bullet \\ \quad \mathit{fst}(x, y) = x$$

In this environment, the formal generic parameters are an optional argument. Omitting this argument results in a box with a solid double bar at the top, which can be used for uniquely defined constants that have no parameters.

Unboxed paragraphs
Some Z paragraphs do not appear in boxes, and for these the `zed` environment is used. They may be basic type definitions (ZRM section 3.2.1), predicates written as global constraints (Section 3.2.3), 'horizontal' schema definitions (Section 3.2.4), abbreviation definitions (Sections 3.2.5 and 3.9.2), or free type definitions (Section 3.10). Here are a few examples:

```
\begin{zed}
    [NAME, DATE]
\also
    REPORT ::= ok | unknown \ldata NAME \rdata
\also
    \exists n: NAME @ \\
\t1     birthday(n) \in December.
\end{zed}
```

$$[\mathit{NAME}, \mathit{DATE}]$$

$$\mathit{REPORT} ::= \mathit{ok} \mid \mathit{unknown}\langle\!\langle \mathit{NAME} \rangle\!\rangle$$

$$\exists n : \mathit{NAME} \bullet \\ \quad \mathit{birthday}(n) \in \mathit{December}.$$

As the example illustrates, a full stop or comma is allowed just before the closing `\end` command of any of the Z environments, if that suits your taste (or is forced on you by a publisher's house rules). This punctuation is ignored by the type checker.

For large free type definitions, the `syntax` environment provides a useful alternative to the `zed` environment, as the following example suggests:

```
\begin{syntax}
    OP  & ::= & plus | minus | times | divide
\also
    EXP & ::= & const \ldata \nat \rdata \\
        &  |  & binop \ldata OP \cross EXP
                                     \cross EXP \rdata
\end{syntax}
```

$$\mathit{OP} ::= \mathit{plus} \mid \mathit{minus} \mid \mathit{times} \mid \mathit{divide}$$

$$\begin{aligned} \mathit{EXP} &::= \mathit{const}\langle\!\langle \mathbb{N} \rangle\!\rangle \\ &\mid \mathit{binop}\langle\!\langle \mathit{OP} \times \mathit{EXP} \times \mathit{EXP} \rangle\!\rangle \end{aligned}$$

Just as in the `eqnarray` environment of LaTeX, the fields are separated by `&` characters; these are ignored by the type checker.

3.2 Inside the boxes

The first thing to notice about the text inside the boxes is that multi-character identifiers look better than they do with ordinary LaTeX: instead of $effective\ specifications$, you get *effective specifications*. The letters are not spread apart, and ligatures like *ff* and *fi* are used. This improvement is achieved by an adjustment to the way TeX treats letters in mathematical formulas, and no special commands are needed in the input file.

Embedded underscore characters in identifiers can be printed with the `\_` command, so that `not\_known` prints as *not_known*. The same command can be used for the dummy arguments that surround infix operators when they appear on their own; for example, `\_ + \_` prints as $\_ + \_$. Occasionally, the `\_` command makes an odd-looking space in an identifier, as in $DISK\_POOL$, where the large 'overhang' of the italic K has pushed the underline too far away. This is an unfortunate consequence of TeX's rules for setting mathematical formulas, but it can be avoided by typing `"DISK\_POOL"` instead (using double quote marks). In Z text, this produces a better-looking *DISK_POOL*; the quote marks are completely ignored by the *f*UZZ type checker.

So much for the identifiers in Z specifications; what about the mathematical symbols? As in ordinary LaTeX, they are typed using LaTeX commands with (hopefully) mnemonic names. Many Z symbols have the same name that they are given by LaTeX, but some are given a shorter name that is more suggestive of their use in Z. The `fuzz` style option also adjusts the way that many of the symbols interact with TeX's rules for spacing in formulas, so that they will look better in Z specifications.

A few symbols have two names, reflecting two different uses for the symbol in Z:

- The symbol $⨾$ is called `\semi` when it is used as an operation on schemas, and `\comp` when it is used for composition of relations.
- The symbol $\setminus$ is called `\hide` as the hiding operator of the schema calculus, and `\setminus` as the set difference operator.

- The symbol $\upharpoonright$ is called `\project` as the schema projection operator, and `\filter` as the filtering operator on sequences.

Although the printed appearance of each of these pairs of symbols is the same, the type checker recognizes each member of the pair only in the appropriate context.

The rest of this section contains lists of the commands defined in the `fuzz` style option for printing the mathematical symbols of Z. The lists are organized according to the class of symbol, with all relation signs in one list, all binary operators in another, and so on. This classification of standard symbols is known to the *f*UZZ type checker, which uses the information in parsing pieces of Z text. The symbols are also listed according to subject on the Z reference card, part of the *f*UZZ package, with all the symbols connected with sets listed together, and so on.

Some of the symbols listed here look like words in a special font: for example, `\dom` yields the word 'dom' in roman type, `\IF` yields '**if**' in bold-face type, and `\prefix` yields 'prefix' in sans-serif type. As far as the type checker is concerned, these symbols are completely different from plain identifiers like *dom*, *if*, and *prefix*.

Language elements

Some symbols are basic elements of the Z language. Here are the mnemonics for them:

$\mathbb{P}$	`\power`	μ	`\mu`
$\times$	`\cross`	**let**	`\LET`
$=$	`=`	Δ	`\Delta`
$\in$	`\in`	Ξ	`\Xi`
$\mid$	`|` or `\mid`	$\widehat{=}$	`\defs`
$\bullet$	`@` or `\spot`	**if then else**	
θ	`\theta`		`\IF \THEN \ELSE`
λ	`\lambda`		

The commands `|` and `@` produce the vertical bar and spot that appear in formulas like

$$\forall x : \mathbb{N} \mid x > 0 \bullet x \geq 1.$$

They are typed as single characters, but like the other commands for mathematical symbols, they may only be used in math mode. For upward compatibility, the commands `\mid` and `\spot` are also provided.

The symbol $\Rrightarrow$ that the ZRM uses to write down bindings isn't really part of Z, but it has a mnemonic anyway:

$\Rrightarrow$	`\bind`

Connectives and quantifiers

Here are the commands for operators of propositional and predicate logic and the schema calculus:

$\neg$	`\lnot`	$\exists$	`\exists`
$\land$	`\land`	$\exists_1$	`\exists_1`
$\lor$	`\lor`	$\setminus$	`\hide`
$\Rightarrow$	`\implies`	$\upharpoonright$	`\project`
$\Leftrightarrow$	`\iff`	pre	`\pre`
$\forall$	`\forall`	$\fatsemi$	`\semi`

Fancy brackets

Next come commands for the various sorts of fancy brackets:

$\{ \}$	`\{ \}`	$\langle\!\langle \rangle\!\rangle$	`\ldata \rdata`
$\langle \rangle$	`\langle \rangle`	$(\!\mid \mid\!)$	`\limg \rimg`
$[\![ ]\!]$	`\lbag \rbag`		

In addition to the brackets used in specifications, there are the special brackets used in the ZRM to write down schema types. Like $\Rrightarrow$, they aren't part of Z, but they are sometimes useful in writing *about* a specification:

$\langle\!\mid \mid\!\rangle$	`\lblot \rblot`

Those are all the symbols 'built-in' to the Z language; now for the symbols that are defined as part of the mathematical tool-kit.

Constants and functions

Some symbols name ordinary mathematical constants and functions:

$\varnothing$	`\emptyset`	$\mathbb{N}$	`\nat`
$\bigcup$	`\bigcup`	$\mathbb{Z}$	`\num`
$\bigcap$	`\bigcap`	$\mathbb{N}_1$	`\nat_1`
dom	`\dom`	$\#$	`\#`
ran	`\ran`	$\frown/$	`\dcat`

The control sequence `\empty` was used in place of `\emptyset` in previous versions of the `fuzz` style option, but this caused problems because it clashes with a macro of plain TEX. If an existing document uses `\empty`, you can still print it by adding the definition

```
\renewcommand{\empty}{\emptyset}
```

in the document preamble (or by changing the name with an editor). Both names are recognized by the *f*UZZ type checker.

Infix function symbols

Next, here are the infix function symbols (class In-Fun), each shown with its priority, which is a number from 1 to 6, with 6 giving the tightest binding:

$\mapsto$	`\mapsto`	1	$\cap$	`\cap`	4
$\ldots$	`\upto`	2	$\upharpoonleft$	`\extract`	4
$+$	`+`	3	$\upharpoonright$	`\filter`	4
$-$	`-`	3	$\fatsemi$	`\comp`	4
$\cup$	`\cup`	3	$\circ$	`\circ`	4
$\setminus$	`\setminus`	3	$\otimes$	`\otimes`	4
$\frown$	`\cat`	3	$\oplus$	`\oplus`	5
$\uplus$	`\uplus`	3	$\sharp$	`\bcount`	5
$\uminus$	`\uminus`	3	$\lhd$	`\dres`	6
$*$	`*`	4	$\rhd$	`\rres`	6
div	`\div`	4	$⩤$	`\ndres`	6
mod	`\mod`	4	$⩥$	`\nrres`	6

The priority of operators means nothing to LATEX, but it is used by the *f*UZZ type checker when it analyses expressions.

Postfix function symbols

The standard postfix function symbols (class Post-Fun) produce different kinds of superscripts:

$^{\sim}$	`\inv`	$^{+}$	`\plus`
*	`\star`	^{n}	`^{n}`

For example, `R \star` is printed as R^*, and `R^{n}` is printed as R^n. The type checker regards this second formula as equivalent to *iter n R*, as explained on page 110 of the ZRM; the braces are *not* optional, even when the superscript is a single digit. For upward compatibility, the form `R \bsup n \esup` is also recognized by both the style option and the type checker.

Infix relation symbols

The infix relation symbols have class In-Rel:

$\neq$	`\neq`	prefix	`\prefix`
$\notin$	`\notin`	suffix	`\suffix`
$\subseteq$	`\subseteq`	in	`\inseq`
$\subset$	`\subset`	⋿	`\inbag`
$<$	`<`	$\sqsubseteq$	`\subbageq`
$\leq$	`\leq`	partition	`\partition`
$\geq$	`\geq`	$\underline{R}$	`\inrel{R}`
$>$	`>`		

Besides the fixed infix relation symbols, an ordinary identifier may be used as an infix relation if it is underlined; to get $x \ \underline{R} \ y$, the input is `x \inrel{R} y`.

Prefix relation symbols

There is only one standard prefix relation symbol (class Pre-Rel):

disjoint	`\disjoint`

Infix generic symbols

The infix generic symbols are assigned the class In-Gen. The standard ones are all more-or-less fancy arrows, including the famous 'dead fish'. The arrows used by *f*uzz are a little bigger than the ones that come with LaTeX:

$\leftrightarrow$	`\rel`	⤀	`\psurj`
⇸	`\pfun`	$\twoheadrightarrow$	`\surj`
$\rightarrow$	`\fun`	⤖	`\bij`
⤔	`\pinj`	⇻	`\ffun`
$\rightarrowtail$	`\inj`	⤕	`\finj`

Prefix generic symbols

Prefix generic symbols are assigned class Pre-Gen by the type checker. Here is the standard list:

$\mathbb{P}_1$	`\power_1`	seq	`\seq`
id	`\id`	seq_1	`\seq_1`
$\mathbb{F}$	`\finset`	iseq	`\iseq`
$\mathbb{F}_1$	`\finset_1`	bag	`\bag`

As far as LaTeX is concerned, the infix function and generic symbols are defined to be binary operators, so they are separated from their arguments by a medium space. The infix relation symbols are defined as relation symbols, so that LaTeX inserts thick spaces around them. The prefix generic symbols and most of the ordinary symbols that denote functions are defined as operator symbols, so LaTeX inserts a thin space between the symbol and its argument. The spacing rules of TeX work well enough that most Z specifications can be printed without the need for manual adjustment of spacing, but the next section points out some places where human assistance is needed.

3.3 Fine points

In math mode, which is used for type-setting the contents of Z boxes, LaTeX ignores all space characters in the input file. The spaces which appear between elements of a mathematical formula are determined by LaTeX itself, working from information about the symbols in the formula. Although (as described above) this information has been adjusted in the *fuzz* style option to make Z texts look as balanced as possible, there are one or two situations in which LaTeX needs a little help.

Special care is needed when function application is indicated by juxtaposing two identifiers, as in the expression *rev words*, which should be typed as `rev~words`. Typing just `rev words` results in the output *revwords*, since LaTeX ignores the space separating the two identifiers. The `fuzz` style option extends the definition of `~` so that in a formula it inserts the same amount of space as the LaTeX `\,` command. The type checker completely ignores both the `~` character and the LaTeX spacing commands, except that it issues a warning if it finds that one is missing between two identifiers. It is not necessary to separate symbols like `\dom` and `\ran` from their arguments with a `~`, because LaTeX inserts the right amount of space automatically. For example, the input `\dom f` produces '$\operatorname{dom} f$'.

It is good style also to insert small spaces inside the braces of a set comprehension, as in this example:

```
\{~x: \nat | x \leq 10 @ x * x~\}
```

$\{\, x : \mathbb{N} \mid x \leq 10 \bullet x * x \,\}$

This helps to distinguish it visually from a set display, which should not have the space:

```
\{1, 2, 3\}
```

$\{1, 2, 3\}$

Of course, the space symbol `~` is ignored by the type checker, which distinguishes set displays from comprehensions by looking at their contents, so the use of `~` is purely a matter of aesthetics. It also looks better if you add small spaces inside the square brackets of 'horizontal' schema texts.

LaTeX also needs help when a binary operator appears at the end of a line, as in the following example:

```
\begin{zed}
    directory' = directory \cup {} \\
\t3                 \{new\_name? \mapsto new\_number?\}
\end{zed}
```

$$\begin{aligned} &directory' = directory \cup \\ &\qquad\qquad \{new\_name? \mapsto new\_number?\} \end{aligned}$$

LaTeX will not recognize `\cup` as a binary operator and insert the correct space unless it is surrounded by two operands, so the empty operand `{}` has been inserted: this is ignored by the type checker. This problem affects only binary operators; relation signs do not need to be surrounded by arguments to be recognized by LaTeX.

3.4 Bits and pieces

Specification documents often contain mathematical text which does not form part of the formal specification proper. This section describes some environments for setting various kinds of informal mathematics; they are provided for convenience, and they are all ignored by the type checker. Besides these environments for making displays, run-in mathematics can be set with the usual `math` environment, or with the commands `$ ... $` or `\( ... \)`. All the Z symbols listed in Section 3.2 can be used with these commands.

The simplest displays are produced by the commands `\[ ... \]`. This form acts just like `\begin{zed} ... \end{zed}`, except that the contents are ignored by the type checker; it can be used to state informal theorems about a specification, or to quote a piece of mathematical text for discussion. The `\[ ... \]` commands generalize the standard LaTeX ones, because the displayed material can be several lines. Note, however, that the contents are set in text style rather than display style. Here is an example:

```
\[
    \exists PhoneDB @ \\
\t1     known = \emptyset
\]
```

$$\exists \mathit{PhoneDB} \bullet \\ \quad \mathit{known} = \varnothing$$

Sometimes it is nice to expand a complicated schema expression and display the results in a schema box with no name. Such boxes can be printed using the `schema*` environment, like this:

```
\begin{schema*}
        x, y: \nat
\where
        x > y
\end{schema*}
```

$$\begin{array}{|l} x, y : \mathbb{N} \\ \hline x > y \end{array}$$

Like all these environments, the `schema*` environment is ignored by the type checker, and contributes nothing to the picture of the formal specification it is building.

Another kind of mathematical display is provided by the `argue` environment. This is like the `zed` environment, but the separation between lines is increased a little, and page breaks may occur between lines. The intended use is for arguments like this:

```
\begin{argue}
    S \dres (T \dres R) \\
\t1     = \id S \comp \id T \comp R \\
\t1     = \id (S \cap T) \comp R & law about $\id$ \\
\t1     = (S \cap T) \dres R.
\end{argue}
```

$$\begin{array}{ll} S \lhd (T \lhd R) & \\ \quad = \mathrm{id}\, S \mathbin{⨾} \mathrm{id}\, T \mathbin{⨾} R & \\ \quad = \mathrm{id}(S \cap T) \mathbin{⨾} R & \text{[law about id]} \\ \quad = (S \cap T) \lhd R. & \end{array}$$

When the left-hand side is long, I find this style better than the LaTeX `eqnarray` style, which wastes a lot of space. Each line can have an optional second field, delimited by an `&` character: it can be used for a hint why the expression is considered equal to the previous line.

Finally, there is the `infrule` environment, used for inference rules:

```
\begin{infrule}
    \Gamma \vdash P
\derive[x \notin freevars(\Gamma)]
    \Gamma \vdash \forall x @ P
\end{infrule}
```

$$\frac{\Gamma \vdash P}{\Gamma \vdash \forall x \bullet P} \quad [\, x \notin \mathit{freevars}(\Gamma) \,]$$

The horizontal line is generated by `\derive`; the optional argument is a side-condition of the rule.

3.5 Style parameters

A few style parameters affect the way Z text is set out; they can be changed at any time if your taste doesn't match mine.

`\zedindent` The indentation for mathematical text. By default, this is the same as `\leftmargini`, the indentation used for list environments.

`\zedleftsep` The space between the vertical line on the left of schemas, etc., and the maths inside. The default is 1 em.

`\zedtab` The unit of indentation used by `\t`. The default is 2 em.

`\zedbar` The length of the horizontal bar in the middle of a schema. The default is 6 em.

`\zedskip` The vertical space inserted by `\also`. By default, this is the same as that inserted by `\medskip`.

`\zedsize` A size-changing command used for Z text. It may be redefined by a command such as `\renewcommand{\zedsize}{\small}`. The default is for Z text to be the same size as surrounding prose.

4 Checking Specifications

The *f*UZZ type checker can be run with the command

`fuzz` *[*`-aqstv`*] [*`-p` *prelude] [file ...]*

It reads the input files, which are Z specifications written as described in Chapter 3, extracts the pieces of formal text, and produces messages on the standard error stream describing any errors it finds. If no input files are listed on the command line, *f*UZZ reads the standard input stream instead. Various optional flags can be given to make *f*UZZ output additional information or to modify the processing it performs.

Before reading the input files, the type checker reads a prelude file that contains, amongst other things, the definitions from the mathematical tool-kit in Chapter 4 of the ZRM. This prelude is a text file, and apart from a few special commands it has the same form as any other input file.

Next, *f*UZZ reads the input files in sequence, keeping the definitions from each file as it processes the next one, so that specifications in several files can be analysed, with each file using notation introduced by the ones before it. The LaTeX commands `\input{...}` and `\include{...}` are completely ignored by *f*UZZ, but much the same effect can be obtained by listing the files on the command line.

The following flags can be given on the command line to modify the processing *f*UZZ performs:

`-a` Disable the system of type abbreviations that is normally used for printing types in reports and error messages. Further details of the system of type abbreviations are in Section 5.3.

`-p` *file* Use *file* in place of the standard prelude. This flag is useful if there are local extensions to the mathematical library; you can append them to a copy of the usual prelude file, and use `-p` to have *f*UZZ read the extended prelude in place of the usual one.

`-q` Allow implicit quantifiers. Any undeclared identifier that appears in the predicate part of a schema, axiomatic description, or generic definition is treated as if it were declared by a universal quantifier surrounding the predicate, with a type inferred from the context.

`-d` Dependency analysis. The paragraphs of Z text are topologically sorted before checking, so that (with a few restrictions), the paragraphs can appear in the document in an order that best suits exposition.

`-s` Syntax check only. This flag overrides the `-t` flag, and can be used with the `-v` flag to extract the paragraphs of specification from the input files without checking them for type errors.

`-t` Display on the standard output the types of all names defined globally in the specification. See Section 4.2 for more details.

`-v` Display each paragraph of Z text on the standard output as it is extracted from the input files. See Section 4.2 for more details.

`-l` Echo each paragraph of Z text in a Lisp-like syntax.

If no `-p` flag is present, the type checker looks for the prelude file in an implementation-dependent way. In UNIX versions of *f*UZZ, if there is an environment variable called `FUZZLIB`, its value is used as the location of the prelude file; otherwise, the type checker looks in a place determined when *f*UZZ is installed. In the IBM PC version, the type checker looks in all the directories on the search path for a file named `fuzzlib`, and uses that as the prelude file.

4.1 Definition before use

*f*UZZ checks the input specification for conformance with the language rules of Z described in the ZRM. These rules require that each identifier

or schema name is declared or defined before it is first used. *f*UZZ relaxes this condition if the `-d` flag is specified on the command line, by reading the whole specification, then re-ordering the paragraphs of Z material so that they appear in an order that is acceptable according to the rules. It is still necessary that the specification can be re-ordered in this way, so it is not permitted (for example) to have two paragraphs, each of which refers to definitions made in the other.

Another kind of exception to the principle of definition before use is made for the symbols Δ and Ξ. These may appear only as the first character of a schema name such as $\Delta \mathit{State}$ (typed as `\Delta State`). If no schema with this name has been defined before the first use of the name, the standard definition is inserted by the type checker: see, for example, the small specification in Chapter 2. This convention allows both the style where $\Delta \mathit{State}$ is an abbreviation for $\mathit{State} \land \mathit{State}'$, and the style where it may be defined explicitly to be something else. Of course, if the definition is left implicit, then $\mathit{State} : \mathit{Exp}$ must be defined as a schema *before* the first use of $\Delta \mathit{State}$.

Normally, it is an error for a Z specification to use any identifier that has not been declared; but some specifications are made much shorter by the convention that undeclared variables are universally quantified in the largest surrounding predicate. This allows definitions like this:

$$\begin{array}{|l} \mathit{double} : \mathbb{N} \longrightarrow \mathbb{N} \\ \hline \mathit{double}(x) = 2 * x \end{array}$$

which would otherwise need a universal quantifier for x. The `-q` flag is provided so that specifications using this informal convention can still be checked for other errors.

4.2 Reports

In addition to checking a specification for errors, the type checker can produce two kinds of reports about the formal text it finds. One report is activated by the `-v` flag, and contains an ASCII representation of each Z paragraph; the other is activated by `-t`, and lists the type of every name that is globally defined by the specification. For example, the schema

$$\begin{array}{|l}\hline \multicolumn{1}{l}{\mathit{PhoneDB}} \\ \mathit{known} : \mathbb{P}\,\mathit{NAME} \\ \mathit{phone} : \mathit{NAME} \pfun \mathit{PHONE} \\ \hline \mathit{known} = \operatorname{dom} \mathit{phone} \\ \hline \end{array}$$

is represented like this in the `-v` report:

```
schema PhoneDB
    known: P NAME
    phone: NAME -+> PHONE
where
    known = dom phone
end
```

This output attempts to show the internal form of the schema in a readable way. The schema box is represented by the keywords `schema`, `where` and `end`, and symbols like $\pfun$ are built up from ASCII characters. The `-t` flag generates a report like this from the *PhoneDB* schema:

```
Schema PhoneDB
    known: P NAME
    phone: NAME -+> PHONE
End
```

*f*uzz discards the predicate part of the schema after checking it for errors, because it contains no information that is needed to check the rest of the specification; so the predicate part does not appear in the `-t` report either. On the other hand, all inclusions of one schema in another are expanded before the report is produced, so you can see exactly what components each schema has. Like the types displayed in error messages, the types shown in `-t` reports use the system of type abbreviations described in Section 5.3 to make them shorter and more readable. If you want to see the 'real' types, you can use the `-a` flag in combination with `-t`. In the example, the type of *phone* would then be

shown as

$$\mathbb{P}(\mathit{NAME} \times \mathit{PHONE})$$

instead of

$$\mathit{NAME} \pfun \mathit{PHONE}.$$

An axiomatic description like this:

$$\begin{array}{|l} u, v : \mathbb{N} \\ \hline 2 * u + v < 10 \end{array}$$

gives a report like this with the `-v` flag:

```
axdef
    u, v: NN
where
    (2 * u) + v < 10
end
```

One thing that makes the `-v` reports particularly useful is that extra parentheses are used to clarify the binding power of operators, as in the expression `(2 * u) - v` here. We all know that multiplication binds more tightly than addition, but if *f*uzz reports an error in your specification that you just can't find, sometimes the `-v` report will reveal that an expression is begin parsed in an unexpected way.

The `-t` flag shows separately each global definition which *f*uzz stores as it checks the specification: for our example, the output is

```
Var u: NN

Var v: NN
```

This shows that two global variables, u and v, have been introduced, and both have type $\mathbb{N}$. Again, the predicate relating u and v has been checked for errors and discarded.

The types of generic constants are shown under `-t` with numeric markers in place of the formal generic parameters. For example, the definition

$$
\begin{array}{|l}
\hline [X, Y] \\
\mathit{fst} : X \times Y \longrightarrow X \\
\mathit{snd} : X \times Y \longrightarrow Y \\
\hline
\forall x : X; y : Y \bullet \\
\qquad \mathit{fst}(x, y) = x \land \\
\qquad \mathit{snd}(x, y) = y \\
\hline
\end{array}
$$

produces this output under `-t`:

```
Genconst fst[2] : @1 x @2 -+> @1

Genconst snd[2] : @1 x @2 -+> @2
```

Here, the `[2]` means that *fst* and *snd* each have two generic parameters, and the markers `@1` and `@2` represent the first and the second parameters respectively. This form of output is also used for generic constants defined with ==.

5 Advanced Features

For simple specifications, the features described in Chapter 4 are enough. But more complex specifications may need additional features of the type checker: they may add to Z's standard list of infix operators, they may introduce new mathematical concepts that can be reflected in the way *f*UZZ computes types for expressions, or they may be presented in an informal way that requires slight adjustments to the strict principle of definition before use. All these features of the type checker are activated by *directives*, special comments that are processed by the type checker, but are ignored by LaTeX when it formats the specification for printing.

5.1 Directives

All the directives that activate special features of *f*UZZ start with the characters `%%`, so LaTeX sees them as comments, and they have no effect on the printed version of the specification. Several of them introduce infix function or relation symbols, and so on; others introduce new type abbreviations or mark text to be ignored in type-checking. A special directive allows text to be included for type-checking but ignored by LaTeX. Each directive must appear at the beginning of a line in the input file, and it continues to the end of that line. Any other TeX comment – beginning with only one `%` or not appearing at the start of a line – is ignored by the type checker.

Here is a list of all the directives, with a brief explanation of their meanings. In this list, *symbols* stands for a sequence of one or more identifiers or special symbols, separated by spaces.

`%%inop` *symbols n*
: The symbols are introduced as infix function symbols with priority n, a digit in the range 1 up to 6.

`%%postop` *symbols*
: The symbols are introduced as postfix function symbols.

`%%inrel` *symbols*
: The symbols are introduced as infix relation symbols.

`%%prerel` *symbols*
: The symbols are introduced as prefix relation symbols.

`%%ingen` *symbols*
: The symbols are introduced as infix generic symbols.

`%%pregen` *symbols*
: The symbols are introduced as prefix generic symbols.

`%%type` *symbols*
: Each of the symbols must have a previous global definition as a variable or generic constant with a set type, or as a schema; they are made into type abbreviations. The definitions of the symbols must be monotonic, as described in Section 5.3.

`%%tame` *symbols*
: Each of the symbols must have a previous global definition as a generic function; they are marked as tame functions in the sense explained in Section 5.3.

`%%unchecked`
: The immediately following Z environment is completely ignored by the type checker.

`%%` *text*
: The string *text* is processed by the type checker, but ignored in printing the specification.

The directives which introduce infix function symbols, etc., override any previous directive for the same symbol. There are a few other directives not listed here that are used in the standard prelude to set up the type checker's symbol tables. They may not be used in ordinary specification documents.

5.2 User-defined operators

Specifications that use Z's tool-kit of mathematical symbols are made easier to read by the fact that many of these symbols are defined as infix operators of one kind or another. The ZRM lists the standard operator symbols on page 46, but does not prescribe any way of adding to this list. With *f*UZZ, the standard set of operators may be extended using the directives listed in Section 5.1.

There are three stages in adding a new operator symbol to a specification. The first is to write a definition that tells LaTeX how to print the symbol, choosing a command name to represent the symbol in the input file. The second stage is to announce to *f*UZZ (using one of the directives) that the symbol is to be used as an infix operator of a specified kind; this announcement is often best complemented by an explanation for the human reader of the relative binding power of the operator. The third stage in adding a new infix symbol is to write its mathematical definition in Z.

To understand this procedure, it may help to think of three potential 'audiences' for the specification document: LaTeX, the *f*UZZ type checker, and the human reader. The specification is written as an input file for LaTeX using a certain command to represent the symbol. To format the specification, the LaTeX program must know what printed symbol to associate with the command. This association is set up in the first stage of introducing the symbol, by defining the LaTeX command with `\newcommand`.

The type checker does not need to know how the symbol is to appear on the page, and it ignores the LaTeX definition that contains this information. But it does need to know whether it is an infix function symbol, a prefix generic symbol, and so on. This information is given by a directive in the second stage; this directive is seen by LaTeX as a comment, so it does not affect the printed appearance of the specification.

Human readers want to be told how to read formulas that use the symbol, information that is best conveyed by a small example given in stage 2. They also want to see its definition, given in stage 3. The fact that the definition for LaTeX in stage 1 and the directive for *fuzz* in stage 2 are invisible in the printed text allows the specification author to explain the significance of the new symbol appropriately.

Here is an example which shows how to introduce a new infix function symbol. Let's define $\diamond$ to be an operator which takes two sequences and concatenates the reverse of the first with the second. Luckily, there is already a LaTeX command `\diamond` for $\diamond$, so we don't need to bother with the first stage. For the second stage, we announce in a directive that `\diamond` is an infix function symbol, choosing its priority to be 3:

```
%%inop \diamond 3
```

It is important to announce `\diamond` as an infix symbol *before* giving its definition, or the type checker will not be able to parse the definition correctly. The new symbol should be introduced for the human reader with a suitable English sentence. In this case, one might say "The symbol $\diamond$ will be used as an infix function symbol; it has the same binding power as $\frown$." Personally, I like to avoid using numeric priorities to explain the binding power of infix symbols: if you exploit relative binding powers, it's a good idea to give an example, such as "The expression $a \upharpoonright S \diamond b$ should be read as $(a \upharpoonright S) \diamond b$."

Now for the third stage: we define the operator with a generic definition like this:

```
\begin{gendef}[X]
    \_ \diamond \_: \seq X \cross \seq X \fun \seq X
\where
    \forall a, b: \seq X @ \\
\t1     a \diamond b = (rev~a) \cat b
\end{gendef}
```

$$\begin{array}{|l}
[X] \\
\hline
\_ \diamond \_ : \mathrm{seq}\, X \times \mathrm{seq}\, X \longrightarrow \mathrm{seq}\, X \\
\hline
\forall a, b : \mathrm{seq}\, X \bullet \\
\qquad a \diamond b = (rev\ a) \frown b \\
\hline
\end{array}$$

The 'quoted' name $\_ \diamond \_$ is used in the declaration, because $\diamond$ is now an infix operator. After defining $\diamond$, we can begin to use it in expressions, like this:

```
\begin{zed}
    \langle 1, 2, 3 \rangle
    \diamond \langle 4, 5, 6 \rangle
        = \langle 3, 2, 1, 4, 5, 6 \rangle.
\end{zed}
```

$$\langle 1, 2, 3 \rangle \diamond \langle 4, 5, 6 \rangle = \langle 3, 2, 1, 4, 5, 6 \rangle.$$

As another example, let's define subseq as the relation which holds between two sequences when the all the elements of the first one also appear in the second one in the same order. The first stage is to define a LaTeX command `\subseq` which produces the relation symbol subseq in the right style of type. This is achieved by

```
\newcommand{\subseq}{\mathrel{\sf subseq}}
```

The TeX primitive `\mathrel` is not documented in the LaTeX manual, but it causes its argument to be considered as a relation symbol in mathematical formulas, so that the right amount of space will be inserted around it. There is also `\mathbin` for binary operators, and others for brackets, prefix operators like 'dom' and so on: see the TeXbook, page 155. The font-changing command `\sf` causes the word 'subseq' to appear in sans-serif type, like the standard infix relation symbols in the ZRM.

The second stage is to announce to the type checker that `\subseq` is to be used as an infix relation symbol:

```
%%inrel \subseq
```

Both LaTeX and the type checker now know all they need about our new symbol, and we can proceed to the third stage, giving its mathematical definition:

```
\begin{gendef}[X]
    \_ \subseq \_: \seq X \rel \seq X
\where
    \forall a, b: \seq X @ \\
\t1     a \subseq b \iff (\exists S:
            \power \nat_1 @ a = S \extract b)
\end{gendef}
```

$$\begin{array}{|l}
=[X]= \\
\_\ \mathsf{subseq}\ \_ : \mathrm{seq}\,X \leftrightarrow \mathrm{seq}\,X \\
\hline
\forall\, a, b : \mathrm{seq}\,X \bullet \\
\qquad a\ \mathsf{subseq}\ b \Leftrightarrow (\exists\, S : \mathbb{P}\,\mathbb{N}_1 \bullet a = S \upharpoonright b) \\
\hline
\end{array}$$

The symbol ‘subseq’ can now be used in formulas:

```
\begin{zed}
    \langle 2, 4 \rangle
        \subseq \langle 1, 2, 3, 4 \rangle.
\end{zed}
```

$$\langle 2, 4\rangle\ \mathsf{subseq}\ \langle 1, 2, 3, 4\rangle.$$

The technique of defining a new LaTeX command for an infix operator can also be used with schemas. This is useful because some styles of specification encourage schema names like $\Phi State$ that do not conform to *f*UZZ’s syntax for schema names (see page 59). The trick is to define a new LaTeX command like this:

```
\newcommand{\PhiState}{\Phi State}
```

then define the schema like this:

```
\begin{schema}{\PhiState} ...
```

After these definitions, the command `\PhiState` prints as $\Phi State$, and is recognized as a schema name by the type checker.

5.3 Type abbreviations

The *f*UZZ type checker separates its work into two parts: computing the types of all the expressions which appear in a specification, and matching types with each other where they must agree. *f*UZZ uses the type system described in the ZRM for comparing types, but it uses a richer type system when computing the types of expressions. This makes for better error messages, because types in the richer system are shorter and easier to understand. Because the type system used for matching types is exactly that described in the ZRM, none of the power or expressiveness of Z is lost.

The ZRM’s type system has only three type constructors: power set $\mathbb{P}$, Cartesian product $\times$, and schema type $\langle\!|\ldots|\!\rangle$. In this system, even simple expressions can have rather complicated types: for example, $\mathrm{dom}[X, Y]$ has the type

$$\mathbb{P}(\mathbb{P}(X \times Y) \times \mathbb{P}\,X).$$

This is rather difficult to read, and the fact that dom is a function has been lost. For the same expression, *f*UZZ computes the type

$$(X \leftrightarrow Y) ⇸ \mathbb{P}\,X,$$

which is shorter, and closer to the ‘type’ given to dom in its declaration. This has been achieved by using the *type abbreviations* ‘$\leftrightarrow$’ and ‘⇸’.

We can recover the real type of $\mathrm{dom}[X, Y]$ by expanding the abbreviations, replacing both $A \leftrightarrow B$ and $A ⇸ B$ by $\mathbb{P}(A \times B)$. The value of $A \leftrightarrow B$ is actually equal to this set, but $A ⇸ B$ is a proper subset of $\mathbb{P}(A \times B)$, so the type we obtain by expansion is larger as a set than the unexpanded type. This means that information has been lost in the expansion, but it is safe to assume that any object in the original, abbreviated type is also in the larger, expanded type.

The algorithm used by *f*UZZ to check that types agree across equality and membership signs and in function applications is equivalent to expanding all abbreviations and checking for an exact match, so you are still free to exploit the way notions like function and sequence are defined in Z: where convenient, you can use the fact that functions are sets of ordered pairs, and sequences are functions defined on a segment of the integers.

Here is a list of the type abbreviations defined in the standard prelude:

$$\begin{array}{ll}
\mathbb{N} & \subseteq \mathbb{Z} \\
X \leftrightarrow Y & = \mathbb{P}(X \times Y) \\
X \mathrel{+\!\!\!\rightarrow} Y & \subseteq \mathbb{P}(X \times Y) \\
X \mathrel{+\!\!\!\!\!\!+\!\!\!\rightarrow} Y & \subseteq \mathbb{P}(X \times Y) \\
\mathbb{F}\,X & \subseteq \mathbb{P}\,X \\
\mathrm{seq}\,X & \subseteq \mathbb{P}(\mathbb{N} \times X) \\
\mathrm{bag}\,X & \subseteq \mathbb{P}(X \times \mathbb{N})
\end{array}$$

A notable absence from this list is the total function arrow $\longrightarrow$. It is omitted because it is not *monotonic* like the others: if $S \subseteq T$, then it is not generally true that $(S \longrightarrow V) \subseteq (T \longrightarrow V)$. For example, the set $even \longrightarrow \mathbb{Z}$ contains the functions from numbers to numbers that are defined exactly when the argument is in the set *even*; one of its members is the function *half* that takes an even argument and halves it. This set is not a subset of $\mathbb{Z} \longrightarrow \mathbb{Z}$, which contains only functions that are defined on the whole of $\mathbb{Z}$; in fact, the two sets of functions are disjoint.

It is important that abbreviations are monotonic, because otherwise the method *f*UZZ uses to calculate types may conclude that an expression has a certain type, when in fact the value of the expression is not a member of the type. The property of being a total function is simply not one that can be expressed in the type system.

The way that the type checker uses the two-level type system is illustrated by the predicate

$$phone' = phone \oplus \{name? \mapsto number?\}.$$

The type checker calculates the types in this equation as shown in the following listing, where each line shows a sub-expression and its type:

$$\begin{array}{ll}
phone & : NAME \mathrel{+\!\!\!\rightarrow} PHONE \\
name? & : NAME \\
number? & : PHONE \\
name? \mapsto number? & : NAME \times PHONE \\
\{name? \mapsto number?\} & : \mathbb{P}(NAME \times PHONE) \\
\_ \oplus \_ & : (X \leftrightarrow Y) \times (X \leftrightarrow Y) \mathrel{+\!\!\!\rightarrow} (X \leftrightarrow Y)
\end{array}$$

So far, so good. Now the type checker finds that the operator $\oplus$, which expects two relations as its arguments, is being applied to arguments that are a partial function and a set of ordered pairs. This is allowed, for although they are superficially different, all three types

$$\begin{array}{l}
NAME \leftrightarrow PHONE \\
NAME \mathrel{+\!\!\!\rightarrow} PHONE \\
\mathbb{P}(NAME \times PHONE)
\end{array}$$

can be changed into the third one by expanding abbreviations. So the type checker ascribes to the right-hand side the type $NAME \leftrightarrow PHONE$, the result type of $\oplus$ with $NAME$ substituted for the generic parameter X and $PHONE$ substituted for Y. The left-hand side of the equation has type $NAME \mathrel{+\!\!\!\rightarrow} PHONE$. Again, this is superficially different from the type ascribed to the right-hand side, but they have a common expansion, so the equation is accepted.

New type abbreviations can be introduced using the `%%type` directive (see Section 5.1). Any symbol defined globally in the specification as having a set type can be marked as a type abbreviation, but it is up to the user to check that the requirement is satisfied that it be monotonic, because there is no way to check this mechanically. If a type abbreviation is not monotonic, no spurious error messages will be produced, but the types displayed in error messages may be misleading.

Another concept related to type abbreviations is that of a *tame* function. An example is the concatenation operator $\frown$ on sequences. The enriched type system gives this operator the type

$$\mathrm{seq}\,X \times \mathrm{seq}\,X \mathrel{+\!\!\!\rightarrow} \mathrm{seq}\,X,$$

and allows the type checker to deduce unaided that applying this operator to two sequences s and t of the same type $\mathrm{seq}\,X$ yields another *sequence* $s \frown t$, and not simply an element of the expanded type $\mathbb{P}(\mathbb{Z} \times X)$. But there are other relevant properties of $\frown$ that cannot be deduced from its enriched type. One such property is that concatenating two elements of $\mathrm{seq}\,\mathbb{N}$ gives another element of $\mathrm{seq}\,\mathbb{N}$, and not simply an element of the type seq $\mathbb{Z}$. This is a consequence of the fact that $\frown$ is tame, meaning that for any set S, if s and t are in seq S, so is $s \frown t$. Another application of the tameness of $\frown$ is the observation that concatenating two sequences of sequences gives another sequence of *sequences*, and not just a sequence of relations.

Almost all the generic functions defined in the standard library are tame, but one that is not tame is the reflexive–transitive closure operator $\_^*$. Even though *succ* is in $\mathbb{N} \leftrightarrow \mathbb{N}$ according to its ZRM definition, $succ^*$ is not in $\mathbb{N} \leftrightarrow \mathbb{N}$ but is in $\mathbb{Z} \leftrightarrow \mathbb{Z}$.

In order to allow the type checker to deduce reasonable types for expressions involving tame functions, while treating non-tame functions correctly, there is a directive `%%tame` for announcing that a generic function is tame for the types with which it was declared (for details, see Section 5.1). Like the `%%type` directive, it is used extensively in the standard prelude, but is less commonly used in ordinary specification documents. In checking a function application, the type checker treats it as a special case if the function being applied is known to be tame. If the function being applied is not a generic function symbol with implicit parameters that has been announced as tame, the type checker uses a more conservative method of calculating the type.

5.4 Invisible and unchecked paragraphs

The `%%unchecked` directive causes the next use of a Z environment – one of `zed`, `axdef`, `schema` or `gendef` – to be ignored by the type checker. It is useful for schemas which contain informal nonsense, as in the following example:

```
%%unchecked
\begin{schema}{Register}
    enrolled: \power STUDENT \\
    completed: \power STUDENT
\where
    \mbox{\dots\ Invariant \dots}
\end{schema}
```

Register
$enrolled : \mathbb{P}\, STUDENT$
$completed : \mathbb{P}\, STUDENT$

. . . Invariant . . .

A specification document might contain an informal definition of the *Register* schema like this one, followed by a discussion of what invariant might be appropriate, and finally the proper definition of *Register*. By using the `%%unchecked` directive, the writer can have both schema boxes printed like any other, but have the type checker take notice of only the second, complete one.

A sort of opposite to `%%unchecked` is the `%%` directive. This is useful for including short paragraphs that are processed by the type checker but should not appear in the output:

```
We give the name $FN$
%% \begin{zed} [FN] \end{zed}
to the set of all file names.
```

We give the name *FN* to the set of all file names.

Here the author wanted to avoid cluttering up the specification as printed with the formal definition of *FN*, but needed to include the definition for the sake of the type checker.

A second use for `%%` is in combination with `%%unchecked`. In the *Register* example, the whole schema is ignored by the type checker, and this may cause problems if a complete definition is not included later, and the declaration part of the schema is needed for type-checking the rest of the specification. The best thing in this case is to include a brief definition of *Register* using `%%`:

```
%% \begin{schema}{Register}
%%     enrolled, completed: \power STUDENT
%% \end{schema}
```

This can be put next to the unchecked but visible definition. This technique is sometimes useful when it seems clearer to present the parts of a specification in a different order from the one that would be needed to follow strictly the principle of definition before use.

Another use for the %% directive is to introduce local variables for a predicate:

```
A name $n$ is included in the output exactly if
\begin{zed}
%% \exists BirthdayBook; n: NAME; today: DATE @
    birthday(n) = today
\end{zed}
```

A name $n$ is included in the output exactly if

$$birthday(n) = today$$

The existential quantifier is included only to introduce the variables used in the predicate; it is used for its effect of declaring some variables rather than for its logical meaning. In such a simple example as this one, it would be easier to exclude the predicate from checking, either by marking it with `%%unchecked`, or by using `\[ ... \]` in place of the `zed` environment. In more complicated examples, however, this technique allows more thorough type-checking to be done.

6 Error Messages

Most of the messages about type errors produced by *fuzz* come with supporting evidence about the expressions and types involved. These expressions and types are obtained from the internal data structures that *fuzz* uses to represent them, so they may differ slightly from the actual forms written in the specification. For example, expressions tend to have extra brackets added to clarify the precedence of operators.

This corroborative detail makes the meaning of most error messages clear; but here anyway is a list of the semantic error messages *fuzz* produces, together with a little more information about each one. The numbers in square brackets give the pages in the ZRM where relevant rules of the language are explained.

Application of a non-function
In an expression of the form $f(E)$, the sub-expression f does not have type $\mathbb{P}(t_1 \times t_2)$ for some types t_1 and t_2. [60]

Argument k has wrong type
In an expression of the form $f(E_1, E_2, \ldots, E_n)$, the type of E_k does not match the type expected for the k'th argument of f. Also applies to expressions of the form $E_1 \, f \, E_2$, where f is an infix function symbol. [60]

Argument k of $\times$ must be a set
In an expression $E_1 \times E_2 \times \cdots \times E_n$, the sub-expression E_k does not have type $\mathbb{P}\, t$ for some type t. [56]

Argument of application has wrong type
In an expression $f(E)$, the type of E is different from the type expected for the argument of f. [56]

Argument of $\mathbb{P}$ must be a set
In an expression $\mathbb{P}\, E$, the sub-expression E does not have type $\mathbb{P}\, t$ for some type t. [56]

Argument of postfix operator has wrong type
In an expression $E\,\omega$, where ω is a postfix function symbol, the type of E is not the type expected for the argument of ω. [65]

Argument of prefix relation has wrong type
In a predicate $R\ E$, where R is a prefix relation symbol, the type of E is not the type expected for the argument of R. [73]

Argument of selection must have schema type
In an expression $E.x$, the sub-expression E does not have a schema type. [61]

Basic type X cannot have parameters
A reference to the basic type X has actual generic parameters. Only generic constants and schemas can have parameters. [80]

Basic type name X multiply declared
The basic type name X appears more than once in the same basic type definition or formal generic parameter list. The second occurrence is ignored. [47, 79, 80]

Component x has wrong type in schema reference
A schema reference is used as a predicate, and the type of the component x in the schema is different from the type x has in the current context. [72]

Constructor name x multiply declared
The constructor name x appears in more than one branch of the same free type definition. The second occurrence is ignored. [82]

Decoration ignored in schema reference
A schema reference used as an expression contains a non-empty decoration. The decoration is ignored. [63]

Function expects n arguments
In an expression $f(E_1, E_2, \ldots, E_k)$, the number of arguments supplied, k, is different from the number n expected by f. [60]

Generic constant x expects n parameters
In an expression $x[E_1, E_2, \ldots, E_k]$, the number of parameters supplied, k, is different from the number n expected by x. Also applies to expressions of the form $E_1\ x\ E_2$ or $x\ E$ where x is an infix or prefix generic symbol. [80]

Global name x multiply declared
The name x has more than one global definition. The second definition overrides the first. [36]

Hiding non-existent component x
In a schema expression which hides certain components of a schema, the component x being hidden does not exist. Applies to the quantifiers $\forall$, $\exists$ and $\exists_1$, and the explicit hiding operator $\setminus$. [76]

Identifier x is not declared
No declaration or definition of x is in scope. [34]

Implicit parameters not completely determined
The actual parameters of a generic constant have been left implicit, but they are not completely determined by the context. Also applies to set, sequence and bag displays. [81]

Infix operator ω is not a function
In an expression $E_1\ \omega\ E_2$, where ω is an infix function symbol, ω does not have the right type to be a function. [65]

Infix relation symbol R is not a binary relation
In a predicate $E_1\ R\ E_2$, where R is an infix relation symbol, R does not have type $\mathbb{P}(t_1 \times t_2)$ for some types t_1 and t_2. [73]

Left argument of operator ω has wrong type
In an expression $E_1\ \omega\ E_2$, where ω is an infix function symbol, the type of E_1 does not match the type expected for the left argument of ω. [65]

Let variable x multiply declared
In a let-expression **let** $x_1 == E_1;\ x_2 == E_2 \ldots$, the same identifier appears more that once among the left-hand sides x_1, x_2, $\ldots$. [59]

Name x is not a schema
The name x appears in a schema reference, but it is not defined as a schema. [50]

Postfix operator ω is not a function
In a predicate $E\,\omega$, where ω is a postfix function symbol, ω does not have the right type to be a function. [65]

Prefix relation symbol R is not a set
In a predicate $R\ E$, where R is a prefix relation symbol, R does not have a set type. [73]

Renamed component x does not exist
In a schema reference $S[y/x, \ldots]$, there is no component of S with the name x. [50]

Renaming ignored in schema reference
A schema reference used as an expression contains a non-empty renaming part. The renaming part is ignored. [63]

Right argument of operator ω has wrong type
In an expression $E_1\ \omega\ E_2$, where ω is an infix function symbol, the type of E_2 does not match the type expected for the right argument of ω. [65]

Schema S expects n parameters
In a schema reference $S[E_1, E_2, \ldots, E_k]$, the number of actual generic parameters, k, is different from the number n expected by S. [79]

Schema S is not defined
The schema S appearing in a schema reference has not been defined. [50]

Selecting non-existent component x
In an expression $E.x$, the name x is not one of the components in the type of E. [61]

Set-valued expression required in actual generic parameter
Set-valued expression required in declaration
Set-valued expression required in free type definition
An expression E appearing as an actual generic parameter (as in $A[E]$), in a declaration (as in $x : E$), or in a free type definition – as in

$$T ::= \ldots \mid f\langle\!\langle E\rangle\!\rangle \mid \ldots$$

– does not have a set type. [51, 79, 80, 82]

Tame function x has no global definition
Tame function x is not a generic function
The name x appears in a `%%tame` directive, but either it has no global definition, or it is not a generic function. The directive is ignored.

Type abbreviation x has no global definition
Type abbreviation x is not a set
The name x appears in a `%%type` directive, but either it has no global definition, or it does not have a set type. The directive is ignored.

Type mismatch in bag display
Type mismatch in sequence display
Type mismatch in set display
In a set display $\{E_1, E_2, \ldots, E_n\}$, the expressions E_i do not all have the same type. Also applies to sequence and bag displays. [55, 66]

Type mismatch in conditional expression
In a conditional expression **if** P **then** E_1 **else** E_2, the types of E_1 and E_2 are different. [64]

Type mismatch in declarations of x
In a declaration part, or a schema expression, there are two declarations of x which give it different types. [31, 51]

Type mismatch in hiding variable x
In a quantified schema expression $\forall D \mid P \bullet S$, the type of component x in the schema S is different from the type it is given by the declaration D. Also applies to the quantifiers $\exists$ and $\exists_1$. [76]

Type mismatch in left argument of infix relation
Type mismatch in right argument of infix relation
In a predicate $E_1\ R\ E_2$, where R is an infix relation symbol, either E_1 or E_2 has a type different from that expected by R. [73]

Type mismatch in piping
In a piping $S_1 \gg S_2$, the type of some output $x!$ of S_1 is different from the type of the input $x?$ of S_2. [78]

Type mismatch in sequential composition
In a sequential composition $S_1 \fatsemi S_2$, the type of some component x' in S_1 is different from the type of x in S_2. [78]

Types do not agree in equation
In a predicate $E_1 = E_2$, the left hand side has a different type from the right hand side. [68]

Types do not agree in set membership
In a predicate $E_1 \in E_2$, the type of the right hand side is not the same as $\mathbb{P}\, t$, where t is the type of the left hand side. [68]

Variable x cannot have parameters
A reference to the variable x has actual generic parameters. Only generic constants and schemas can have parameters. [80]

Warning – implicitly quantified name x appears only once
Under the `-q` switch, an implicit quantifier has been inserted for the name x, which occurs in a predicate only once. This is allowed, but may indicate a spelling mistake.

Warning – infix relation symbol ω is not declared as a binary relation
Warning – infix function symbol ω is not declared as a binary function
Warning – postfix function symbol ω is not declared as a function
Warning – prefix relation symbol ω is not declared as a set
The symbol ω has been declared with a type that is not appropriate to its syntactic class. This is allowed, but will cause an error if the symbol is ever used.

Warning – X already declared as a basic type
The identifier X appears to the left of ::=, but has been declared as a basic type earlier in the specification. *f*UZZ allows this so that mutually recursive free type definitions can be written.

If there are errors in the specification, the fictitious type `*errtype*` may appear in the types reported for variables, or in error messages. This indicates that the type checker could not ascribe a type to some expression in the specification, and replaced its type with a special marker.

The type `*errtype*` never appears in the output produced from a correct specification; it is simply a device which allows more checking to be done on a specification after errors have been detected.

Types appearing in error messages may also contain 'place-markers', indicated by `?` . These place-markers are generated when generic constants are written without explicit parameters, and the type checker works out from the context what the parameters must be. If `?` appears in an error message, it means that the type checker has not completely worked out the type of a phrase, but the type it has found so far has the wrong shape to match the context. For example, the predicate

$$\varnothing = 3$$

results in the error message

```
Types do not agree in equation
> Predicate: \emptyset = 3
> LHS type:  P ?
> RHS type:  NN
```

This indicates that $\varnothing$ always has a set type, but whatever type is given to the elements of the set, this cannot be the same as the type $\mathbb{N}$ ascribed to 3.

7 Syntax Summary

This chapter contains a syntax summary of the input language recognized by the *f*UZZ type checker, showing the LaTeX commands for the constructs of Z rather than the way they look when printed. Except for the use of LaTeX commands in place of the printed symbols, this syntax is identical with the one in the ZRM. Only the formal parts of the specification are shown in the summary, but of course one Paragraph should be separated from the next by plenty of explanatory text.

The following conventions about repeated and optional phrases are used: `S, ..., S` stands for a list of one or more instances of the class S separated by commas, and `S; ...; S` stands for one or more instances of S separated by semicolons. The notation S ... S stands for one or more adjacent instances of S with no separators. Phrases enclosed in slanted square brackets *[* ... *]* are optional.

Certain collections of symbols have a range of binding powers: they are the logical connectives, used in predicates and schema expressions, the special-purpose schema operators, and infix function symbols, used in expressions. The relative binding powers of the logical connectives are indicated by listing them in decreasing order of binding power; the binding powers of infix function symbols are given in Section 3.2. Each production for which a binding power is relevant has been marked with an upper-case letter at the right margin; 'L' marks a symbol which associates to the left – so $A \land B \land C$ means $(A \land B) \land C$ – and 'R' marks a symbol which associates to the right. Unary symbols are marked with 'U'.

```
Specification  ::= Paragraph ... Paragraph

Paragraph      ::= Unboxed-Para
                |  Axiomatic-Box
                |  Schema-Box
                |  Generic-Box

Unboxed-Para   ::= \begin{zed}
                       Item Sep ... Sep Item
                   \end{zed}

Item           ::= [Ident, ..., Ident]
                |  Schema-Name[Gen-Formals] \defs Schema-Exp
                |  Def-Lhs == Expression
                |  Ident ::= Branch | ... | Branch
                |  Predicate

Axiomatic-Box  ::= \begin{axdef}
                       Decl-Part
                 [ \where
                       Axiom-Part   ]
                   \end{axdef}

Schema-Box     ::= \begin{schema}{Schema-Name}[Gen-Formals]
                       Decl-Part
                 [ \where
                       Axiom-Part   ]
                   \end{schema}

Generic-Box    ::= \begin{gendef}[Gen-Formals]
                       Decl-Part
                 [ \where
                       Axiom-Part   ]
                   \end{gendef}

Decl-Part      ::= Basic-Decl Sep ... Sep Basic-Decl

Axiom-Part     ::= Predicate Sep ... Sep Predicate

Sep            ::= ; | \\ | \also
```

```
Def-Lhs       ::= Var-Name[Gen-Formals]
                | Pre-Gen Decoration Ident
                | Ident In-Gen Decoration Ident
Branch        ::= Ident
                | Var-Name \ldata Expression \rdata
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                          U
                | \pre Schema-Exp-1                           U
                | Schema-Exp-1 \land Schema-Exp-1             L
                | Schema-Exp-1 \lor Schema-Exp-1              L
                | Schema-Exp-1 \implies Schema-Exp-1          R
                | Schema-Exp-1 \iff Schema-Exp-1              L
                | Schema-Exp-1 \project Schema-Exp-1          L
                | Schema-Exp-1 \hide
                      (Decl-Name, ... , Decl-Name)            L
                | Schema-Exp-1 \semi Schema-Exp-1             L
                | Schema-Exp-1 \pipe Schema-Exp-1             L
                | ( Schema-Exp )
Schema-Text   ::= Declaration [| Predicate]
Schema-Ref    ::= Schema-Name Decoration [Gen-Actuals][Renaming]
Renaming      ::= [Decl-Name/Decl-Name, ... , Decl-Name/Decl-Name]
Declaration   ::= Basic-Decl; ... ; Basic-Decl
Basic-Decl    ::= Decl-Name, ... , Decl-Name : Expression
                | Schema-Ref
Predicate     ::= \forall Schema-Text @ Predicate
                | \exists Schema-Text @ Predicate
                | \exists_1 Schema-Text @ Predicate
                | \LET Let-Def; ... ; Let-Def @ Predicate
                | Predicate-1
```

```
Predicate-1   ::= Expression Rel Expression Rel ... Rel Expression
                | Pre-Rel Decoration Expression
                | Schema-Ref
                | \pre Schema-Ref
                | true
                | false
                | \lnot Predicate-1                           U
                | Predicate-1 \land Predicate-1               L
                | Predicate-1 \lor Predicate-1                L
                | Predicate-1 \implies Predicate-1            R
                | Predicate-1 \iff Predicate-1                L
                | ( Predicate )
Rel           ::= = | \in | In-Rel Decoration | \inrel{ Ident }
Let-Def       ::= Var-Name == Expression
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 R
                | Expression-2 \cross Expression-2
                      \cross ... \cross Expression-2
                | Expression-2
Expression-2  ::= Expression-2 In-Fun Decoration Expression-2 L
                | \power Expression-4
                | Pre-Gen Decoration Expression-4
                | - Decoration Expression-4
                | Expression-4 \limg Expression-0 \rimg Decoration
                | Expression-3
Expression-3  ::= Expression-3 Expression-4
                | Expression-4
```

```
Expression-4  ::= Var-Name[Gen-Actuals]
               |  Number
               |  Schema-Ref
               |  Set-Exp
               |  \langle[Expression, ..., Expression]\rangle
               |  \lbag[Expression, ..., Expression]\rbag
               |  ( Expression, ..., Expression )
               |  \theta Schema-Name Decoration [Renaming]
               |  Expression-4 . Var-Name
               |  Expression-4 Post-Fun Decoration
               |  Expression-4 \bsup Expression \esup
               |  ( Expression-0 )

Set-Exp       ::= \{ [Expression, ..., Expression] \}
               |  \{ Schema-Text [@ Expression ] \}

Ident         ::= Word Decoration

Decl-Name     ::= Ident | Op-Name

Var-Name      ::= Ident | (Op-Name)

Op-Name       ::= \_ In-Sym Decoration \_
               |  Pre-Sym Decoration \_
               |  \_ Post-Sym Decoration
               |  \_ \limg \_ \rimg Decoration
               |  - Decoration

In-Sym        ::= In-Fun | In-Gen | In-Rel

Pre-Sym       ::= Pre-Gen | Pre-Rel

Post-Sym      ::= Post-Fun

Decoration    ::= [Stroke ... Stroke]

Gen-Formals   ::= [Ident, ..., Ident]

Gen-Actuals   ::= [Expression, ..., Expression]
```

The syntax summary uses several classes of terminal symbol that are defined as follows:

- a Word is an undecorated name or special symbol. It may be either a non-empty sequence of letters, digits and underscores (written using the `\_` command) that starts with a letter, a non-empty sequence of characters drawn from the list `+-*.=<>`, or a LaTeX command. Some strings that would otherwise be Word's are reserved for other purposes, and others are taken as Schema-Name's or operator symbols.
- a Schema-Name is either a Word that has been defined as a schema, or one of the Greek letters `\Delta` or `\Xi` followed by a single space and a Word.
- the classes In-Fun, Pre-Rel, etc., stand for members of the class Word that have been announced as infix function symbols, prefix relation symbols, etc., either in the prelude or by an explicit directive.
- a Number is a non-empty sequence of decimal digits.
- a Stroke is a single decoration: one of `'`, `?`, `!`, or a subscript digit entered as `_0`, `_1`, and so on.

Another essential Mikro*nella* design

Index