The Z/EVES 2.0 User's Guide

TR-99-5493-06a

Mark Saaltink

Release date: October 1999

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

Contents

Chapter 1

Introduction

This user's guide describes how and why to use Z/EVES to develop or analyse a Z specification. This is not a tutorial on Z; there are already a number of readily available books describing Z and its application (*e.g.*, those by Barden *et al.* [2], Hayes [4], Jackey [6], Potter *et al.* [9], Spivey [13], and Woodcock and Davies [15]). We assume the reader has a solid grounding in the Z language and its use; instead, we focus on the mechanics of using Z/EVES and techniques for using Z/EVES to analyse Z specifications.

This version of the user's guide describes the graphical user interface to Z/EVES. Users of the command line interface should consult the previous version [11] instead.

Detailed descriptions of the syntax used by Z/EVES, the proof commands, and the handling of Z paragraphs can be found in the Z/EVES Reference Manual [7]. A detailed description of the Z/EVES Mathematical Toolkit appears as a separate report [10].

Chapter 2 describes the mechanics of using Z/EVES. Chapter 3 describes several kinds of analysis that can be applied to expose errors in Z specifications or to explore the specification. Chapter 4 describes the logical foundations and basic mechanisms of the Z/EVES theorem prover. Chapter 5 describes how to use the Z/EVES prover. The prover has an automatic mode which can prove many simple theorems. When that fails, however, it is not always obvious how to proceed. This chapter describes the strategies for successful application of the prover. Chapter 6 describes how to define new concepts in a specification in a way that supports proofs.

Prover commands are shown in the text in `typewriter` font, *e.g.*, `rewrite`.

The terms "user", "specifier", and "reader" all refer to Z or Z/EVES users.

Chapter 2

Using Z/EVES

Z/EVES is an interactive system for composing, checking, and analysing Z specifications. Specifications can be entered and checked one paragraph at a time; checked paragraphs can be revised and rechecked; and theorems can be stated and proofs attempted at any time. Z/EVES is also able to read entire files of specifications that have been previously prepared using LaTeX markup. This chapter describes the mechanics of using Z/EVES.

2.1 Running Z/EVES

Z/EVES is run on a Unix system by the command `z-eves` (if it has been installed correctly); on Windows it is started by running a shortcut. The installation guide describes how to install and run it.

Z/EVES has two parts: a server that checks paragraphs and executes proof commands, and a graphical interface that manages specifications, sends commands to the server, and displays results. In Windows, the server displays a small window; however, it does not respond to user input or mouse clicks. Usually, one does not need to be aware of these two parts. However, if something goes wrong it may be necessary to terminate one or both of these processes.

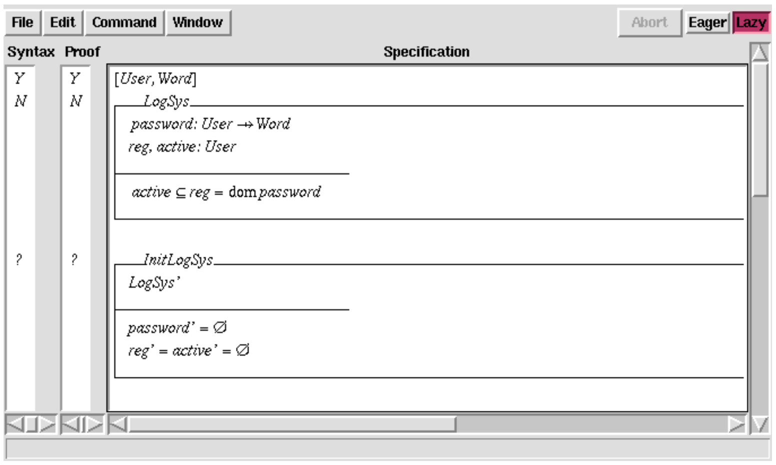
2.2 Z/EVES Displays

Z/EVES has six main displays, five of which can be selected under the "Window" menu in the menu bar. (The Edit window is instead exposed when edit commands are selected, and disappears when editing is complete.)

2.2.1 The Specification window

The Specification window is the main display, and several important functions (such as saving or exiting) are only available from it.

The specification window shows the formal part of a specification. That is, Z paragraphs are shown, but no additional commentary or narrative text. Text can be composed within Z/EVES, but can also be composed externally and imported in to it. Eventually, we intend to support a direct linkage with word processors; at the moment only ASCII files marked up in LaTeX can be imported (see Section 2.5).



```
Y  Y  [User, Word]
N  N  LogSys
      password: User ⇸ Word
      reg, active: User
      ─────────────
      active ⊆ reg = dom password

?  ?  InitLogSys
      LogSys'
      ─────────────
      password' = ∅
      reg' = active' = ∅
```

The status of each paragraph is shown in two columns to the left of the paragraph. The leftmost column shows one of three symbols:

- "?" indicates that the paragraph has not been checked.
- "Y" indicates that the paragraph has been checked and has no syntax or type errors.
- "N" indicates that the paragraph has been checked and has errors.

The next column shows the proof status for the paragraph, using one of three symbols

- "?" indicates that the paragraph has not been successfully checked (so proof status cannot be determined).
- "N" indicates that the paragraph has an associated goal that is unproven.
- "Y" indicates that the paragraph has no unproved goals.

Paragraphs can be added in several ways:

- "New Paragraph" can be selected from the "Edit" menu. This exposes the Edit window (described in Section 2.2.5), which allows the paragraph to be composed. The new paragraph is added to the end of the specification.
- Paragraphs can be imported from a LaTeX file, as described in Section 2.5.
- A "Paste" operation inserts the contents of the clipboard immediately before the paragraph containing the *insertion point* (where the blinking cursor is) in the specification window.

Paragraphs in the specification window can be checked, modified, deleted, or moved. A paragraph can be checked by double clicking on it, or by selecting it, right clicking, and selecting "Check" from the pop-up menu. Selecting "Check up to here" checks all preceding paragraphs before checking the selected paragraph. The entire specification can be checked by selecting "Check all paragraphs" from the "Command" menu. While paragraphs are being checked, most Z/EVES functions are disabled. However, it is still possible to select different windows and to scroll them. The "Abort" button can be used to stop checking.

A paragraph can be modified by selecting it, right clicking, and selecting "Edit" from the pop-up menu. This brings up the Edit window with the selected paragraph as its initial contents. When the edit is finished, the revised paragraph replaces the original. The revised paragraph is not checked. (Following paragraphs will also become not checked, if the original paragraph had been checked.)

A paragraph (or group of paragraphs) can be deleted by selecting it (or them), right clicking, and selecting “Delete” from the pop-up menu. There is no “undo” capability in the interface, so once deleted, the paragraph cannot be recovered. If a deleted paragraph had been checked, all following paragraphs will become not checked.

A paragraph (or group of paragraphs) can be moved by selecting it (them), right clicking, and selecting “Move” from the pop-up menu. This changes the mode of the interface. As the mouse pointer moves over paragraphs, they are darkened. Selecting a paragraph causes the originally selected paragraph or paragraphs to be inserted immediately before the selected location. The move can be canceled by right clicking, then selecting “Cancel” from the pop-up menu.

Paragraph order is important. A Z specification is a sequence of paragraphs and requires declaration before use. The order in which paragraphs appear in the Specification window determines this declaration before use order. Documents often present Z specifications with the paragraphs out of order; if such a specification is imported into Z/EVES it is necessary to move the paragraphs into a suitable checking order. Order is also important for theorems, as the proof of a given theorem can only use other theorems that precede the given theorem in the specification. Thus, lemmas must appear before the theorems whose proofs they are used in.

Z/EVES offers two modes of operation, “Eager” and “Lazy”. In “Eager” mode, the rules of checking are strict: a paragraph can only be checked if all preceding paragraphs have been checked. As one’s goal is to have the entire sequence of paragraphs checked, this is reasonable. “Lazy” mode allows paragraphs to be skipped. This can be useful in experimentation; for example, the specification might contain two alternative versions of a paragraph. In “Lazy” mode, one of these can be checked and the other left unchecked, and the remainder of the specification can be analysed. The status bars help a user to keep track of the experiment, by showing what has been checked and what has not. By default, Z/EVES is in the “Lazy” mode.

When paragraphs have been checked, their phrase structure is browsable in the specification window. When the mouse is clicked over a part of a checked paragraph, the smallest containing phrase (*i.e.*, name, expression, predicate, or paragraph) is highlighted. Successive clicks will cause larger and larger containing phrases to be highlighted. Alternatively, the mouse can be moved with the left button depressed, in which case the smallest phrase containing both the place where the button was first depressed and the current location of the pointer is highlighted. A selected part can be copied to the clipboard by selecting “Copy” from the “Edit” menu in the menu bar.

2.2.2 The Proof window

The Proof window provides three functions: inspection and modification of a proof script; interactive construction of a proof; and proof browsing.

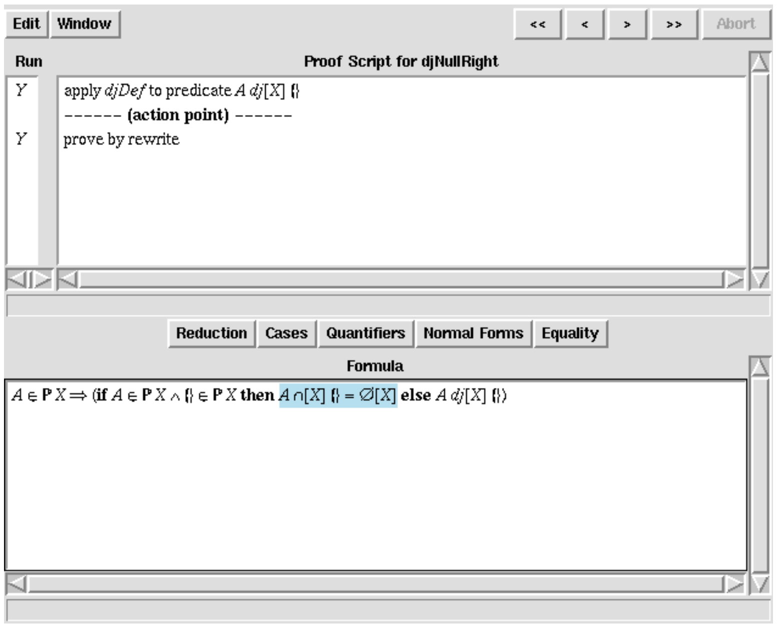
A proof script is a sequence of Z/EVES prover commands. Like the specification, these commands are displayed along with their status, which indicates whether the command has been executed or not. Like paragraphs within a specification, individual steps can be inserted, deleted, moved, or modified.

If a proof script is *active*, its steps can be executed and the proof can be browsed. In this case, below the proof script is a display labeled “Formula”, showing a goal predicate, and the script contains an *action point* that appears in the script. The action point shows what stage in the proof script the formula corresponds to. There are four browsing buttons on the right side of the top menu bar:

- “<<” moves the action point to the start of the proof, thus displaying the original goal predicate;
- “<” moves the action point back one step;
- “>” moves the action point forward one step; and

- ">>" moves the action point to the end of the proof.

The action point can also be moved by selecting it, right clicking, selecting "Move" from the pop-up menu, then clicking on a command or just after the last command. The interstion point will be moved to just before the selected place.



Run | Proof Script for djNullRight

```
Y   apply djDef to predicate A dj[X] {}
    ------ (action point) ------
Y   prove by rewrite
```

Formula

$A \in \mathbb{P}\, X \Rightarrow (\textbf{if } A \in \mathbb{P}\, X \wedge \{\} \in \mathbb{P}\, X \textbf{ then } A \cap[X]\ \{\} = \varnothing[X] \textbf{ else } A\ dj[X]\ \{\})$

The illustrated proof window shows a proof script with two steps, both of which have been run. The user has selected the back button and moved the action point one step back, and the Formula area shows the goal predicate that resulted from the application of the first command.

Commands can be added to a proof script in several ways:

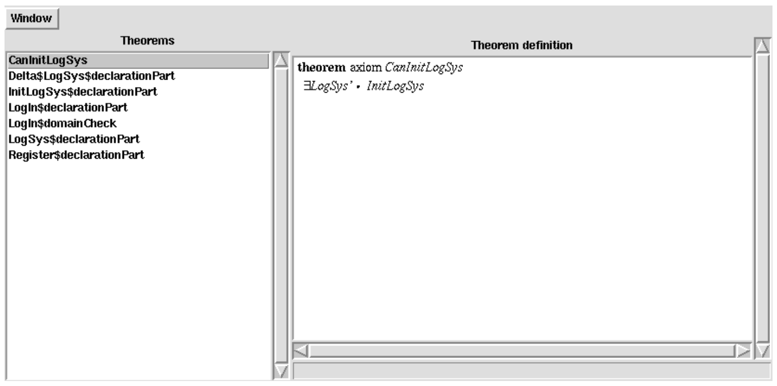
- "New Command" can be selected from the "Edit" menu. This exposes the Edit window, which allows the command to be composed. The new command is added to the end of the script when "Done" is selected from the "File" menu in the Edit window. (Alternatively, "Cancel" can be selected from the "File" menu, in which case the edit is abandoned.)

- A "Paste" operation inserts the contents of the clipboard immediately before the *insertion point* (where the blinking cursor is) in the script window, or at the end if there is no insertion point.

- Some proof commands are available from a menus on a bar above the goal predicate display. These commands are entered at the action point and are immediately executed. Unfortunately, not all prover commands are available from menus, and for effective use of the prover it is necessary to learn the syntax of commands and to use one of the other two techniques to add them to a script.

When a proof command is running, most Z/EVES functions are disabled. However, it is still possible to select different windows and to scroll them. A long-running proof command can be stopped by selecting the "Abort" button.

2.2.3 The Theorems window



```
CanInitLogSys
Delta$LogSys$declarationPart
InitLogSys$declarationPart
LogIn$declarationPart
LogIn$domainCheck
LogSys$declarationPart
Register$declarationPart
```

theorem axiom *CanInitLogSys*
$\exists LogSys' \bullet InitLogSys$

The Theorems window has two main parts. On the left is a list of theorem names. Double clicking on one of these names causes the theorem to be displayed in the right part of the Theorem window. As for checked paragraphs in the Specification window, the phrase structure of the displayed theorem can be explored.

Theorems correspond to theorem paragraphs in a specification or are generated when a paragraph is checked. Generated theorems can be *axioms*, which are facts that may be useful in later proofs, or *goals* that are in need of proof. These goals are either explicit theorem paragraphs, or domain checks generated as described in Section 3.1.2.

2.2.4 The Proof Scripts window

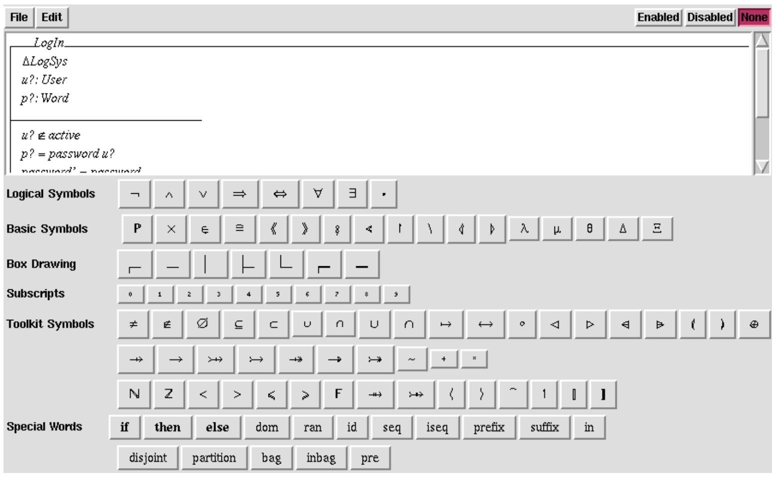
The Proof Scripts window contains a list of proof scripts associated with the specification. Each proof script is sequence of proof commands for a goal. The script can be viewed (in a Proof window) by selecting the script's name, right clicking, and selecting "View" from the pop-up menu.

Proof scripts are retained even when the associated goal disappears, which can happen when a paragraph is deleted or becomes not checked. The script is retained so that the proof can be reexecuted if the paragraph is checked again and the goal exists again.

A proof script is *active* if its goal exists. Inactive scripts can be deleted by selecting the script's name, right clicking, and selecting "Delete" from the pop-up menu.

2.2.5 The Edit window

The Edit window allows new paragraphs to be composed or existing paragraphs to be modified. The editor provides standard editing capabilities such as insertion and deletion of characters, cut and paste, motion using cursor keys, and mouse selection. In addition, it provides a palette of Z symbols below the edit area. Clicking on a symbol inserts it into the edit area at at the insertion point (where the cursor is).



```
LogIn
ΔLogSys
u?: User
p?: Word
────────
u? ∉ active
p? = password u?
password' = password
```

Boxed paragraphs can be entered by first selecting the appropriate box type from the "Edit" menu, which clears the editor and inserts a schema, axiomatic, or generic box. The name, declarations , and predicates can then be inserted into the appropriate parts of the box. Alternatively, boxes can be drawn by selecting the individual box characters from the palette.

Keyboard shortcuts are also available for most special characters. For all letters except j and v, control-letter is bound to the corresponding Greek letter (control-shift-letter to the capitalized Greek letter). The most commonly used Greek letters are in the following table.

Symbol	λ	μ	θ	Δ	Ξ
Key	Ctl-l	Ctl-m	Ctl-q	Ctl-D	Ctl-X

Other symbols are available as shown in the following table:

Symbol	$\neg$	$\wedge$	$\vee$	$\Rightarrow$	$\Leftrightarrow$	$\forall$	$\exists$	$\bullet$
Key	Alt-!	Alt-&	Alt-v	Alt-)	Alt-=	Alt-A	Alt-E	Alt-.
Symbol	$\mathbb{P}$	$\times$	$\in$	$\emptyset$	$\subseteq$	$\subset$	$\bigcup$	$\bigcap$
Key	Alt-P	Alt-x	Alt-e	Alt-0	Alt-z	Alt-c	Alt-U	Alt-I
Symbol	$\leftrightarrow$	$\rightarrow$	$\circ$	$\lhd$	$\rhd$	⩤	⩥	
Key	Alt-j	Alt-f	Alt-o	Alt-r	Alt-t	Alt-y	Alt-u	
Symbol	$\mathbb{N}$	$\mathbb{Z}$	$<$	$>$	$\mathbb{F}$	$\frown$	↿	
Key	Alt-N	Alt-Z	Alt-<	Alt->	Alt-F	Alt-^	Alt-@	

Several Z characters are similar in appearance and should be carefully distinguished.

- The infix union function $\cup$ is slightly smaller than the generalized union function $\bigcup$.
- The infix intersection function $\cap$ is slightly smaller than the generalized intersection function $\bigcap$.
- The sequence concatenation function $\frown$ is different from the caret ^, which is not used in Z.
- The schema operator $\semi$ is different from the semicolon ; (which separates declarations and is relational composition).

2.2.6 The Clipboard window

The Clipboard window shows the contents of the Z/EVES clipboard, which is set by a "Cut" or "Copy" operation and used by a "Paste" operation. The clipboard contents can be edited; the editor is the same as used in the Edit window. The clipboard window has the same appearance as the Edit window.

2.3 Composing and Checking a Specification

We will use a small example, a logging system adapted from Exercise 6.2 in Potter, Sinclair, and Till's book [9] to illustrate the use of Z/EVES. For the sake of brevity, we will use only the first few paragraphs of that specification.

$$[User, Word]$$

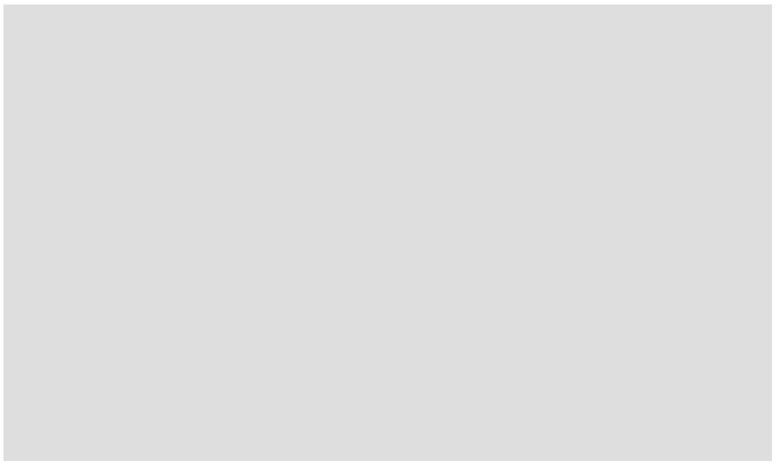
$$\begin{array}{|l}
\textit{LogSys} \\
\hline
password : User \mathrel{+\!\!\!\!\rightarrow} Word \\
reg, active : \mathbb{P}\, User \\
\hline
active \subseteq reg = \operatorname{dom} password \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{InitLogSys} \\
\hline
LogSys' \\
\hline
password' = \emptyset \\
active' = reg' = \emptyset \\
\hline
\end{array}$$

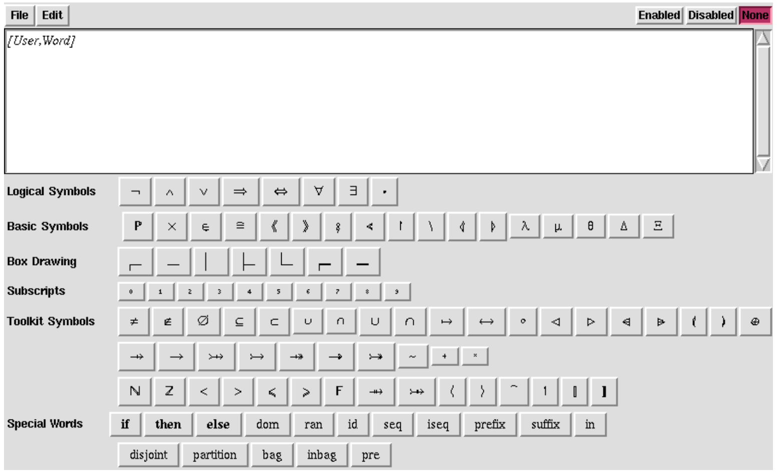
$$\begin{array}{|l}
\textit{Register} \\
\hline
\Delta LogSys \\
u? : User;\ p? : Word \\
\hline
password' = password \cup \{u? \mapsto p?\} \\
active' = active \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{LogIn} \\
\hline
\Delta LogSys \\
u? : User \\
p? : Word \\
\hline
u? \notin active \\
p? = password(u?) \\
password' = password \\
active' = active \cup \{u?\} \\
\hline
\end{array}$$

We begin by running Z/EVES, which will eventually display an empty Specification window.

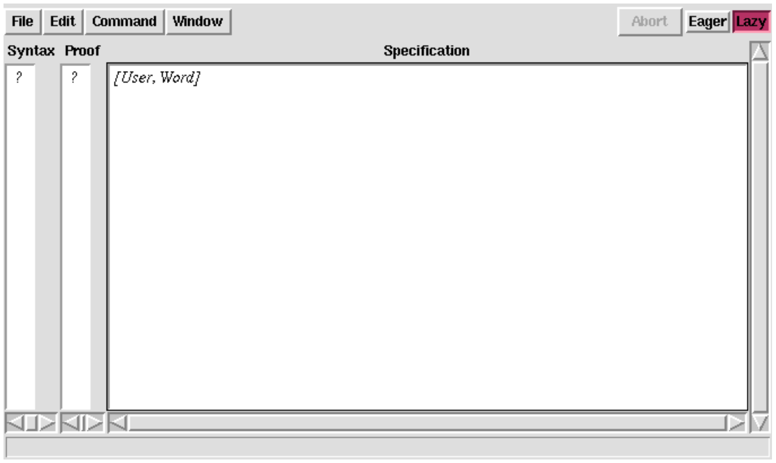


Selecting "New Paragraph" from the "Edit" menu brings up the Edit window. After clicking the mouse in the edit area, we can type "[User, Word]".



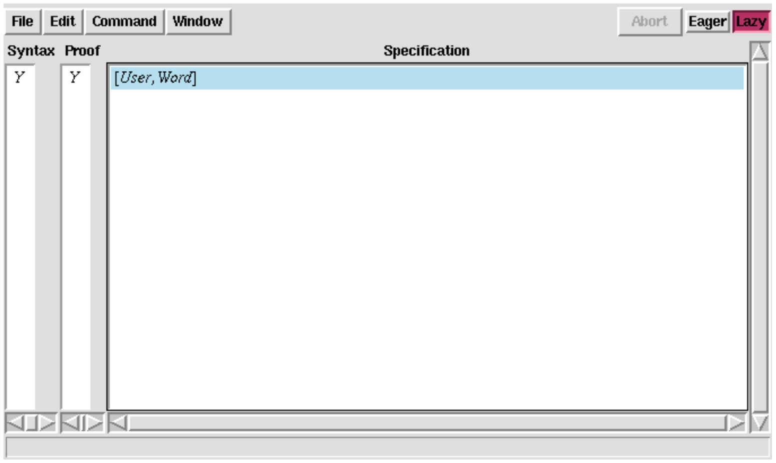
[User,Word]

After select "Done" from the "File" menu in the Edit window, the Specification window now shows the new paragraph:



Syntax	Proof	Specification
?	?	*[User, Word]*

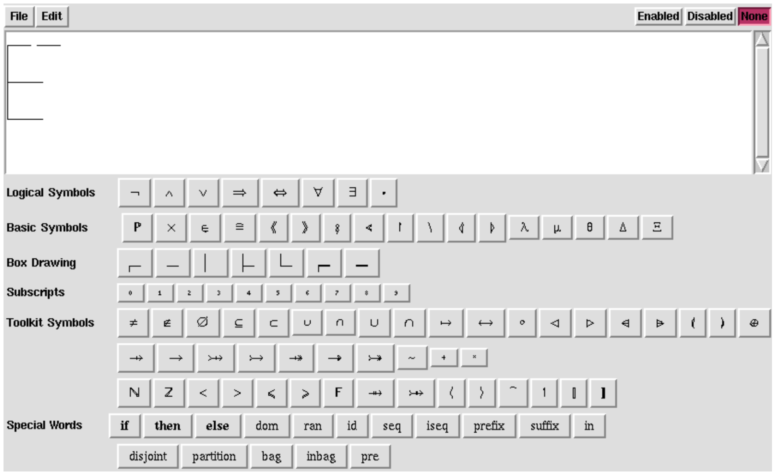
The two status bars to the left of the paragraph show question marks, showing that it is not known to be correct. We can check the paragraph by double clicking on it, or by right clicking on it and selecting "Check" from the pop-up menu. When the paragraph is checked, the status bars are updated:



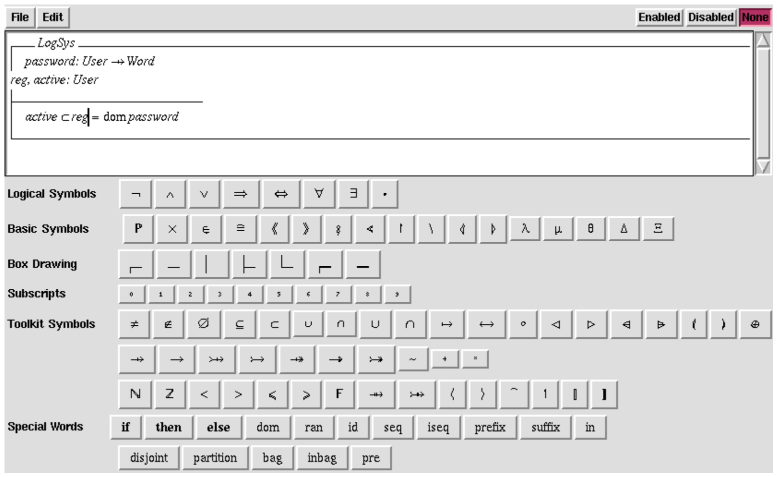
Syntax	Proof	Specification
Y	*Y*	$[User, Word]$

(The paragraph is also selected and so is highlighted.) Both columns contain "Y": the syntax and type check succeeded, and there is nothing interesting to prove.

The state schema can be added similarly. Selecting "New Paragraph" from the "Edit" menu again brings up the Edit window. The easiest way to enter the schema is to select "Schema Box" from the "Edit" menu, which draws the schema box outlines:



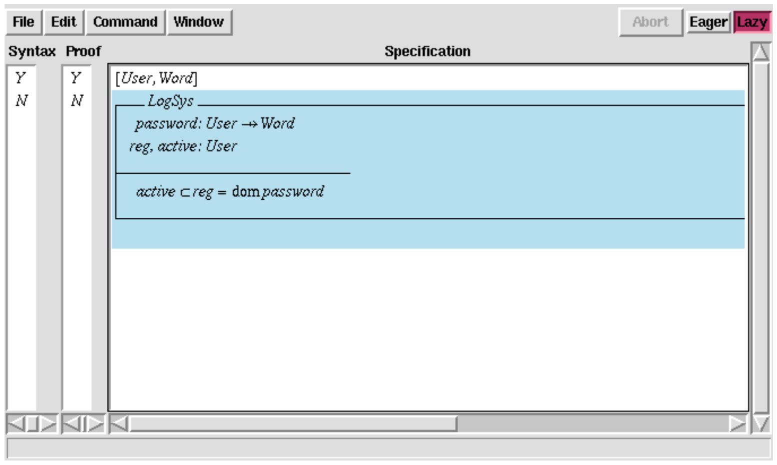
We can then click in the top line, type in the schema name "LogSys", click in the declaration part of the box and type in the declarations, then click in the predicate part and type in the schema's constraints. Special characters can be entered by clicking on them in the palette below the edit area. The results (including two typing mistakes included for illustrative purposes) appear as follows:



LogSys
password: User ⇸ Word
reg, active: User

active ⊂ reg = dom password

Note that when a new line is started in the schema box, the box is not drawn. This does not matter; the schema will be correctly rendered in the Specification window. Selecting "Done" from the "File" menu adds the paragraph to the specification. Double clicking on the paragraph in the Specification window checks it:

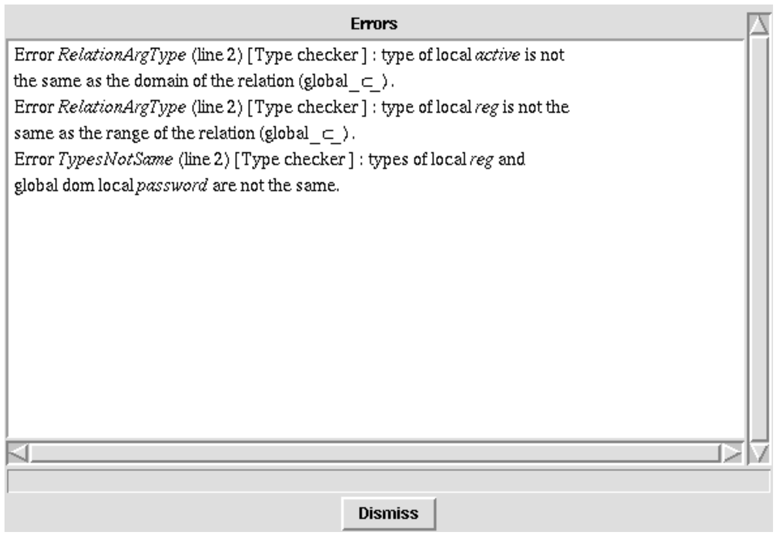


Syntax	Proof	Specification
Y	Y	[User, Word]
N	N	LogSys

LogSys
password: User ⇸ Word
reg, active: User

active ⊂ reg = dom password

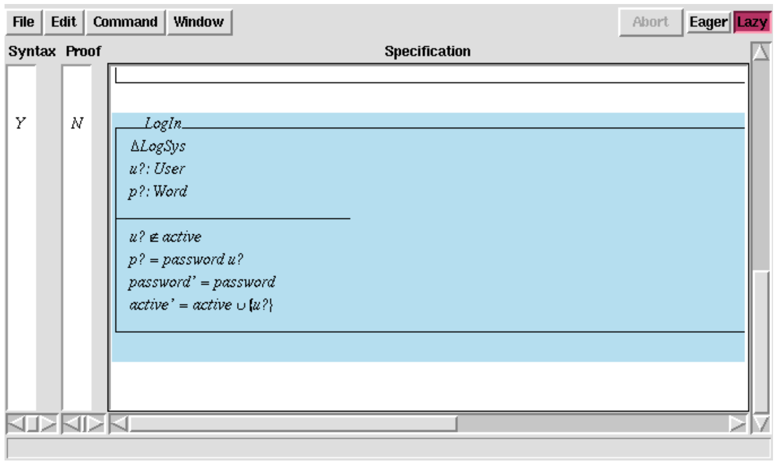
This time, there are errors. Right clicking on the paragraph and selecting "Show Errors" from the pop-up menu shows the error messages:



Error *RelationArgType* (line 2) [Type checker] : type of local *active* is not the same as the domain of the relation (global _ ⊆ _).
Error *RelationArgType* (line 2) [Type checker] : type of local *reg* is not the same as the range of the relation (global _ ⊆ _).
Error *TypesNotSame* (line 2) [Type checker] : types of local *reg* and global dom local *password* are not the same.

Evidently, *active* and *reg* do not have the correct type for the subset relation, and the two sides of the equality have different types. On inspection we can see we omitted the $\mathbb{P}$ operation in the declaration of *active* and *reg* (that is, we wrote $reg, active : User$ instead of $reg, active : \mathbb{P}\, User$). This is readily fixed; we can right click on the paragraph and select "Edit" from the pop-up menu. When the Edit window appears, we can click on the "U" in "User"; a vertical cursor appears just before it. Then selecting $\mathbb{P}$ from the palette inserts it, so that the declaration is correct. Selecting "Done" from the "File" menu brings back the Specification window, with a revised definition of *LogSys*. The status bars both hold question marks, since the revised version of *LogSys* has not yet been checked. After double clicking on the revised paragraph, the check succeeds and both status bars show "Y".

Continuing in this way, we can add and check the remaining three paragraphs shown at the start of this section. When schema *Login* is checked, the syntax status bar holds a "Y", as there are no syntax or type errors, but the "Proof" bar holds a "N".



LogIn
$\Delta LogSys$
$u? : User$
$p? : Word$

$u? \notin active$
$p? = password\ u?$
$password' = password$
$active' = active \cup \{u?\}$

This paragraph has an associated goal that has not been proved. This is a *domain check* associated with the paragraph, as explained in Section 3.1.2. We will ignore it for now.

The work done so far can be saved by selecting "Save" or "Save As ..." from the "File" menu. Work on the specification can be continued later by opening the saved file.

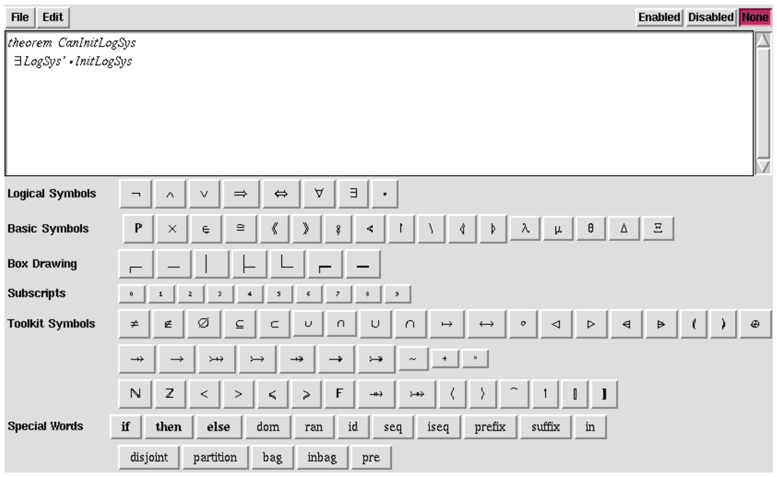
2.4 Analysing a Specification

As will be explained in more detail in Chapter 3, Z/EVES supports various kinds of analysis that go beyond type checking. We will illustrate the mechanics of doing the checks here, and explain their significance in Chapter 3.

One simple check is to see whether the initial state is satisfiable. This will show both that the state schema is consistent and that there are possible initial states. We can add the following conjecture asserting this:

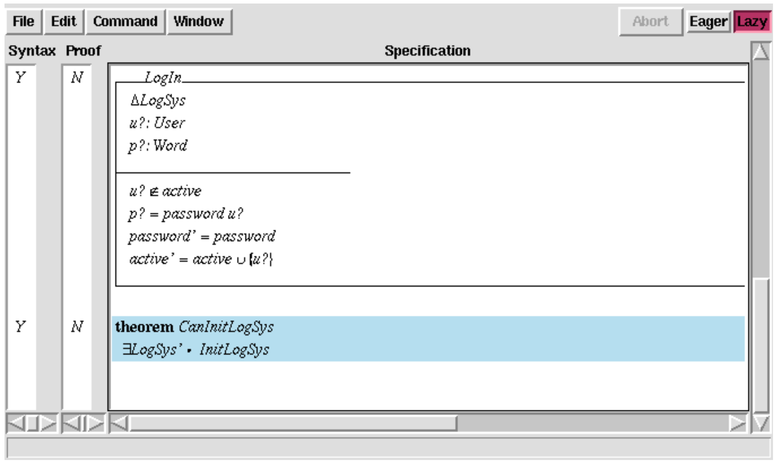
theorem CanInitLogSys
$\exists LogSys' \bullet InitLogSys$

As for other paragraphs, this can be added by selecting "New Paragraph" from the "Edit" menu, clicking the mouse in the edit area when the Edit window appears, and typing in the theorem using a combination of keystrokes and selections from the palette.



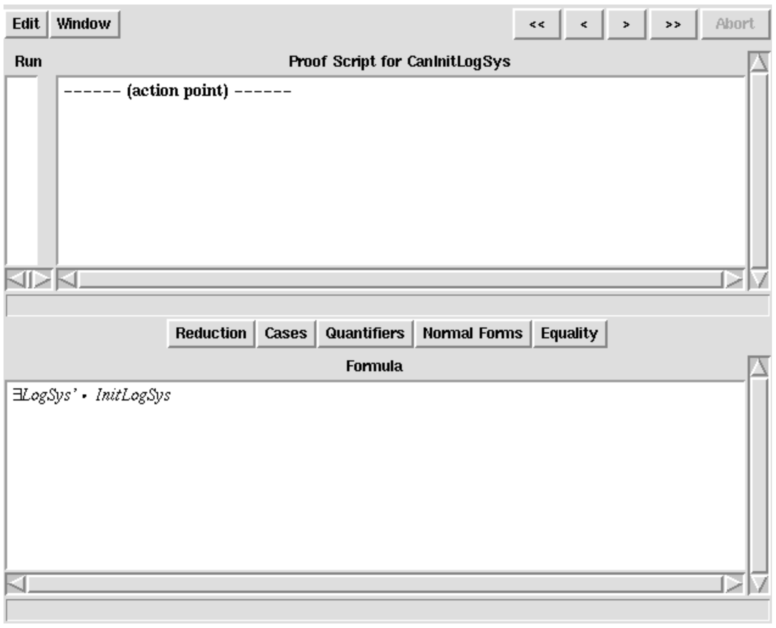
theorem CanInitLogSys
$\exists LogSys' \bullet InitLogSys$

When "Done" is selected from the "File" menu, this theorem is added to the specification, and the specification window reappears. When the theorem paragraph is checked, the status bar shows it to be free of errors but not proved:



Syntax	Proof	Specification
Y	N	*LogIn* $\Delta LogSys$ $u?: User$ $p?: Word$ $u? \notin active$ $p? = password\ u?$ $password' = password$ $active' = active \cup \{u?\}$
Y	N	**theorem** *CanInitLogSys* $\exists LogSys' \bullet InitLogSys$

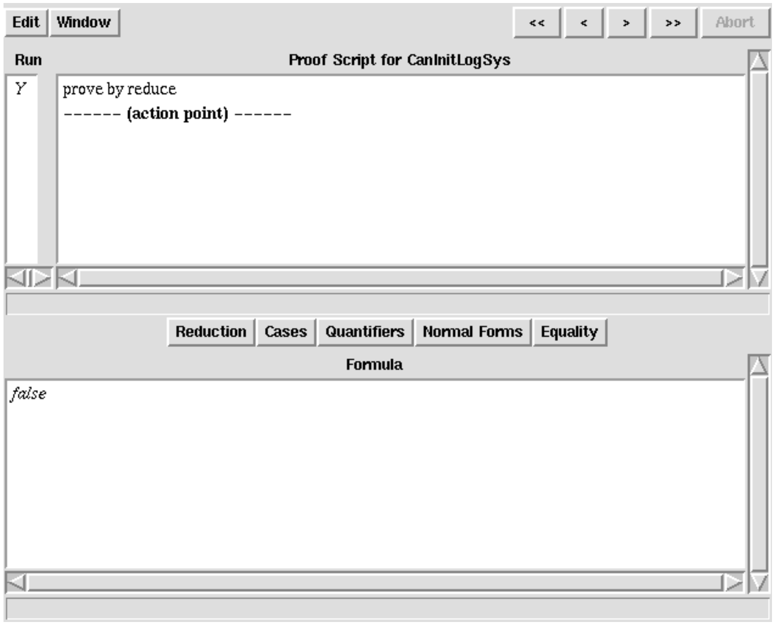
The proof can be started by selecting the theorem in the Specification window, right clicking, and selecting "Show proof". This exposes the Proof window:



```
------ (action point) ------
```

$\exists LogSys' \bullet InitLogSys$

The theorem's goal predicate is shown in the bottom half of the window. An empty proof script occupies the top half. As the proof of this theorem involves using the definitions of the schemas mentioned, we can apply a powerful automatic proving capability by selecting "Prove by reduce" from the "Reduction" menu appearing in the middle bar of the Proof window. This results in the following display:

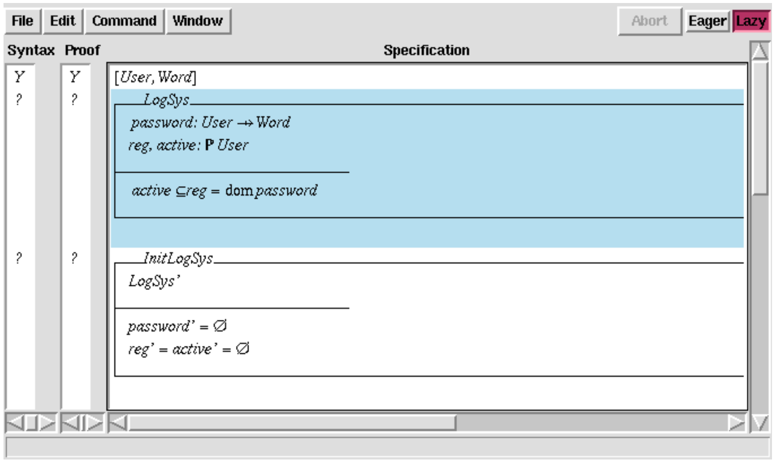


```
Y   prove by reduce
    ------ (action point) ------
```

false

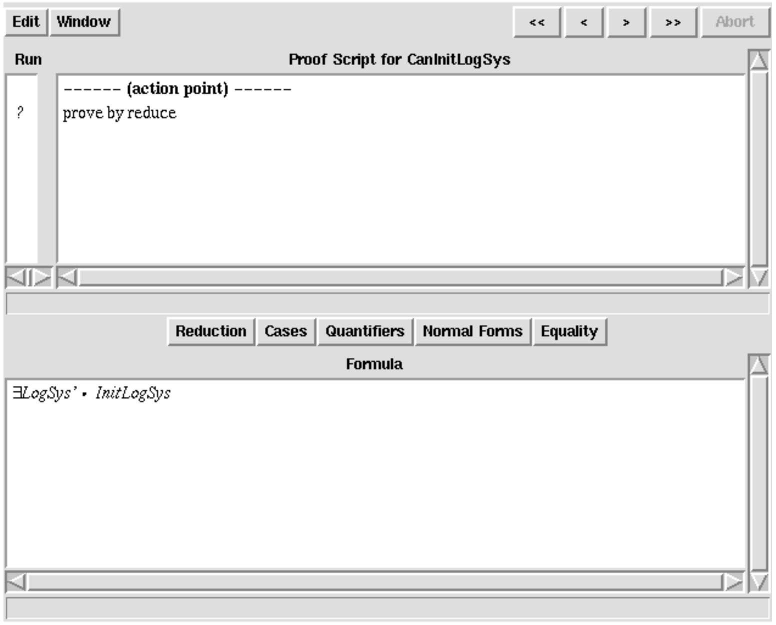
The proof script now contains the proof command we just selected; the "action point" is after that step, so the formula window shows the result of the step. This result is the predicate *false*—the conjecture is not, after all, a theorem.

We can return to the specification window by selecting "Specification" from the "Window" menu at the top of the Proof window. Inspection of schemas *Logsys* and *InitLogSys* shows the source of the problem: we erroneously wrote $active \subset reg$ instead of $active \subseteq reg$ in schema *Logsys*. This is easily repaired: selecting schema *LogSys* in the Specification window, right clicking, and selecting "Edit" from the pop-up menu exposes the Edit window with the *LogSys* schema in it. The correction is easily made. When "Done" is selected from the "File" menu, we are returned to the Specification window, which now has the following appearance:



Syntax	Proof	Specification
Y	Y	$[User, Word]$
?	?	LogSys: $password: User \rightarrowtail Word$; $reg, active: \mathbb{P}\ User$ | $active \subseteq reg = \mathrm{dom}\ password$
?	?	InitLogSys: $LogSys'$ | $password' = \varnothing$; $reg' = active' = \varnothing$

As the status bars show, the revised version of *LogSys* is not checked. Furthermore, all the following paragraphs are not checked either. Since the modification to *LogSys* might have changed the schema's type, all these paragraphs need to be rechecked with respect to the new definition. These paragraphs can be checked one by one, by couble clicking on them on at a time. Alternatively, "Check all paragraphs" can be selected from the "Commands" menu; this will check all the unchecked paragraphs. There are no syntax or type errors. We can revisit the proof of theorem *CanInitLogSys* by right clicking on it and selecting "Show proof" from the pop-up menu. The old proof commands are retained, but have not yet been executed:

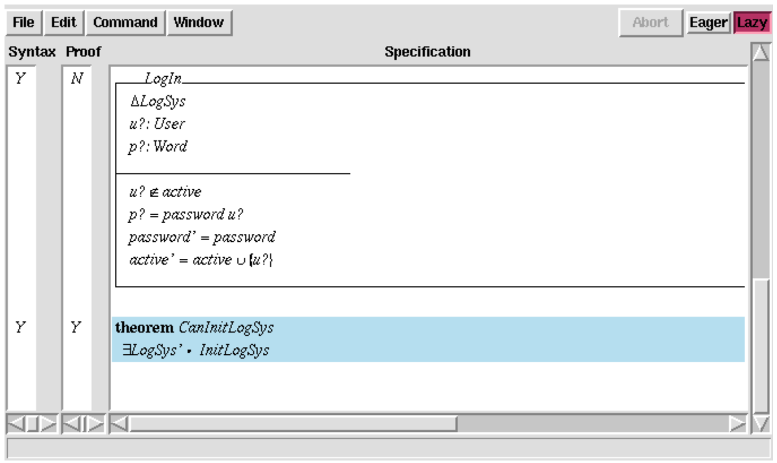


Run	Proof Script for CanInitLogSys
	------ **(action point)** ------
?	prove by reduce

Formula:

$\exists LogSys' \bullet InitLogSys$

Double clicking on the "prove by reduce" command in the script causes it to be run. Now, it results in the predicate *true*, so the conjecture is, indeed, a theorem. If we return to the Specification window, by selecting “Specification” from the “Window” menu, the status bar has been updated to show that the proof is complete.



Syntax	Proof	Specification
Y	N	*LogIn*: $\Delta LogSys$; $u?: User$; $p?: Word$ | $u? \notin active$; $p? = password\ u?$; $password' = password$; $active' = active \cup \{u?\}$
Y	Y	**theorem** *CanInitLogSys* $\exists LogSys' \bullet InitLogSys$

Chapter 3 describes the various analyses supported by Z/EVES in more detail. Chapter 4 describes the Z/EVES prover; Chapter 5 describes techniques for doing proofs.

2.5 LaTeX Markup

Z/EVES can read a LaTeX markup representation of a Z specification, so that files can be typeset as well as analysed. The Z/EVES distribution provides a package file called `z-eves.sty` that defines the necessary commands. This file is compatible with LaTeX2e, and can be applied to a document with the LaTeX command `\usepackage{z-eves}`.

Aside from a small number of syntax extensions for Z/EVES, the markup macros defined by the `z-eves` package are the same as those provided by Spivey’s `zed` style for LaTeX 2.09. Spivey’s guide [14] provides details of the markup form. However, for the `z-eves` package in LaTeX2e, the beginning of a file must be

```
\documentclass{article}
\usepackage{z-eves}
```

Other document classes, *e.g.* `report` or `slides`, are allowed.

A LaTeX specification can be read by the “Import ...” function in the “File” menu. Importing a file adds its paragraphs to the current specification. A file can be imported a second time, in which case the paragraphs are updated. (Paragraphs are identified only by their order of occurrence in the LaTeX file, so this updating only works properly if paragraphs are revised, or new paragraphs are appended.) Some features of LaTeX, such as fuzz declarations (*e.g.*, `%%inrel R`), are not recognnized by the importer.

Chapter 3

Analysing Z Specifications

The use of Z, or any other formal notation, in a specification is a great step forward: natural language specifications are notorious for their ambiguity. Making formal statements about a system can in itself be beneficial, as it can lead to a careful consideration of the important aspects of the system and to the development of a consistent terminology.

However, just because a specification is expressed in a formal notation does not mean it is correct, complete, or even meaningful. There are several different kinds of errors that might appear in a specification, ranging from trivial spelling errors to subtle inconsistencies in meaning. These errors are discussed in Section 3.1. Z/EVES can help specifiers avoid many of these errors.

There is a benefit to a formal specification that goes beyond its precision. Formal specifications can be manipulated, allowing properties of the specification or the specified system to be explored. One simple operation is the "expansion" and simplification of a schema definition, which might help a reader better understand the meaning of a complex specification, or might help a designer when it is not possible to structure the implementation in the way the specification is structured. A specification is a model of the specified system, and can sometimes be used as a sort of prototype. It might be possible, for example, to determine the effect of a certain sequence of operations by calculation. Similarly, it may be possible to show that a system never gets into an undesirable state by stating and proving appropriate theorems. Section 3.2 shows how Z/EVES can be used to manipulate specifications in these and other ways.

3.1 Checking for Errors

Z/EVES can help specifiers avoid many different kinds of error. However, except for syntax and type checking, the analyses described in this section are not mandatory. Domain checking is strongly recommended (and the domain checks are automatically generated), but can be omitted if desired. The user must decide what kind of analysis is appropriate or cost effective.

3.1.1 Syntax and type errors

The Z language has quite a complex syntax, and the Mathematical Toolkit contains dozens of functions. It is easy, especially for an inexperienced Z user, to make a mistake. Z/EVES, like most Z tools, detects and reports such type errors. Unlike many other tools, however, Z/EVES can be used incrementally: as each paragraph of a specification is written, it can be immediately checked and, if necessary, corrected.

$$\begin{array}{|l}
\textit{DataBase} \\
\hline
\textit{data} : \textit{Key} \pfun \textit{Data} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\textit{DeleteRecord} \\
\hline
\Delta \textit{DataBase} \\
k? : \textit{Key} \\
\hline
\textit{data}' = k? \ndres \textit{data} \\
\hline
\end{array}$$

When schema *DeleteRecord* is checked, an error is detected, and the "Syntax" status column shows that there is an error. Selecting "Show errors" from the pop-up menu for the paragraph displays the following message:

Error *FunctionArgType* (line 2) [Type checker]: in application of **global** $(\_ \ndres \_)$, argument 1 has the wrong type.

Unfortunately, the line numbers in error messages are not useful; in any case we plan to replace the line number with an exact location marked in the given paragraph. This message shows that the error concerns the type of the first argument of $\ndres$. (The prefix **global** indicates that $\ndres$ is a global constant; it is not logically necessary here, but may sometimes help reveal how Z/EVES has interpreted its input.)

3.1.2 Domain errors

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 $1 \mathsf{div} 0$, $max\,\mathbb{Z}$, or $\#\,\mathbb{N}$. Second, a definite description (μ-term) is not meaningful if there is not 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.

Different Z authors have adopted different positions on undefinedness in Z:

- Spivey [13] states that any atomic predicate containing an undefined term is *indeterminate*, meaning that it has a truth value (so classical logic applies) but he deliberately does not tell us whether it is true or false.

- The draft ISO standard for Z [5] specifies that an atomic predicate containing an undefined term is false. However, the draft is not definitive on which terms are undefined; the semantics leaves open the possibility that, say, $\mu\, x : \mathbb{Z} \mid x > 0$ may or may not have a value. Furthermore, undefinedness is not inherited in all expression contexts, *e.g.*, if $1 \mathsf{div} 0$ is undefined, the set $\{x : -1 \mathinner{..} 1 \bullet 1 \mathsf{div}\, x\}$ is defined and has the value $\{-1, 1\}$, as any undefined value is simply ignored in a set comprehension.

- Some authors (*e.g.*, Woodcock and Davies [15]) use "strong equality," which is true if both arguments are undefined or if both arguments are defined and equal.

There is also some question about which expressions can be defined even when some subexpression is undefined (*e.g.*, the set comprehension above might be defined even though it contains an expression that may be undefined). So, the proper treatment of undefinedness in Z is not clear. In any case, it is clear that normal mathematical reasoning is not always applicable. For example, the formula $a \in \{a, b\}$ is always true in normal set theory; it can fail in Standard Z if, say, a is defined and b is undefined (as then $\{a, b\}$ is also undefined).

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 $\_ \mathsf{div} \_$ is not necessarily defined; type checking does not show that the second argument is non-zero. When type checking does not guarantee meaningfulness, a predicate called a *domain check* is generated. If the domain check is true, then the original expression is meaningful.

An example may help to clarify how domain checking works. Consider the following schema, intended to describe a simple personnel database:

$$\begin{array}{|l}
\textit{Personnel} \\
\hline
\textit{employees} : \mathbb{P}\,\textit{PERSON} \\
\textit{boss\_of} : \textit{PERSON} \pfun \textit{PERSON} \\
\textit{salary} : \textit{PERSON} \pfun \mathbb{N} \\
\hline
\operatorname{dom} \textit{salary} = \textit{employees} \\
\forall\, e : \textit{employees} \bullet \textit{salary}(e) < \textit{salary}(\textit{boss\_of}\ e) \\
\hline
\end{array}$$

When schema *Personnel* is checked, its status shows that it is syntactically and type correct, but that it has an unproved goal. Selecting "Show Proof" from the pop-up menu shows the proof script for *Personnel\$DomainCheck*, with the following goal predicate:

$$\begin{array}{ll}
& \textit{employees} \in \mathbb{P}\,\textit{PERSON} \\
& \land\ \textit{boss\_of} \in \textit{PERSON} \pfun \textit{PERSON} \\
& \land\ \textit{salary} \in \textit{PERSON} \pfun \mathbb{N} \\
& \land\ \operatorname{dom} \textit{salary} = \textit{employees} \\
& \land\ e \in \textit{employees} \\
\Rightarrow & e \in \operatorname{dom} \textit{salary} \\
& \land\ e \in \operatorname{dom} \textit{boss\_of} \\
& \land\ \textit{boss\_of}\ e \in \operatorname{dom} \textit{salary}
\end{array}$$

There are three conjuncts in the conclusion to prove, corresponding to the three function applications in the last predicate of the schema.

We can simplify this predicate by using a `rewrite` or `reduce` command (available in the "Reduction" menu in the Proof window). The `rewrite` command produces

$$\begin{array}{ll}
& \textit{employees} \in \mathbb{P}\,\textit{PERSON} \\
& \land\ \textit{boss\_of} \in \textit{PERSON} \pfun \textit{PERSON} \\
& \land\ \textit{salary} \in \textit{PERSON} \pfun \mathbb{N} \\
& \land\ \operatorname{dom} \textit{salary} = \textit{employees} \\
& \land\ e \in \textit{employees} \\
\Rightarrow & e \in \operatorname{dom} \textit{boss\_of} \\
& \land\ \textit{boss\_of}\ e \in \operatorname{dom} \textit{salary}
\end{array}$$

One of the conjuncts, $e \in \operatorname{dom} \textit{salary}$, was eliminated, but the other two remain. There are several reasons, in general, why Z/EVES might be unable to prove a predicate; in this case, these predicates are *not* true. In fact, these domain checking conditions show that we have forgotten some constraints in the schema. Both conditions concern the well-definedness of the final condition in the schema, $\forall\, e : \textit{employees} \bullet \textit{salary}(e) < \textit{salary}(\textit{boss\_of}\ e)$. The first, $e \in \operatorname{dom} \textit{boss\_of}$ concerns the expression $\textit{boss\_of}\ e$. In this context, e is known only to be an employee. However, not every employee has a boss (for example, the president would not). So, this quantification should, in fact, range over only those employees having bosses. (A closer inspection of the schema shows that *boss_of* is some partial function, with an unspecified domain. This is probably a mistake; bosses of non-employees should probably not be recorded.)

The second condition, $\textit{boss\_of}\ e \in \operatorname{dom} \textit{salary}$, concerns the expression $\textit{salary}(\textit{boss\_of}\ e)$. Does the boss have a salary? This should be obviously true, but it is not—the specification says nothing about the range of function *boss_of*, so a boss might not be an employee. Again, this was an oversight in the schema definition.

In the light of this analysis, we can revise the definition to eliminate these flaws:

$$\begin{array}{|l}
\textit{Personnel} \\
\hline
\textit{employees} : \mathbb{P}\,\textit{PERSON} \\
\textit{boss\_of} : \textit{PERSON} \nrightarrow \textit{PERSON} \\
\textit{salary} : \textit{PERSON} \nrightarrow \mathbb{N} \\
\hline
\operatorname{dom} \textit{salary} = \textit{employees} \\
\operatorname{dom} \textit{boss\_of} \subseteq \textit{employees} \\
\operatorname{ran} \textit{boss\_of} \subseteq \textit{employees} \\
\forall\, e : \operatorname{dom} \textit{boss\_of} \bullet \textit{salary}(e) < \textit{salary}(\textit{boss\_of}\ e) \\
\hline
\end{array}$$

(This schema also has a nontrivial domain check condition, but this time we can show it to be true.)

3.1.3 Inconsistency

A specification is meant to be a model of some possible system. A specification is *inconsistent* if it has no models. There are two different types of inconsistency, which we call *global inconsistency* and *local inconsistency*. Of the two, global inconsistency is the more serious, as it renders an entire specification unsatisfiable.

Global inconsistency

An entire specification can be inconsistent if some axiom box, generic box, or predicate is too strong. For example, any specification containing the paragraph

$$\begin{array}{|l}
n : \mathbb{Z} \\
\hline
n \neq n
\end{array}$$

has no models, as there is no possible value for n. In this case, the inconsistency is obvious, but in general it can be less so. For example, there is no function f satisfying the following description:

$$\begin{array}{|l}
f : \mathbb{N} \rightarrow \mathbb{N} \\
\hline
\forall\, n, n' : \mathbb{N} \bullet n < n' \bullet f(n) > f(n')
\end{array}$$

An inconsistency can also arise between different paragraphs in a specification, each of which, on its own, is consistent.

Z/EVES does not provide any specific mechanism for detecting such inconsistencies, but has general mechanisms that can be used to detect them. For example, an axiomatic box

$$\begin{array}{|l}
D \\
\hline
P
\end{array}$$

can be checked for inconsistency by preceding it with the conjecture $\exists\, D \bullet P$.

For example, proving $\exists\, \textit{maxLength} : \mathbb{N} \bullet \textit{maxLength} > 80$ demonstrates that the axiomatic box

$$\begin{array}{|l}
\textit{maxLength} : \mathbb{N} \\
\hline
\textit{maxLength} > 80
\end{array}$$

is satisfiable.

Unfortunately, it is not always easy to prove such conjectures; to do so, it is necessary to exhibit suitable values for the declared names. The Z notation does not always provide convenient expressions for this. For example, to show the satisfiability of the definition

$$\begin{array}{|l} exp : \mathbb{Z} \times \mathbb{N} \rightarrow \mathbb{Z} \\ \hline \forall x : \mathbb{Z};\ n : \mathbb{N} \bullet \\ \qquad exp(x, 0) = 1 \\ \qquad \land\ exp(x, n+1) = x * exp(x, n), \end{array}$$

we need to prove

$$\begin{array}{l} \exists\, exp : \mathbb{Z} \times \mathbb{N} \rightarrow \mathbb{Z} \bullet \\ \quad \forall x : \mathbb{Z};\ n : \mathbb{N} \bullet \\ \qquad exp(x, 0) = 1 \\ \qquad \land\ exp(x, n+1) = x * exp(x, n). \end{array}$$

There is no convenient expression to use for *exp* here; the most obvious candidate is defined recursively, and Z provides no facility for recursive definitions. However, the Z/EVES Mathematical Toolkit offers a theorem *generalPrimitiveRecursion* that shows that functions like *exp* exists:

$$\begin{array}{l} \textbf{theorem}\ \text{generalPrimitiveRecursion}\ [Result, Parameter] \\ \quad \forall\, base : Parameter \rightarrow Result;\ step : Result \times \mathbb{N} \times Parameter \rightarrow Result \bullet \\ \qquad \exists f : \mathbb{N} \times Parameter \rightarrow Result \bullet \\ \qquad\quad \forall p : Parameter \bullet \\ \qquad\qquad f(0, p) = base(p) \land (\forall n : \mathbb{N} \bullet f(n+1, p) = step(f(n, p), n, p)) \end{array}$$

Unfortunately (and rather typically), this theorem gives us a function that takes its arguments the wrong way around. However, it is possible, although rather excruciating, to use this theorem to show that *exp* exists. The full proof can be found in the examples directory distributed with Z/EVES.

It is not always possible to show consistency one paragraph at a time; sometimes a group of paragraphs need to be considered as a whole. This is often the case for a constraint paragraph. For example, consider the two paragraphs

$$\begin{array}{|l} count : \mathbb{Z} \end{array}$$

$$count \in 0 \mathinner{\ldotp\ldotp} 99$$

(which in a real specification might be separated by several other paragraphs). These need to be combined before showing consistency. In general, it might be necessary to combine several paragraphs before showing consistency.

Situations involving given types are more complex. Consider, for example, the given type

$$[Name]$$

and the assertion that Tom, Dick, and Harry are three distinct names:

$$\begin{array}{|l} Tom, Dick, Harry : Name \\ \hline \#\{Tom, Dick, Harry\} = 3. \end{array}$$

We cannot state the relevant consistency condition (which would express the proposition that *Name* :, *Tom*, *Dick*, and *Harry* exist). However, given a set S that could be a suitable meaning for *Name* :, we can then express the proposition

$$\exists\, Tom, Dick, Harry : S \bullet \#\{Tom, Dick, Harry\} = 3,$$

which could be proved.

It is also possible for a free type definition to be inconsistent. For example, the definition

$$BigSet ::= make\_set \langle\!\langle \mathbb{P}\, BigSet \rangle\!\rangle$$

has no models; it specifies that *BigSet* is isomorphic to its powerset. This is impossible, as the powerset always has more elements. Restrictions on free type definitions that guarantee consistency are described by Arthan [1], Smith [12], and Spivey [Section 3.10.2, page 84] [13]. Z/EVES does not provide any special facility for generating these conditions.

The reader might be wondering at this point if *anything* is safe! Indeed, some things are: given set declarations, abbreviations, and schema definitions cannot introduce global inconsistency.

No theorem that has been proved in a globally inconsistent specification can be trusted, as its proof is potentially based on impossible assumptions.

Local inconsistency

A schema can have a predicate that is not satisfiable. If such a schema is meant to describe the state of a system, then that system is impossible; if it is meant to describe an operation, then the operation can never be successfully invoked. In either case, the specification is probably in error. Such an error is *local*, however, in that theorems proved in a specification having only local inconsistencies are, in fact, theorems, and specifications of other components of a system may still be meaningful.

It is easy to express the satisfiability of a schema S, using the predicate $\exists\, S \bullet true$. Sometimes a more impressive alternative is available: many specifications give an initialization schema of the form $Init\_S \mathrel{\widehat{=}} [S \mid P]$, where the predicate P further constrains the state. In such a case, showing $\exists\, S \bullet Init\_S$ not only shows that S is satisfiable, it also shows that initial states are possible.

For example, given the (corrected) schema declaration *Personnel* above, and the initialization schema

$$\begin{array}{|l}
\multicolumn{1}{l}{InitialPersonnel} \\
\hline
Personnel \\
\hline
salary = \emptyset, \\
\hline
\end{array}$$

we can show

theorem InitialPersonnelIsPossible
  $\exists\, Personnel \bullet InitialPersonnel$

as follows:

proof
invoke;
prove;
instantiate boss_of == $\emptyset$;
invoke;
prove;
■

3.2 Exploring a Specification

Formal specifications offer opportunities that are completely absent from informal prose descriptions: Formal specifications can be manipulated according to rigorous rules of logic, allowing properties of the specification or of the specified system to be explored. Z/EVES supports several different types of exploration, which are described in the sections below.

3.2.1 Schema expansion

The schema calculus provides a facility for the compact expression of models. Sometimes, though, these representations may be too compact to be easily read. Browsers can help by making it easy to find the definitions of any references in a schema box or schema expression. Z/EVES can help a reader by allowing a schema definition to be expanded and simplified.

The Z/EVES schema expansion facility uses the prover, and treats a schema as a predicate. The `invoke` command can be used to expand one or more references in the predicate, and the other proof commands can be used to simplify the results of the expansion.

For example, consider the bank specification developed in Chapter 2. There, we defined a schema

```
── NewAccount ──────────────────────────────────
│ ΔBank
│ users? : ℙ₁ Client
│ id! : ID
├──────────────
│ id! ∉ accounts
│ ∃ Account′ | OpenAccount • account′ = account ⊕ {id! ↦ θAccount′}
└────────────────────────────────────────────────
```

which refers to schemas $\Delta Bank$, *Account*, and *OpenAccount*. We can begin a proof attempt, with schema *NewAccount* used as a predicate, by adding a paragraph

theorem NewAccountExploration
NewAccount

and checking it. This gives a new goal. Selecting the proof for this paragraph (*e.g.*, by right clicking on the theorem in the specification window and selecting "Show proof"), exposes a proof window with this predicate *NewAccount* as the goal.

Issuing the command `prove by reduce` then results in the new goal predicate

$$
\begin{array}{l}
account \in ID \pfun Account \\
\land accounts = \mathrm{dom}\, account \\
\land users? \in \mathbb{P}\, Client \\
\land id! \in ID \\
\land account' = account \oplus \{(id!, \theta Account[balance := 0, users?/users])\} \\
\land accounts' = \{id!\} \cup \mathrm{dom}\, account \\
\land \neg\ users? = \{\} \\
\land \neg\ id! \in \mathrm{dom}\, account
\end{array}
$$

where all the schema references have been expanded, and some simplification has occurred. In particular, the one-point rule applied to eliminate the existential quantifier, and some redundant conjuncts were eliminated.

After the exploration is complete, the theorem paragraph should be deleted.

3.2.2 Preconditions

The Z style of specifying operations is relational; an operation is specified (in the normal convention) as a relation between initial and final states. This style has a number of advantages, but has the disadvantage of leaving the *precondition* of an operation implicit. For example, an operation

$$\begin{array}{|l}
\textit{Op} \\
\hline
n, n' : \mathbb{N} \\
\hline
n' = n - 1 \\
\hline
\end{array}$$

describes only situations where the initial value of n is non-zero. The conventional interpretation of this is that if Op is invoked with $n = 0$, there are no guarantees about what might happen—the operation might fail catastrophically, might never terminate, or might yield some ordinary result state (which need not even satisfy $n' \in \mathbb{N}$; more generally, such an operation may result in a state that does not satisfy the predicate part of the state schema). Clearly, the circumstances under which this can occur are of some interest.

Z provides a reference to the precondition of an operation schema; for a schema $Op \mathrel{\widehat=} [\Delta S;\ in? : IN;\ out! : OUT]$ the schema reference pre Op is equivalent to $\exists S';\ out! : OUT \bullet Op$, and describes the initial states for which an output and a final state are possible. If the operation is total, it can be executed in any starting state and with any inputs; thus $\forall S;\ in? : IN \bullet \text{pre}\ Op$ should be a theorem. Trying to prove this conjecture can uncover any missing hypotheses. A correct precondition theorem, of the form $\forall S;\ in? : IN \mid P \bullet \text{pre}\ Op$, where P gives the precondition, can be a useful form of documentation of a specification.

For example, a Canadian City Hall marriage registry might contain a record of all married couples, using a relation *wife_of*. This relation is, in fact, a partial injection from men to women—partial because not all men are married; a function because a man may have at most one wife, an injection because a woman may have at most one husband.

$$\begin{array}{|l}
\textit{Registry} \\
\hline
wife\_of : Man ⤔ Woman \\
\hline
\end{array}$$

The *Wedding* operation adds new couple to the registry:

$$\begin{array}{|l}
\textit{Wedding} \\
\hline
\Delta Registry \\
bride? : Woman \\
groom? : Man \\
\hline
wife\_of' = wife\_of \cup \{groom? \mapsto bride?\} \\
\hline
\end{array}$$

We can explore the precondition of this operation by adding a theorem paragraph:

theorem WeddingPrecondition
$\forall Registry;\ bride? : Woman;\ groom? : Man \bullet \text{pre}\ Wedding$

After checking this paragraph, we can work on its proof. The command `prove by reduce` results in the predicate

$$\begin{array}{l}
\quad wife\_of \in Man ⤔ Woman \\
\land\ bride? \in Woman \\
\land\ groom? \in Man \\
\Rightarrow wife\_of \cup \{(groom?, bride?)\} \in Man ⤔ Woman.
\end{array}$$

The various schema definitions have been expanded, the one-point rule was applied, and some simplification was done. The predicate asserts that the updated value of *wife_of* must be a partial injection. We can explore this predicate further. The Z/EVES library has many rewriting rules that are not applied automatically. If a part of the goal predicate is selected and the right mouse button is pressed, any of these disabled rewrite rules that apply to the selected part of the formula can be seen in the "Apply to predicate" or "Apply to expression" submenu. In this case, selecting $\mathit{wife\_of} \cup \{(\mathit{groom?}, \mathit{bride?})\} \in \mathit{Man} \rightarrowtail\!\!\!\!\!\!\!+\;\; \mathit{Woman}$ (*e.g.*, by sweeping the mouse across it or by clicking inside it enough times), there is only one applicable rewrite rule, *cupInPinj*. This theorem describes when a union is a partial injection, and is *disabled*, so that it is not normally applied automatically by a Z/EVES step. (The Toolkit author made this choice because it seemed unwise to introduce the rather complex replacement predicate of this rule without the user's consent.)

Selecting this name from the "Apply to predicate" submenu results in a lengthy predicate we will not show here. The `reduce` command then results in the predicate

$$\begin{array}{l}
\quad \mathit{wife\_of} \in \mathit{Man} \rightarrowtail\!\!\!\!\!\!\!+\;\; \mathit{Woman} \\
\land\ \mathit{bride?} \in \mathit{Woman} \\
\land\ \mathit{groom?} \in \mathit{Man} \\
\Rightarrow (\mathit{groom?} \in \operatorname{dom} \mathit{wife\_of} \Rightarrow \mathit{wife\_of}\ \mathit{groom?} = \mathit{bride?}) \\
\land\ (\mathit{bride?} \in \operatorname{ran} \mathit{wife\_of} \Rightarrow \mathit{wife\_of}^{\sim}\ \mathit{bride?} = \mathit{groom?}),
\end{array}$$

which expresses rather concisely the precondition: if the groom is previously married, then the bride must be his wife, and similarly, if the bride is already married, then the groom must be her husband.

We can go back to the Specification window to revise the theorem to its correct form, making the actual precondition apparent:

$$\begin{array}{l}
\textbf{theorem}\ \text{WeddingPrecondition} \\
\quad \forall \mathit{Registry};\ \mathit{bride?} : \mathit{Woman};\ \mathit{groom?} : \mathit{Man} \\
\qquad \mid \mathit{bride?} \notin \operatorname{ran} \mathit{wife\_of} \land \mathit{groom?} \notin \operatorname{dom} \mathit{wife\_of} \\
\qquad \bullet \operatorname{pre} \mathit{Wedding}
\end{array}$$

This can be rechecked and reproved. The commands used in the exploration are still in the proof script, and can be rerun (by double clicking them in sequence) to complete the proof.

3.2.3 Invariants

Given a sequential system, defined with an initialization schema *Init* and a collection $\mathit{Op}_1, \ldots, \mathit{Op}_n$ of operations, a predicate I is an invariant if $\mathit{Init} \Rightarrow I$, and for every $i \in 1 \mathrel{..} k$, I is preserved by the ith operation (that is, we have $I \land \mathit{Op}_i \Rightarrow I'$). Such invariants can be important for a variety of reasons:

- An implementor can take advantage of an invariant to simplify an implementation. If, for example, $x = y + z$ is an invariant, then one of x, y, or z may be computed as needed rather than stored explicitly in the state. If some sequence is always ordered, efficient searching algorithms are possible.
- An analyst might wish to verify that the operations of a system preserve some important safety properties.

The conditions on an invariant are directly expressible in Z, and so no special mechanisms are needed in Z/EVES to allow their proof.

Here is a trivial example of an invariant proof. Our system has a simple state consisting of a number

$$\begin{array}{|l}
\hline \mathit{State} \\
n : \mathbb{Z}, \\
\hline
\end{array}$$

initializes the number to zero:

Init
State

$n = 0,$

and provides an operation to increment the number

Inc
$\Delta State$

$n' > n.$

Obviously, $n \geq 0$ is an invariant of the system.

theorem SystemInvariant
$(Init \Rightarrow n \geq 0) \wedge (Inc \wedge n \geq 0 \Rightarrow n' \geq 0).$

This can be proved with the `reduce` or `prove by reduce` commands.
It is usually convenient to name the invariant, *e.g.*, in the above example, to define

NonNegative
State

$n \geq 0$

and to express the invariant theorem as

theorem SystemInvariant
$(Init \Rightarrow NonNegative) \wedge (Inc \wedge NonNegative \Rightarrow NonNegative').$

A larger example of an invariant proof has been published [3].

The usual Z style includes all the significant invariant properties in the state schema, and so invariant proofs are not that common. The analyst's job is thus easier; the important properties are obviously maintained. The implementor who wants to simplify the state schema can use a *refinement* proof instead of an invariant proof.

There is an interesting case where it is not possible to add the invariant to the state schema. Given a system with operations $Op1 \ldots\ Opn$, then if pre $Op1 \vee \ldots \vee$ pre Opn is invariant , the specified system never gets stuck—in any reachable state, some operation is always possible.[1] This desired invariant cannot be included in the state schema, because it cannot be expressed until the operations have been defined, and the operations cannot be defined until the state has been defined. In this case, therefore, the kind of proof described in this section is applicable.

3.2.4 Refinement

Z is often used to describe abstract data types. A state schema gives an abstract model of the set of values of the type, an initialization schema describes the possible initial values, and one or more operation schemas describe the operations on the type. In these cases, Z can also be used to express implementations of abstract data types, by giving a second (more concrete) abstract data type and proving that a refinement relation holds between the two types. The rules for proving a refinement are described in detail by Woodcock and Davies [15], who give two different sets of rules (one

[1]This uses an interpretation of operation schemas that considers the precondition of a schema to be an *enabling* condition, in which the operation is allowed to be invoked.

for "forward simulation" and the other for "backward simulation"). Of the two systems, forward simulation is the most commonly used. Given an "abstract" system with state A, initialization schema AI, and operation AO, and given a similar "concrete" system with state C, initialization schema CI, and operation CO, we can demonstrate a forward simulation by exhibiting some relation R between the abstract and concrete states, proving the initialization theorem

$$\forall\, CI \bullet \exists\, AI \bullet R,$$

and proving two theorems for the operation. Firstly,

$$\forall\, R \mid \mathrm{pre}\ AO \bullet \mathrm{pre}\ CO$$

shows that the concrete operation can be invoked whenever the abstract operation can. Secondly,

$$\forall\, R;\ CO \mid \mathrm{pre}\ AO \bullet \exists\, A' \bullet AO \land R'$$

shows that when the abstract operation can be invoked, it can give a result consistent with the concrete operation. (There are several equivalent ways to express these relations; the presentation here differs from Woodcock and Davies in form but not in meaning.)

For example, here is a definition of a trivial system that dispenses numbered "tickets," so that a different number is issued on every call. The state thus has a component that records the numbers that have been issued:

$$\begin{array}{|l}
\hline
\textit{TicketA} \\
\hline
\textit{given} : \mathbb{P}\,\mathbb{Z} \\
\hline
\end{array}$$

Initially, no tickets have been issued.

$$\begin{array}{|l}
\hline
\textit{InitA} \\
\hline
\textit{TicketA} \\
\hline
\textit{given} = \emptyset \\
\hline
\end{array}$$

A new ticket may be requested; the resulting number has not previously been issued, and is recorded in the set of issued tickets:

$$\begin{array}{|l}
\hline
\textit{NewA} \\
\hline
\Delta\,\textit{TicketA} \\
t! : \mathbb{Z} \\
\hline
t! \notin \textit{given} \\
\textit{given}' = \textit{given} \cup \{t!\} \\
\hline
\end{array}$$

That ends the abstract description of the ticket system.

A concrete implementation issues tickets in sequence, beginning with 0, recording the next available number:

$$\begin{array}{|l}
\hline
\textit{TicketC} \\
\hline
\textit{next} : \mathbb{N} \\
\hline
\end{array}$$

Initially, the next available number is 0:

$$\begin{array}{|l}
\textit{InitC} \\
\hline
\textit{TicketC} \\
\hline
\textit{next} = 0
\end{array}$$

When a ticket is requested, the next available number is issued:

$$\begin{array}{|l}
\textit{NewC} \\
\hline
\Delta \textit{TicketC} \\
t! : \mathbb{Z} \\
\hline
t! = \textit{next} \\
\textit{next}' = \textit{next} + 1
\end{array}$$

To show that the concrete system refines the abstract system, we need the abstraction relation. Here, the set of given numbers is the range $0 \mathrel{..} (\textit{next} - 1)$:

$$\begin{array}{|l}
\textit{TicketAbs} \\
\hline
\textit{TicketA} \\
\textit{TicketC} \\
\hline
\textit{given} = 0 \mathrel{..} (\textit{next} - 1)
\end{array}$$

Now we can prove that corresponding operations are refinements. For initialization, we show

theorem initRefinement
$\forall \textit{InitC} \bullet \exists \textit{InitA} \bullet \textit{TicketAbs}$

which can be proved with a `prove by reduce` command. (Obviously, `reduce` is needed here, as the definitions of most of the schemas must be expanded in the proof.)

For the *New* operation, we have two things to show. First, the preconditions are suitably related:

theorem newAbs1
$\forall \textit{TicketAbs} \mid \text{pre } \textit{NewA} \bullet \text{pre } \textit{NewC}$

This can also be proved with a `prove by reduce` command.

Second, we need to show that the effects of the operations are related:

theorem newAbs2
$\forall \textit{NewC}; \ \textit{TicketAbs} \mid \text{pre } \textit{NewA} \bullet (\exists \textit{TicketA}' \bullet \textit{NewA} \land \textit{TicketAbs}')$

This time, the `prove by reduce` command results in

$$\begin{array}{l}
\quad \textit{next} \in \mathbb{Z} \\
\quad \land\ \textit{next}' = 1 + \textit{next} \\
\quad \land\ t! = \textit{next} \\
\quad \land\ \textit{given} = 0 \mathrel{..} {}^{-}1 + \textit{next} \\
\quad \land\ t\_\_0! \in \mathbb{Z} \\
\quad \land\ \textit{given}' = \{t\_\_0!\} \cup (0 \mathrel{..} {}^{-}1 + \textit{next}) \\
\quad \land\ \textit{next} \geq 0 \\
\quad \land\ (0 \leq t\_\_0! \Rightarrow \neg\, t\_\_0! \leq {}^{-}1 + \textit{next}) \\
\Rightarrow \{t!\} \cup (0 \mathrel{..} {}^{-}1 + \textit{next}) = 0 \mathrel{..} \textit{next}
\end{array}$$

This is obvious, but Z/EVES is missing a trivial fact about ranges. We can add the following lemma:

theorem rangeExtension
$\forall a, b : \mathbb{Z} \mid a \leq b \bullet a \mathbin{..} b = (a \mathbin{..} b - 1) \cup \{b\}$

This is easily proved by extensionality; after the commands `apply extensionality` and `prove`, we are left with the predicate

$$\begin{array}{l} \quad a \in \mathbb{Z} \\ \quad \land\ b \in \mathbb{Z} \\ \quad \land\ a \leq b \\ \quad \land\ (y = b \lor (a \leq y \land y \leq {}^{-}1 + b)) \\ \Rightarrow a \leq y \land y \leq b \end{array}$$

which Z/EVES failed to prove automatically because it requires some consideration of cases. Using any of the normalizing reduction commands (*i.e.*, `with normalization simplify`, `with normalization rewrite`, or `with normalization reduce`) which can be found under the "Normal Forms" menu item in the proof commands menu bar, is all that is needed to complete the proof.

With this lemma in place, we can complete the refinement proof. In the specification window, the lemma's theorem paragraph must be moved to appear before theorem *newAbs*2. Then *newAbs*2 can be rechecked and its proof retried. The `prove by reduce` command gives the same result as before; we then use an instance of the lemma and let Z/EVES finish the proof:

proof
prove by reduce;
use rangeExtension[a := 0, b := *next*] ;
prove;
■

The interface offers no handy way to enter the "use" command. It can be added by selecting "New command" from the "Edit" menu and typing it in.

3.2.5 Test cases

Many specifications are expressed in a form that makes them amenable to symbolic evaluation. In particular, if an operation defines the outputs and final state variables as functions of the inputs and initial state variables, then it can be symbolically evaluated. In such cases, Z/EVES can be used to investigate the results of sequences of operations, by defining a "test case" schema as a composition of individual operations.

For example, consider the following trivial bank account system, with state that records an account balance

```
┌─ Account ──────────────────────────────
│ balance : ℤ
└────────────────────────────────────────
```

an initializer that zeros the balance

```
┌─ OpenAccount ──────────────────────────
│ Account
├──────────
│ balance = 0
└────────────────────────────────────────
```

and operations to deposit and withdraw funds:

Deposit

$\Delta Account$
$amount? : \mathbb{N}$

$balance' = balance + amount?$

Withdraw

$\Delta Account$
$amount? : \mathbb{N}$

$balance' = balance - amount?$

We can use sequential composition of schemas to define some test scenarios:

$$\begin{aligned}
Test1 &\mathrel{\widehat{=}} OpenAccount' \fatsemi Deposit[amount? := 100] \\
Test2 &\mathrel{\widehat{=}} Test1 \fatsemi Withdraw[amount? := 50] \\
Test3 &\mathrel{\widehat{=}} OpenAccount' \fatsemi Deposit[amount? := 1000] \\
&\qquad \fatsemi Deposit[amount? := 20] \fatsemi Withdraw[amount? := 500]
\end{aligned}$$

Here we have used a Z/EVES syntax extension (schema replacement) to set the values of the inputs to the operations. This extension is not strictly necessary; for example, $Deposit[amount? := 100]$ could be written in Standard Z as $\exists\, amount? : \mathbb{Z} \mid amount? = 100 \bullet Deposit$ or as $[Deposit \mid amount? = 100] \setminus (amount?)$.

With the test cases defined, we can simply use schema expansion to see what happens. For example, trying $Test3$ is possible by defining a theorem

theorem TryTest3
  $Test3$

In the "proof" of this theorem, the command `prove by reduce` results in the formula

$$balance' = 520.$$

Some caution is advised in the use of test cases; sequential composition of operation schemas does not exactly correspond to the sequential composition of the operations. In particular, it does not account for cases where one operation results in a state not in the precondition of the next. Furthermore, in a schema composition, any two outputs of different operations, if they have the same name, must have the same value.

3.2.6 Test theorems

A useful way of assessing a specification is to prove theorems about it. Such theorems are formal "spot checks" of a specification. For example, consider the ticket generating system defined in Section 3.2.4, which we repeat here:

Ticket

$given : \mathbb{P}\,\mathbb{Z}$

Init

$Ticket$

$given = \emptyset$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{New}} \\
\hline
\Delta \textit{Ticket} \\
t! : \mathbb{Z} \\
\hline
t! \notin \textit{given} \\
\textit{given}' = \textit{given} \cup \{t!\} \\
\hline
\end{array}$$

We can show that two successive invocations of the *New* operation will not return the same ticket value.

$$\textit{TwoTickets} \mathrel{\widehat{=}} \textit{New} \mathbin{;} \textit{New}[t2!/t!]$$

theorem twoTicketsNotSame
$\forall \textit{TwoTickets} \bullet t2! \neq t!$

which is easily proved by the `prove by reduce` command.

Chapter 4

The Z/EVES Prover

The Z/EVES prover combines automatic strategies and detailed user steps, allowing for a collaborative effort in completing a proof. Z/EVES can look after mundane details such as side-conditions on proof steps and trivial subgoals, leaving the user free to focus on the main line of argument of a proof.

Z/EVES offers some powerful automatic commands for proving theorems (*e.g.*, `prove`, or `reduce`). However, these commands will only succeed in proving easy theorems, and then only when the way has been prepared. For example, when a name is defined by an abbreviation definition, axiomatic box, or generic box, it may be necessary for some simple theorems to be stated before the automatic steps can succeed.

In this chapter, we will describe the Z/EVES proving mechanisms in detail; later chapters will offer guidance on how to use these mechanisms effectively. There are two main aspects to effective use of the prover: first, guiding the proof, and second, developing a theory in a way that supports the prover's capabilities.

Even with the best use of the automatic capabilities of the system, however, there will be proofs that are not fully automatic. Indeed, this is likely to be the rule rather than the exception. In these situations, the Z/EVES user needs to know a proof, or at least an outline of a proof; Z/EVES can then be used to check this proof (ensuring that all side-conditions are satisfied) and, perhaps, to fill in any missing details. This is described in Chapter 5.

4.1 Basic Concepts of the Prover

4.1.1 Logic

Z/EVES translates Z specifications into untyped first-order predicate calculus,[1] which is the most familiar and well studied logical system. All logical manipulations act on this translation. Some restrictions are imposed on these operations, to ensure that the result can be translated back into Z.[2] In some sense, then, Z/EVES does not have a logic for Z—certainly we cannot write all of the Z/EVES manipulations in Z. In a practical sense, however, Z/EVES does have such a logic; at the user level, logical manipulations seem to be applied to Z predicates.

Because the underlying logic is untyped, and because of the nature of types in Z, information about the types of expressions is expressed and manipulated logically, using set membership. For

[1] Actually, we translate to a slight variant of classical predicate calculus that provides conditionals and does not distinguish predicates from expressions.

[2] Very occasionally the result cannot be translated back. Usually a subsequent `rewrite` or `prove` command corrects the problem.

example, the fact that i is an integer is expressed by the predicate $i \in \mathbb{Z}$. Z/EVES has no special deductive mechanism for type information; its normal rules for predicates are used.

Types appear in a variety of side-conditions, and sneak into proofs through their appearance in generic actuals. While Z allows the generic actual parameters in a global reference to be omitted (*e.g.*, users write $\mathbb{N} \cup \mathbb{N}_1$ or $\#(1 \mathrel{..} 10)$), the semantics of Z rely on the presence of these actuals, and Z tools infer suitable actuals when users omit them. For example, $\mathbb{N} \cup \mathbb{N}_1$ represents the fully explicit term $(\_ \cup \_)[\mathbb{Z}](\mathbb{N}, \mathbb{N}_1)$. While these generic parameters are normally uninteresting, a sound proof system for Z must take them into account; it is easy to produce a "proof" of *false* in a system that ignores implicit generic actuals.[3]

Z/EVES follows common practice in omitting uninteresting generic actuals in its printed representation of a formula; however, its internal representation maintains them, and they sometimes resurface in side conditions. For example, the Z/EVES Toolkit contains the theorem

$$\begin{array}{l} \textbf{theorem } \text{disabled rule subDef } [X] \\ \quad A \in \mathbb{P}\, X \land B \in \mathbb{P}\, X \\ \quad \Rightarrow (A \subseteq B \Leftrightarrow (\forall x : A \bullet x \in B)) \end{array}$$

Whenever we apply this theorem, *e.g.*, to $\mathbb{N}_1 \subseteq \mathbb{N}$, there is a side condition $\mathbb{N}_1 \in \mathbb{P}\,\mathbb{Z} \land \mathbb{N} \in \mathbb{P}\,\mathbb{Z}$. (The $\subseteq$ in $\mathbb{N}_1 \subseteq \mathbb{N}$ has generic actual $\mathbb{Z}$; in the theorem it has generic actual X. So, X must be instantiated to $\mathbb{Z}$ to apply the theorem.)

Generic actuals can play a surprising role in proofs, sometimes making them more difficult than expected. For example, the predicate $f \in A \rightarrow B \Rightarrow \mathrm{dom}\, f = A$ looks like it should make a useful theorem. There is a problem, though: what are the generic actuals for dom? (Recall that we have $\mathrm{dom}[X, Y] : (X \leftrightarrow Y) \rightarrow \mathbb{P}\, X$.) If we use A and B, then if we try to apply the theorem to, say, $g \in \mathbb{N} \rightarrow \mathbb{N}$, we will conclude $\mathrm{dom}[\mathbb{N}, \mathbb{N}]g = \mathbb{N}$. This is not useful; when we write $\mathrm{dom}\, g$ we mean $\mathrm{dom}[\mathbb{Z}, \mathbb{Z}]g$. So, a generally useful theorem needs to state $A \in \mathbb{P}\, X \land B \in \mathbb{P}\, Y \land f \in A \rightarrow B \Rightarrow \mathrm{dom}[X, Y]f = A$. Several theorems in the Toolkit have a similar form, with seemingly irrelevant extra variables.

The theorem $f \in A \mathrel{+\!\!\!\rightarrow} B \land x \in \mathrm{dom}\, A \Rightarrow f(x) \in B$ has a similar problem with the generic actuals for dom. This problem is compounded by the difficulty in making this fact be applied automatically. In consequence, there are a number of special cases of this theorem in the Toolkit. One of these states $s \in \mathrm{seq}\, X \land x \in \mathrm{dom}\, s \Rightarrow s(i) \in X$, asserting that any element of a sequence lies in the declared element type of that sequence. This really means $\mathrm{dom}[\mathbb{Z}, X]s$, which is inappropriate if X is not a type.

4.1.2 The current theory

A proof is conducted in a particular *theory*, that is, in the context of a particular collection of definitions, axioms, and lemmas. This context is fixed for any given proof in Z/EVES, but is different for different proofs. A theorem appearing as part of a specification has as its context all the definitions and theorems preceding it (thus avoiding circular proofs, where theorem A uses theorem B in its proof and vice-versa). This treatment of the context is simple and easily managed, but has a sometimes inconvenient consequence. If, in the course of a proof, a user discovers the need to introduce a lemma, this lemma needs to be introduced into the context. Thus, this lemma is

[3]For example, after the declarations

$$\begin{array}{l} \mathit{any}[X] == X \\ T ::= t \end{array}$$

if we ignore the missing generic actuals we can develop the predicate $\forall x, y : \mathit{any} \mid x = t \bullet x = y$ in two ways; first, by observing that $x = t$ is always true (if well typed), hence the predicate is equivalent to $\forall x, y : \mathit{any} \bullet x = y$, which is clearly false (*e.g.*, for $x = 0$ and $y = 1$); and second, by observing that if $x = t$ holds, and x and y have the same type, then x and y are both in T, are both equal to t, and so are equal. Therefore, the predicate is true.

inserted into the specification at a point before where it is needed. This insertion will cause any later paragraphs to become not checked.

Each specification starts in the context of the Mathematical Toolkit [10]. There is at present no support for *sections* in the Z/EVES interface.

Theorems can be stated explicitly in a specification by theorem paragraphs. Theorems are also derived from the predicates in an axiom box or generic box. Domain-checking theorems can be generated for paragraphs. Finally, Z/EVES may generate "implicit" theorems when a paragraph is added, as explained in the Z/EVES Reference Manual.

Every Z/EVES theorem, whether explicit or implicit, has a name that can be used to refer to it in a proof step. This name is provided as part of a theorem paragraph, and can be attached to a predicate in an axiom box or generic box by means of a label (which appears in chevrons before the predicate, *e.g.*, $\langle\!\langle \mathit{cNonnegative} \rangle\!\rangle\ c \geq 0$). If no name is given, Z/EVES assigns a name. Implicit theorems always have names assigned by Z/EVES.

4.1.3 Proof style

Z/EVES gives a somewhat unusual style of proof. A proof applies to a *goal* predicate, which is manipulated by proof commands. Proof commands, when they succeed, result in a new goal predicate, which is equivalent to the original.[4] When the goal has been transformed to *true*, the proof is complete.

Such a proof can be presented informally as a chain of equivalences. For example:

$$\begin{array}{l} 2 \in 1 \mathinner{..} 3 \\ \quad \Leftrightarrow \quad \text{definition of } \_ \mathinner{..} \_ \\ 2 \in \{x : \mathbb{Z} \mid 1 \leq x \leq 3\} \\ \quad \Leftrightarrow \quad \text{set theory} \\ \exists\, x : \mathbb{Z} \mid 1 \leq x \leq 3 \bullet x = 2 \\ \quad \Leftrightarrow \quad \text{one-point rule} \\ 2 \in \mathbb{Z} \land 1 \leq 2 \leq 3 \\ \quad \Leftrightarrow \quad \text{arithmetic} \\ \mathit{true} \end{array}$$

In this informal proof, we have given a hint for every step in the chain. In developing a proof with Z/EVES, we use proof commands to transform each predicate to the next.

It is possible to follow this proof plan in Z/EVES, but we seldom want to work at this level of small details. Indeed, the `rewrite`, `reduce`, and `prove` commands each complete this proof in a single step. For illustration, here is the detailed proof in Z/EVES:

We begin the proof by adding a theorem (with the "New Paragraph" function in the "Edit" menu)

theorem SimpleTheorem
$2 \in 1 \mathinner{..} 3$

checking it, then selecting it in the Specification window, right clicking, and selecting "Show proof" from the pop-up menu. The "definition of $\_ \mathinner{..} \_$" referred to above is recorded as theorem *rangeDef* in the Toolkit. As it is a (disabled) rewrite rule, we can "apply" it. The simplest way to do so is to select the expression $1 \mathinner{..} 3$ in the formula (by clicking the left mouse button in it until it is fully highlighted), then right clicking, and looking at the choices in the "Apply to expression" submenu

[4]This is not true in two cases. First, a leading universal quantifier can be removed from a predicate, giving a predicate that is not logically equivalent (but which is a theorem if and only if the original goal is). Second, the `cases` command allows a conjunct or disjunct of a goal to become a new goal, with the other parts temporarily hidden.

of the pop-up menu. Selecting *rangeDef* from this menu causes the prover rewrite with it, replacing $1 \mathrel{..} 3$ by a set comprehensioin term and resulting in the new goal predicate

$$2 \in \textbf{if } 1 \in \mathbb{Z} \land 3 \in \mathbb{Z} \textbf{ then } \{k : \mathbb{Z} \mid 1 \leq k \land k \leq 3\} \textbf{ else } 1 \mathrel{..} 3$$

The two side conditions did not appear in our informal proof. We can eliminate them by simplification (command "Simplify" from the "Reduction" menu):

$$2 \in \{k : \mathbb{Z} \mid 1 \leq k \land k \leq 3\}$$

The next step of our informal proof referred to "set theory," the rules of which are encoded by a large collection or rewrite rules in Z/EVES and the Mathematical Toolkit. The `rewrite` command would apply them here and complete the proof. The `trivial rewrite` command (in the "Reduction" menu) applies just the most elementary rules, with no further simplification:

$$\exists k : \mathbb{Z} \bullet 1 \leq k \land k \leq 3 \land 2 = k$$

The `simplify` command (in the "Reduction" menu) applies the one-point rule when possible:

$$2 \in \mathbb{Z}$$

For technical reasons, the application of the one-point rule often results in the "type" conditions of the quantification remaining unsimplified. A final `simplify` step completes the proof, giving the goal predicate

true

Here is an example of a more realistic proof in Z/EVES, using an explorative style, as described in Chapter 5.

theorem SimpleTheorem2
$\forall k : \mathbb{N} \bullet \mathrm{id}(1 \mathrel{..} k) \in \mathrm{seq}\,\mathbb{N}$

The `prove` command (in the "Reduction" menu) often completes simple proofs automatically, and otherwise usually eliminates the uninteresting parts of a conjecture, so we apply it here. It does not complete the proof here, but results in the predicate

$$\begin{array}{l} \quad k \in \mathbb{Z} \land k \geq 0 \\ \Rightarrow \mathrm{id}(1 \mathrel{..} k) \in \mathrm{seq}\,\mathbb{N} \end{array}$$

We will appeal to the definition of seq. Clicking on the "seq" in the goal predicate highlights it; right clicking exposes a pop-up menu. Selecting "Expand this name" directs the prover to use the definition of seq, giving the new goal predicate

$$\begin{array}{l} \quad k \in \mathbb{Z} \land k \geq 0 \\ \Rightarrow \mathrm{id}(1 \mathrel{..} k) \in \{f : \mathbb{N} \rightarrowtail\!\!\!\!\!\!+ \ \mathbb{N} \mid \exists n : \mathbb{N} \bullet \mathrm{dom}[\mathbb{Z}, \mathbb{N}]f = 1 \mathrel{..} n\} \end{array}$$

A `prove` command gives

$$\begin{array}{l} \quad k \in \mathbb{Z} \land k \geq 0 \\ \Rightarrow (\exists n : \mathbb{N} \bullet k = n \lor k < 1 \land n < 1) \end{array}$$

Z/EVES was not able to find the right instantiation for n. We can specify the correct instance in a proof step. There is no convenient gesture in the interface for instantiating, so it is necessary to select "New Command" from the "Edit" menu, type in the command `instantiate n == k`, select

"Done" from the "File" menu in the editor, then run the command from the proof script window (by double clicking on it). This gives the new goal predicate

$$\begin{aligned} &k \in \mathbb{Z} \land k \geq 0 \\ &\land \neg\, (k \in \mathbb{N} \land (k = k \lor k < 1 \land k < 1) \\ \Rightarrow\ &(\exists\, n : \mathbb{N} \bullet k = n \lor k < 1 \land n < 1) \end{aligned}$$

Instantiation is a bit odd in Z/EVES; instantiating x to e turns a predicate $\exists\, x : S \bullet P(x)$ into $(e \in S \land P(s)) \lor \exists\, x : S \bullet P(x)$. This form is chosen to preserve the equivalence of the final goal with the initial goal. Z/EVES then converted the propositional form $P \Rightarrow Q \lor R$ to $P \land \neg\, Q \Rightarrow R$; there are a number of such propositional rearrangements applied automatically. In this case, the resulting goal really is simple, and Z/EVES can look after the few details. The `prove` or `rewrite` command results in a goal predicate of *true*, and the proof is complete.

There is a strong benefit to the way proof steps preserve the meaning of the goal predicate. Frequently, a conjecture is incorrect, and, with luck, after some number of proof steps it is transformed into an obviously unprovable predicate (*e.g.*, $n > 0$). Any counterexample (*e.g.*, 0) is also a counterexample to the original conjecture. The preservation of meaning also allows proof steps to be used for various kinds of analysis of the specification, as described in Chapter 3.

4.2 Automation

Z/EVES has a number of automatic commands, which can drastically change the goal predicate. It is best to have some understanding of the general effects of these commands, and how they use theorems, before embarking on any difficult proof in Z/EVES.

4.2.1 Traversal and reduction

Three *reduction* commands (`simplify`, `rewrite`, and `reduce`) *traverse* the goal predicate—that is, the predicate is examined in a left-to-right, top-to-bottom reading. Each part (whether a predicate or expression) is "considered" and possibly replaced by a new predicate or expression.

During the traversal, Z/EVES maintains a *context*, which is a set of predicates that can be assumed true. For example, in traversing a predicate of the form $P \land Q \Rightarrow R$,

- first P is considered, and possibly replaced by a new predicate P',
- with P' assumed, Q is considered, and possibly replaced by a new predicate Q',
- with P' and Q' assumed, R is considered, and possibly replaced by a new predicate R',
- the result is, in general, $P' \land Q' \Rightarrow R'$. If one or more of the new predicates is *true* or *false*, it can either be eliminated, or the whole predicate replaced by *true* or *false*.

Predicates are added to the context during traversal in four ways:

- traversal adds it,
- an assumption rule adds it,
- a forward rule adds it, or
- the simplifier finds it to be true (by equality chaining or by Boolean simplification).

The three reduction commands differ in the way they "consider" subformulas. Each of the commands can apply certain theorems or definitions, as explained in the detailed descriptions below. 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 the Z/EVES Reference Manual for details.

By default, Z/EVES retains the propositional structure of a predicate during traversal. In some situations, however, consideration of cases is suggested. For example, in traversing the predicate

$$\begin{array}{l} \quad (P \lor Q) \\ \quad \land (P \Rightarrow R) \\ \quad \land (Q \Rightarrow R) \\ \Rightarrow R \end{array}$$

the prover is unable to infer the conclusion R.

If normalization is used, any time there is a Boolean connective nested inside another, there is a consideration of cases. So, for example, a predicate of the form $(P \lor Q) \land R$ is converted into $\mathbf{if}\ P\ \mathbf{then}\ R\ \mathbf{else}\ (Q \land R)$. When the if-form is traversed, R is considered twice, once with P assumed and once with P denied and Q assumed. In the above example, normalization gives a long formula, where R can be inferred.

Thus, normalization increases the power of the prover. However, since normalization involves repeating parts of a formula, the current goal can become extremely large, so it is disabled by default. The `with normalization` command modifier enables normalization during performance of the specified command. Prover comands with normalization are available in the "Normal forms" menu in the Proof window.

The extra power that normalization gives Z/EVES is especially noticeable when, for example, a hypothesis of the current goal is a disjunction or an implication.

Sometimes the `rearrange` command can be used instead of normalization; rearrangement puts simple parts of a formula ahead of complex ones, and tends to move nested Boolean connectives to the end of the conjunction or disjunction containing them. The `rearrange` command appears in the "Normal forms" menu in the Proof window.

4.2.2 Simplification

In simplification, Z/EVES performs equality and integer reasoning, propositional reasoning and tautology checking, and applies forward rules and assumption rules where possible.

Propositional logic

The rules of propositional logic (*e.g.*, the rules for the basic logical connectives) are encoded in the traversal, normalization, and Boolean simplification mechanisms.

As a predicate is read and its parts considered, the context of assumptions available at each part is determined by the propositional rules. During simplification, if a predicate is considered, and it has already been assumed in the context, it is replaced by *true*. If it has been assumed false in the context, it is replaced by *false*.

If a Boolean combination of predicates is considered, and some parts are *true* or *false*, the parts can be deleted (if they are redundant, *e.g.*, $x = 1 \land \mathit{true}$ can be replaced by $x = 1$) or the Boolean combination can be replaced by *true* or *false* (*e.g.*, $x > 2 \Rightarrow \mathit{true}$ can be replaced by *true*).

These two mechanisms (traversal and Boolean simplifications) allow the simplifier to recognize many tautologies. For example, simplification replaces the predicate $x \in S \Rightarrow x \in S$ in two steps: first, the second occurrence of $x \in S$ is replaced by *true*; second, $x \in S \Rightarrow \mathit{true}$ is replaced by *true*.

When normalization is used (*e.g.*, in the command `with normalization simplify`), the simplifier recognizes all tautologies.

Equality

The simplifier has a decision procedure for the facts about equality (including the substitution of equals for equals). For example, the predicates $x = x$, $x = y \Rightarrow y = x$, and $X = Y \Rightarrow \mathbb{P}\,X = \mathbb{P}\,Y$ are all recognized as *true*.

Arithmetic

Simple arithmetic facts are known to the simplifier. For example, simplification turns $1 < 2$ into *true*, and $2 > 3 + 4$ into *false*. The simplifier also deals with linear arithmetic (involving addition, negation, or multiplication by a literal constant). For example, $x > 1 \Rightarrow x > 0$ is recognized as *true*, as are $\forall\, x, y, z : \mathbb{Z} \bullet x \leq y \leq z \Rightarrow x \leq z$ and $\forall\, x, y : \mathbb{Z} \mid 1 \leq x \wedge 2 + 3 * x \leq y \bullet 5 \leq y$.

More complex terms can appear in place of the variables in these examples, so long as these terms are known to be integer. (A term t is known to be integer if the fact $t \in \mathbb{Z}$ is in the context. As explained below, it is sometimes useful to define assumption rules to add such facts to the context.)

Assumption rules

During traversal, any theorems labelled as "grules" (assumption rules) can be used. As explained in the Z/EVES Reference Manual, an assumption rule has a *trigger* formula. If, during traversal, an instance of a rule's trigger is considered, the rule may be applied. If the rule's predicate is not an implication, it is added to the context. If the predicate is conditional, the hypotheses are *looked up*. If the hypotheses are true, the conclusion is added to the context. (A hypothesis is "looked up" by seeing if a contradiction results from assuming it to be false.)

Assumption rules are frequently used to make the type of a global reference known, or to add inequalities concerning integer global constants. Consider, for example, the paragraph

$$\mid \; \mathit{maxLength} : \mathbb{N}_1$$

Z/EVES automatically adds the assumption rule $\mathit{maxLength\$declaration}$, asserting $\mathit{maxLength} \in \mathbb{N}_1$. This is useful in conjunction with the "weakening" rules in the Toolkit when rewriting, allowing, for example, both $\mathit{maxLength} \in \mathbb{N}$ and $\mathit{maxLength} \in \mathbb{Z}$ to be rewritten to *true*. The simplifier, however, knows nothing about natural numbers, and if we try to prove $\forall\, x : \mathbb{Z} \bullet x + \mathit{maxLength} > x$, we get nowhere.

We can add a grule that adds information about *maxLength*:

theorem grule maxLengthBound
$\quad \mathit{maxLength} \geq 1$

Now, whenever the prover encounters *maxLength*, it will add the fact $\mathit{maxLength} \geq 1$ to the context. With this extra fact, the simplifier can prove simple arithmetic facts, *e.g.*, $\forall\, x : \mathbb{Z} \bullet x + \mathit{maxLength} > x$ or $\mathit{maxLength} + \mathit{maxLength} \geq 2$.

Forward rules

As explained above, a *context* is maintained during traversal. This context is a set of predicates that are assumed true. When a predicate is added to the context, it may trigger a forward rule.

A forward rule is a theorem of the form $P \Rightarrow Q$ that has been labelled with the usage `frule`. (There are other restrictions on the form of such a rule; see the Z/EVES Reference Manual for details.) If an instance of P is added to the context, the forward rule fires, and the corresponding instance of Q is also added.

When a schema is declared, a forward rule is automatically generated. For the schema $S \mathrel{\widehat=} [x, y : \mathbb{Z} \mid x < y]$, this rule is $S \Rightarrow x \in \mathbb{Z} \wedge y \in \mathbb{Z}$. Thus, whenever S is assumed, the types of its components x and y are known. For the schema $T \mathrel{\widehat=} [S, S' \mid \ldots]$, this forward rule is $T \Rightarrow S \land S'$. Thus, whenever T is assumed, so are S and S'; this then triggers the forward rule for S (twice) to add information about x, y, x', and y'.

The syntactic restrictions on forward rules make them somewhat limited in their usefulness. The most common use for forward rules is to forward chain from schema references, to add equalities or inequalities to the context. For example, suppose we have the schema

$$\begin{array}{|l}
\textit{ResourceManager} \\
\hline
\textit{owner} : \textit{Resource} \pfun \textit{User} \\
\textit{in\_use} : \mathbb{P}\,\textit{Resource} \\
\hline
\textit{in\_use} = \operatorname{dom} \textit{owner}. \\
\hline
\end{array}$$

A typical operation schema might be

$$\begin{array}{|l}
\textit{Free} \\
\hline
\Delta \textit{ResourceManager} \\
r? : \textit{Resource} \\
u? : \textit{User} \\
\hline
r? \in \textit{in\_use} \\
\textit{owner}(r?) = u? \\
\textit{owner}' = \{r?\} \ndres \textit{owner}, \\
\hline
\end{array}$$

and its domain checking condition is

$$\forall \Delta \textit{ResourceManager};\ r? : \textit{Resource};\ u? : \textit{User} \bullet r? \in \textit{in\_use} \Rightarrow r? \in \operatorname{dom} \textit{owner}$$

This can, of course, be proved with the `reduce` or `prove by reduce` commands. Sometimes, however, it is undesirable to invoke the *ResourceManager* schema. In this simple example, it does not matter much, but if the schema definition and the domain condition were more complex, the use of reduction would expand the formula too much. In such a case, it might be useful to add the forward rule

theorem frule ResourceManagerFact
$\quad \textit{ResourceManager} \Rightarrow \textit{in\_use} = \operatorname{dom} \textit{owner}.$

With this forward rule in place, whenever a reference to schema *ResourceManager* is assumed, the simplifier will also assume $\textit{in\_use} = \operatorname{dom} \textit{owner}$. Thus, in the domain condition above, by the time $r? \in \operatorname{dom} \textit{owner}$ is considered, $\Delta \textit{ResourceManager}$ and $r? \in \textit{in\_use}$ have been assumed. Z/EVES automatically adds a forward rule for the "above the line" material in a schema, so the assumption $\Delta \textit{ResourceManager}$ triggers the automatic rule and adds *ResourceManager* and $\textit{ResourceManager}'$. Forward rule *ResourceManagerFact* is triggered by both these, so $\textit{in\_use} = \operatorname{dom} \textit{owner}$ and $\textit{in\_use}' = \operatorname{dom} \textit{owner}'$ are added. With all these facts in the context, $r? \in \operatorname{dom} \textit{owner}$ is recognized as *true*. Thus, the domain check can be proved with the `rewrite` or `prove` commands. This might be significantly faster than proofs that expand the definition. More significantly, however, if the proof fails, then the resulting goal will be more compact and easier to read if the schemas are not expanded.

4.2.3 Rewriting

In rewriting, Z/EVES performs simplification, and also applies rewrite rules where possible. A rewrite rule, in general, is a theorem having the form $\textit{Condition} \Rightarrow \textit{Pattern} = \textit{Replacement}$ (or $\textit{Condition} \Rightarrow \textit{Pattern} \Leftrightarrow \textit{Replacement}$), where *Condition* is a predicate, and *Pattern* and

Replacement are both expressions or are both predicates. In the `rewrite` command, when a subformula is considered during traversal, a rewrite rule can be applied if its pattern matches the subformula (*i.e.*, the subformula is an instance of the pattern). In this case, the corresponding instance of the condition is considered. If the condition is recognized as *true*, then the subformula is replaced by the corresponding instance of the replacement. This new subformula is then traversed again (so that further simplifications, rewritings, or reductions are possible).

Rewrite rules are the most easily used automation mechanisms. Unlike assumption rules and forward rules, which have an effect only on the context, rewrite rules cause explicit changes to the goal predicate. Furthermore, rewrite rules can be applied with the `apply` command, which has an immediate effect on the goal predicate. In contrast, forward rules and assumption rules have an indirect effect, by adding facts to the context as a goal is traversed.

The Z/EVES Mathematical Toolkit is full of rewrite rules. For example, it contains the rules

theorem rule applyId $[X]$
$\quad \forall x : X \bullet (\mathrm{id}\, X)(x) = x$

(with condition $x \in X$, pattern $(\mathrm{id}\, X)(x)$, and replacement x) and

theorem rule inRange
$\quad \forall a, b : \mathbb{Z} \bullet x \in a \mathinner{..} b \Leftrightarrow a \leq x \leq b.$

(with condition $a \in \mathbb{Z} \land b \in \mathbb{Z}$, pattern $x \in a \mathinner{..} b$, and replacement $a \leq x \leq b$).

Consider now working on the predicate $0 \leq (\mathrm{id}\{1 \mathinner{..} 5\})(2)$. Rule *applyId* applies to the term $(\mathrm{id}\{1 \mathinner{..} 5\})(2)$. This term can be replaced by 2 if the subgoal $2 \in 1 \mathinner{..} 5$ can be shown. Rule *inRange* applies to this subgoal; this has the conditions $1 \in \mathbb{Z} \land 5 \in \mathbb{Z}$. These are simplified to *true* by arithmetic reasoning, so the original subgoal $(2 \in 1 \mathinner{..} 5)$ is replaced by $1 \leq 2 \leq 5$. Arithmetic reasoning replaces this by *true*. So, the conditions of *applyId* are true, and so $(\mathrm{id}\{1 \mathinner{..} 5\})(2)$ is replaced by 2. The whole goal is now $0 \leq 2$, which simplifies to *true*.

Rewrite rules may be disabled when they are declared. This is useful in cases where some fact is expressed in the form of an equality or equivalence, so it is a syntactically valid rewrite rule, but where it might not be a good idea for Z/EVES to use the rule automatically during every rewriting step. For example, the principle of extensionality is declared disabled:

theorem disabled rule extensionality
$\quad X = Y \Leftrightarrow (\forall x : X \bullet x \in Y) \land (\forall y : Y \bullet y \in X).$

It would be absurd to replace every equality in a formula by a consideration of elements. A disabled rule may be used in the `apply` command, or can be enabled for the duration of a reduction command by the `with enabled` command modifier. For example, in proving $(1 \mathinner{..} 5) \cap (3 \mathinner{..} 7) = 3 \mathinner{..} 5$, the reduction commands make no progress. The command `apply extensionality` gives a consideration of elements of these two sets; the command `prove` then lets Z/EVES finish the proof.

During a proof, the disabled rewrite rules that are applicable to an expression or predicate appearing in the goal predicate can be determined by selecting the expression or formula, right clicking, and examining the “Apply to expresion” or “Apply to predicate” submenu of the pop-up menu. Selecting one of the names in the list causes it to be applied to the selected expression or predicate. If the rule is conditional, the result has the form **if** *Condition* **then** *Replacement* **else** *Original*, so further proof steps (*e.g.*, `rewrite` or `prove`) are usually needed.

4.2.4 Reduction and invocation

In reducing, Z/EVES performs simplification and rewriting, and if a subformula is a schema reference or reference to a name defined as an abbreviation, the subformula will be replaced by the definition of the schema or abbreviation, and the result will again be reduced. As well, conditional rewrite rules apply if their conditions can be shown to be true by reducing.

The `reduce` command is useful in proofs where it is necessary to appeal to the definitions of the terms involved. It is very commonly used in proofs about schemas, for example, or about abbreviations.

In proofs about complex schemas, or in theories with long chains of definitions, reduction may be inadvisable, as the goal will be reduced to primitive terms. It is usually better to introduce enough theorems about the defined notions (as described in Chapter 6) so that the notions can be kept in the goal. This usually gives easier to read goals and more manageable proofs. Definitions can be expanded selectively with the `invoke` command when it is appropriate to do so in a proof. A defined name appearing in a goal predicate (in the Formula part of the Proof window) can be invoked by selecting it, right clicking, and selecting "Expand this name" from the pop-up menu.

4.2.5 Prove loops

The `prove` and `prove by reduce` commands provide the highest level of proof automation. These commands apply a commonly used combination of other commands, until the proof is finished or no more progress is made. The other commands used include `prenex`, to eliminate quantifiers if possible, `rearrange`, to re-order the predicate, putting simpler hypotheses before complex ones, `equality substitution`, to use equality hypotheses, and either `rewrite` (for the `prove` command) or `reduce` (for the `prove by reduce` command).

Chapter 5

Proving Theorems

Z/EVES supports theorem proving in a variety of ways. In this chapter, we will describe two styles of using the prover: explorative and planned.

In an explorative proof, a user begins with some goal to prove and does not necessarily know a proof. Various prover commands can be used to explore the proof space, and, with some luck, the goal can be proved. In this mode, the user hopes to make maximum use of the automatic facilities of the prover.

In a planned proof, a user spends some time working out an informal proof of the goal before starting to work on the proof in Z/EVES. As we describe below, an informal proof can usually be used to guide a Z/EVES proof session.

Which of these two modes is used is a matter of personal choice. Explorative proofs are often successful for simple goals, and we often begin a proof session with a brief period of exploration. If that does not look promising, rather than continuing to explore we stop to think about the goal and sketch an informal proof. This informal proof may have fairly large gaps, however, which become new proof goals. We often deal with these exploratively, and so our Z/EVES session is an intermixing of the two modes of proof.

5.1 Explorative Proving

In explorative proving, the user tries to take maximum advantage of the automatic features of Z/EVES, and issues proof commands without necessarily having a clear idea of what the result will be, or what the main line of proof is.

Generally, the highest level of automation is provided by the `prove` or `prove by reduce` commands. Of the two, `prove` is less powerful (but generally faster), as it does not expand names defined as abbreviations or schema references.

Many simple theorems are proved automatically by `prove`. If a `prove` command gets off track, it is often possible to use some combination of the commands `reduce`, `rewrite`, `simplify`, `rearrange`, `prenex`, and `equality substitute` to proceed in the proof, or to bring in some theorem with a `use` or `apply` command. Except for the last two, these commands are available from menus in the Proof window. Candidates for the `apply` command can be found by selecting a predicate or expression in the goal, right clicking, and expanding the "Apply to expression" or "Apply to predicate" submenu of the pop-up menu; this shows the disabled rewrite rules that apply to the selected formulas.

Here are some examples of explorative proofs of some theorems about disjoint sets. (These theorems will be used in Section 6.3.) Disjointness can be defined by the paragraphs

syntax dj *inrel*

$$
\begin{array}{|l}
\hline [X] \\
\_ \mathrm{dj} \_ : \mathbb{P}\, X \leftrightarrow \mathbb{P}\, X \\
\hline
\langle\!\langle\, \text{disabled rule djDef} \,\rangle\!\rangle \\
\forall A, B : \mathbb{P}\, X \bullet A \mathrel{\mathrm{dj}} B \Leftrightarrow A \cap B = \emptyset \\
\hline
\end{array}
$$

We have attached a label (⟪ disabled rule djDef ⟫
) to the defining predicate. This lets us give the associated theorem a name that can be used in proof commands, allows a *usage* to be specified (determining how the automatic prover commands can use the theorem), and allows the theorem to be disabled. Here, the defining axiom is disabled, as we do not normally want Z/EVES to expand out the concept of disjointness; but is a rule, as rewriting is the most convenient way to use a theorem like this.

theorem rule djNullRight $[X]$
$\quad \forall A : \mathbb{P}\, X \bullet A \mathrel{\mathrm{dj}} \{\}$

In trying this proof, a `prove` or `prove by reduce` command does nothing. Of course, we need to use the definition of relation $\_ \mathrm{dj} \_$, as expressed by theorem *djDef*. As that is a disabled rule, we can `apply` it. There are two ways to do so: compose a new proof command `apply djDef` in the editor and add it to the proof script, or select the expression $A \mathrel{\mathrm{dj}} [X]\{\}$ in the goal predicate, right click, and select *djDef* from the “Apply to expression” submenu of the pop-up menu. In either case, we get the new goal

$$
\begin{array}{l}
\quad A \in \mathbb{P}\, X \\
\Rightarrow (\mathbf{if}\ A \in \mathbb{P}\, X \wedge \{\} \in \mathbb{P}\, X \\
\quad \mathbf{then}\ A \cap [X]\{\} = \emptyset[X] \\
\quad \mathbf{else}\ A \mathrel{\mathrm{dj}} [X]\{\})
\end{array}
$$

Now we will use the `prove` command to clean this up and see what results. In this case, `prove` results in *true* and our proof is complete.

The next theorem shows that disjointness is a symmetric relation:

theorem djIsSymmetric $[X]$
$\quad \forall A, B : \mathbb{P}\, X \bullet A \mathrel{\mathrm{dj}} B \Leftrightarrow B \mathrel{\mathrm{dj}} A$

The symmetry of disjointness similarly needs to use the definition of $\_ \mathrm{dj} \_$, and is otherwise trivial:

proof
apply djDef;
prove;
■

A set is disjoint from a union only if it is disjoint from the two summands:

theorem rule djCupLeft $[X]$
$\quad \forall A, B, C : \mathbb{P}\, X \bullet A \cup B \mathrel{\mathrm{dj}} C \Leftrightarrow A \mathrel{\mathrm{dj}} C \wedge B \mathrel{\mathrm{dj}} C$

We will begin the proof with our usual line of attack:

proof
apply djDef;
prove;
■

This time, the proof is not complete; we are left with the goal

$$\begin{array}{l} A \in \mathbb{P}\,X \wedge B \in \mathbb{P}\,X \wedge C \in \mathbb{P}\,X \\ \Rightarrow (\textbf{if } A \cap [X] C = \{\} \\ \quad\quad \textbf{then } C \cap [x](A \cup [x]B) = \{\} \Leftrightarrow B \cap [X]C = \{\} \\ \quad\quad \textbf{else } \neg\ C \cap [X](A \cup [X]B) = \{\}) \end{array}$$

We can try some algebraic reasoning here, using the fact $C \cap (A \cup B) = (C \cap A) \cup (C \cap B)$ followed by the observation that a union of two sets is empty if and only if the two sets are themselves both empty. We could look for this theorem in the Toolkit documentation, or alternatively, can select the subexpression $C \cap [x](A \cup [x]B) = \{\}$, right click, and look through the "Apply to expression" submenu. The very first choice is a theorem called *distributeCapOverCupRight*. Selecting this causes both occurrences of the selected subexpression to be rewritten. Running a `prove` command to clean up the formula in fact completes the proof: the goal is now *true*. We are in luck; the fact about empty unions was applied automatically by the `prove` command.

Theprevious theorem has a symmetric counterpart (with the union in the left argument of dj rather than the right):

$$\begin{array}{l} \textbf{theorem} \text{ rule djCupRight } [X] \\ \quad \forall A, B, C : \mathbb{P}\,X \bullet A \text{ dj } B \cup C \Leftrightarrow A \text{ dj } B \wedge A \text{ dj } C. \end{array}$$

We could prove this theorem in the same way we proved *djCupLeft* (but probably using a different distribution law). Instead, here we will try a different line, appealing to to symmetry of disjointness and using *djCupLeft*:

proof
use djIsSymmetric$[X][B\ :=\ B\ \cup [X]\ C]$;
prove;
■

This results in the goal

$$\begin{array}{l} \quad\quad A \in \mathbb{P}\,X \\ \quad\quad \wedge\ B \in \mathbb{P}\,X \\ \quad\quad \wedge\ C \in \mathbb{P}\,X \\ \quad\quad \wedge\ (\textbf{if } B \text{ dj } [X]A \textbf{ then } (A \text{ dj } [X]B \cup [X]C \Leftrightarrow C \text{ dj } [X]A) \textbf{ else } \neg\ A \text{ dj } [X]B \cup [X]C) \\ \Rightarrow (\textbf{if } A \text{ dj } [X]B \textbf{ then } (A \text{ dj } [X]B \cup [X]C \Leftrightarrow A \text{ dj } [X]C) \textbf{ else } \neg\ A \text{ dj } [X]B \cup [X]C) \end{array}$$

We need to use the symmetry of disjointness again. This suggests that making *djIsSymmetric* a rewrite rule might not have been a bad idea; here it would have finished the proof automatically.

proof
use djIsSymmetric$[X]$;
use djIsSymmetric$[X][B\ :=\ C]$;
prove;
■

This turned out to be more work than the original line of argument.
A singleton set is disjoint from second set if its sole member is not in that set:

$$\begin{array}{l} \textbf{theorem} \text{ rule djUnitLeft } [X] \\ \quad \forall x : X;\ A : \mathbb{P}\,X \bullet \{x\} \text{ dj } A \Leftrightarrow \neg\ x \in A \end{array}$$

The usual line of argument is successful:

proof
apply djDef;
prove;
■

Subsets of disjoint sets are also disjoint:

theorem djMonotone $[X]$
$\forall A, A', B, B' : \mathbb{P}\, X \mid A \text{ dj } B \land A' \subseteq A \land B' \subseteq B \bullet A' \text{ dj } B'$

We will start the proof of theorem *djMonotone* in the usual way; applying the defining axiom for _ dj _ and running the `prove` command. This gives the new goal

$$\begin{aligned}
&A \in \mathbb{P}\, X \\
&\land A' \in \mathbb{P}\, X \\
&\land B \in \mathbb{P}\, X \\
&\land B' \in \mathbb{P}\, X \\
&\land A' \in \mathbb{P}\, A \\
&\land B' \in \mathbb{P}\, B \\
&\land A \cap [X] B = \{\} \\
\Rightarrow\ &A' \cap [X] B' = \{\}
\end{aligned}$$

Sets can be shown equal by considering their members, and the Toolkit contains at several different versions of the axiom of extensionality. The theorem called *extensionality* is a disabled rewrite rule, and can be applied to this goal to replace the two equalities. This can be done by selecting “New Command” from the “Edit” menu, entering the command `apply extensionality`, selecting “Done” from the “File” menu, then double clicking on this command in the proof script. Alternatively, for each equality, it can be selected with the mouse, then “extensionality” can be selected from the “Apply to predicate” submenu of the pop-up menu exposed by right-clicking. In either case, the result is

$$\begin{aligned}
&A \in \mathbb{P}\, X \\
&\land A' \in \mathbb{P}\, X \\
&\land B \in \mathbb{P}\, X \\
&\land B' \in \mathbb{P}\, X \\
&\land A' \in \mathbb{P}\, A \\
&\land B' \in \mathbb{P}\, B \\
&\land (\forall x : A \cap [X] B \bullet x \in \{\}) \\
&\land (\forall y : \{\} \bullet y \in A \cap [X] B) \\
\Rightarrow\ &(\forall x\_\_0 : A' \cap [X] B' \bullet x\_\_0 \in \{\}) \\
&\land (\forall y\_\_0 : \{\} \bullet y\_\_0 \in A' \cap [X] B')
\end{aligned}$$

As usual in an exploration, we immediately try the `prove` command and hope for the best. In this case, the result is

$$\begin{aligned}
&A \in \mathbb{P}\, X \\
&\land A' \in \mathbb{P}\, X \\
&\land B \in \mathbb{P}\, X \\
&\land B' \in \mathbb{P}\, X \\
&\land A' \in \mathbb{P}\, A \\
&\land B' \in \mathbb{P}\, B \\
&\land x \in A' \\
&\land \neg\, (\exists x\_\_0 : A \cap [X] B \bullet \mathit{true}) \\
\Rightarrow\ &\neg\, x \in B'
\end{aligned}$$

Z/EVES eliminated several quantifiers and simplified somewhat. At this stage, we need to think a bit. We can argue as follows: suppose $x \in B'$ held, given the hypotheses of the goal. Then we would have $x \in A$ and $x \in B$ as well, because $A \subseteq A'$ and $B \subseteq B'$. This contradicts the hypothesis $\neg (\exists x\_\_0 : A \cap [X] B \bullet \mathit{true})$.

We could proceed in several different ways. Here, we start with the hardest step (for Z/EVES): dealing with quantifiers. We argued that x is a "witness" value that contradicts the existentially quantified hypothesis, so willtype in an instantiation command `instantiate x__0 == x`, execute it, then clean up the formula, as usual, by a `prove` command. This results in the goal

$$\begin{array}{l} \quad A \in \mathbb{P}\, X \\ \quad \land A' \in \mathbb{P}\, X \\ \quad \land B \in \mathbb{P}\, X \\ \quad \land B' \in \mathbb{P}\, X \\ \quad \land A' \in \mathbb{P}\, A \\ \quad \land B' \in \mathbb{P}\, B \\ \quad \land x \in A' \\ \quad \land \neg\, x \in B \\ \quad \land \neg\, (\exists x\_\_0 : A \cap [X] B \bullet \mathit{true}) \\ \Rightarrow \neg\, x \in B' \end{array}$$

We still want to argue by contradiction; we can use a `split` command to suppose $x \in B'$, then simplify. Splitting is easy, by selecting the desired predicate (here, $x \in B'$, right clicking, and selecting "Split on this predicate" from the pop-up menu). Following this with a `prove` command finishes the proof.

5.2 Planned Proofs

In a planned proof, some time is spent in developing an informal proof of a theorem before a Z/EVES proof is started. The informal proof is used as a guide to the development of the formal proof.

In the simplest case, an informal proof has the form of a sequence of lines, each of which is a predicate that is either an instance of a previous theorem, or which follows from earlier predicates in the sequence, and where the final predicate is the theorem to be proved. For example, given $Q : X \leftrightarrow Y$; $R : Y \leftrightarrow Z$; and $S : \mathbb{P}\, Y$, we can prove $(Q \rhd S) \fatsemi R = Q \fatsemi (S \lhd R)$ using the following steps:

1. $Q \rhd S = Q \fatsemi \mathrm{id}\, S$, by Toolkit theorem *compIdRight*.
2. Thus, $(Q \rhd S) \fatsemi R = (Q \fatsemi \mathrm{id}\, S) \fatsemi R$, by Leibniz' Law (the substitution of equals for equals).
3. $(Q \fatsemi \mathrm{id}\, S) \fatsemi R = Q \fatsemi (\mathrm{id}\, S \fatsemi R)$, by Toolkit theorem *compAssociates*.
4. $\mathrm{id}\, S \fatsemi R = S \lhd R$, by Toolkit theorem *compIdLeft*.
5. Thus, $Q \fatsemi (\mathrm{id}\, S \fatsemi R) = Q \fatsemi (S \lhd R)$, by Leibniz' Law.
6. $(Q \rhd S) \fatsemi R = Q \fatsemi (S \lhd R)$ by (2), (3), and (5).

Such a proof can be presented to Z/EVES as a series of `use` commands, one for each line of the proof, possibly after some theorems are stated. A theorem is introduced for each step that deduces a predicate from some prior predicates; this theorem asserts that the earlier predicates imply the conclusion. Where a step brings in a prior theorem, the `use` command does so as well. Where a step deduces a new predicate from prior predicates, the `use` command brings in the associated theorem. Finally, a `simplify` or `prove` command completes the proof. Of course, the theorems that are introduced are themselves in need of proof. However, each of these lemmas is usually simpler than the original theorem.

In practice, it is not always necessary to have a theorem for every line of the proof; some steps are so simple that the final `prove` command can make them automatically. This is an advantage that the `prove` command has over the `simplify` command in completing a proof: it is more powerful in filling in gaps in the proof.

For the example above, we need no theorems; all the steps except for the two marked "Leibniz' Law" are references to theorems of the Toolkit, and Z/EVES can do simple equality reasoning. So, the proof can be presented to Z/EVES as follows:

theorem example $[X, Y, Z]$
$\forall Q : X \leftrightarrow Y; \ R : Y \leftrightarrow Z; \ S : \mathbb{P}\, Y \bullet (Q \rhd S) \mathbin{;} R = Q \mathbin{;} (S \lhd R)$

proof
use compIdRight$[X, Y][R := Q]$;
use compAssociates$[X, Y, Y, Z][P := Q, \ Q := \mathrm{id}\ S]$;
use compIdLeft$[Y, Z]$;
prove;
■

Interestingly, a final `simplify` command will not complete the proof, because the instance of theorem *compAssociates* has the side condition $\mathrm{id}\, S \in Y \leftrightarrow Y$, which is not immediately obvious. We could flesh out the proof with a detail of how to derive this, but it is simpler to use the `prove` command and let Z/EVES fill in the details itself.

Our next example is to prove that the domain and range a finite injection are of the same size:

theorem injSizeDomRan $[X, Y]$
$\forall g : X \mathrel{\rightarrowtail\!\!\!\!\!\twoheadrightarrow} Y \bullet \#(\mathrm{dom}\, g) = \#(\mathrm{ran}\, g)$

(Recall that $X \mathrel{\rightarrowtail\!\!\!\!\!\twoheadrightarrow} Y$ is the set of finite injections (that is, finite one-to-one mappings) from X to Y, and $X \mathrel{\twoheadrightarrow} Y$ is the set of finite partial functions from X to Y.)

Before starting an informal proof, it is worth checking the Toolkit to see what is already known. The section on finiteness contains something likely:

theorem axiom finiteFunction $[X, Y]$
$\quad f \in X \mathrel{\twoheadrightarrow} Y$
$\Rightarrow \mathrm{dom}\, f \in \mathbb{F}\, X$
$\quad \land \mathrm{ran}\, f \in \mathbb{F}\, Y$
$\quad \land \#(\mathrm{ran}\, f) \leq \#(\mathrm{dom}\, f)$
$\quad \land \#(\mathrm{dom}\, f) = \#f$

Using this fact, we can argue as follows:

1. Suppose $g \in X \mathrel{\rightarrowtail\!\!\!\!\!\twoheadrightarrow} Y$.
2. Then $g \in X \mathrel{\twoheadrightarrow} Y$, because injections are functions.
3. Therefore, by *finiteFunction* (with f replaced by g), we have $\mathrm{dom}\, g \in \mathbb{F}\, X \land \mathrm{ran}\, g \in \mathbb{F}\, Y \land \#(\mathrm{ran}\, g) \leq \#(\mathrm{dom}\, g) \leq \#g$.
4. We also have $g^{\sim} \in Y \mathrel{\twoheadrightarrow} X$, because the inverse of an injection is a partial function.
5. Therefore $\#(\mathrm{dom}\, g) \leq \#(\mathrm{ran}\, g)$ by *finiteFunction* with X and Y interchanged, and with f replaced by $g^{\sim}$.

6. Since $\#(\mathrm{ran}\, g) \leq \#(\mathrm{dom}\, g)$ (by step 3) and $\#(\mathrm{dom}\, g) \leq \#(\mathrm{ran}\, g)$ (by step 5), we have $\#(\mathrm{dom}\, g) = \#(\mathrm{ran}\, g)$.

There are just three steps in this argument that need further justification:

- $g \in X \rightarrowtail\!\!\!\!\!\!\!\!\;\;\, \twoheadrightarrow Y \Rightarrow g \in X \twoheadrightarrow\!\!\!\!\!\!\!\!\!\;\;\;\;\, \mapsto Y$. Z/EVES can make this inference automatically in a `prove` step. So, we do not need to make a lemma for this line.
- $g \in X \rightarrowtail\!\!\!\!\!\!\!\!\;\;\, \twoheadrightarrow Y \Rightarrow g^{\sim} \in Y \twoheadrightarrow\!\!\!\!\!\!\!\!\!\;\;\;\;\, \mapsto X$, which we will state as a lemma, and
- the arithmetic reasoning in the conclusion, which Z/EVES can complete automatically.

Therefore, we state an auxiliary lemma

theorem step4Lemma $[X, Y]$
$\forall g : X \rightarrowtail\!\!\!\!\!\!\!\!\;\;\, \twoheadrightarrow Y \bullet g^{\sim} \in Y \twoheadrightarrow\!\!\!\!\!\!\!\!\!\;\;\;\;\, \mapsto Y$

Now the main theorem can be proved following the argument outlined above:

theorem injSizeDomRan $[X, Y]$
$\forall g : X \rightarrowtail\!\!\!\!\!\!\!\!\;\;\, \twoheadrightarrow Y \bullet \#(\mathrm{dom}\, g) = \#(\mathrm{ran}\, g)$

proof
use finiteFunction$[X,\ Y][f\ :=\ g]$;
use step4Lemma$[X, Y]$;
use finiteFunction$[Y,\ X][f\ :=\ g\ {}^{\sim}[X,\ Y]]$;
prove;
■

The lemma still needs to be proved, but we will not show that proof here.

A more complex informal proof uses quantifier reasoning. For example, in trying to prove a universally quantified conclusion, it is common to eliminate the quantifier, stating something like "let x be arbitrary." Alternatively, having proved an existentially quantified predicate of the form $\exists x : X \bullet P(x)$, it is common to again "let $x : X \mid P(x)$ be given", and assume $P(x)$. Both these steps can be presented to Z/EVES as a prenex operation, using the `prenex` command. Existentially quantified conclusions are proved by proving some instance; such an instance can be provided in a Z/EVES proof with the `instantiate` command.

For example, we can show $(\exists x : S \bullet x \notin T) \Rightarrow \neg S \subseteq T$ using the following argument:

1. Assume $\exists x : S \bullet x \notin T$.
2. Choose such an x (so assume $x \in S$ and $x \notin T$).
3. $S \subseteq T \Leftrightarrow (\forall e : S \bullet e \in T)$, by theorem *subDef*.
4. Putting $e = x$ shows $S \subseteq T$ is false.

theorem example2
$\forall S, T : \mathbb{P}\mathbb{Z} \bullet (\exists x : S \bullet x \notin T) \Rightarrow \neg S \subseteq T$

The choice of x in step 2 can be presented as a `prenex` command, which results in the goal

$S \in \mathbb{P}\mathbb{Z} \land T \in \mathbb{P}\mathbb{Z} \land x \in S \land x \notin T$
$\Rightarrow \neg S \subseteq T$

We can now bring in the instance of theorem *subDef* needed for step 3, with the command `use subDef[\num][A := S, B := T]`. This gives the goal

$$\begin{array}{l} \quad (S \in \mathbb{P}\mathbb{Z} \land T \in \mathbb{P}\mathbb{Z} \Rightarrow (S \subseteq T \Leftrightarrow (\forall x\_\_0 : S \bullet x\_\_0 \in T))) \\ \quad \land S \in \mathbb{P}\mathbb{Z} \\ \quad \land T \in \mathbb{P}\mathbb{Z} \\ \quad \land x \in S \\ \quad \land x \notin T \\ \Rightarrow \neg S \subseteq T \end{array}$$

Theorem *subDef* actually uses x as its bound variable, instead of the e we used in the informal proof sketch. When the theorem is used, Z/EVES renames the bound variable to avoid conflict with the x aleady appearing in the goal. So, our subsequent instantiation needs to use the new name. The command `instantiate x__0 == x` can be followed by `prove` and the proof is complete. In summary, the proof script is

proof
prenex;
use subDef[$\mathbb{Z}$][A := S, B := T];
instantiate $x\_\_0$ == x;
prove;
■

Our next example goal is to prove

theorem relationInDomCrossRan $[X, Y]$
$\forall R : X \leftrightarrow Y \bullet R \in (\operatorname{dom} R) \leftrightarrow (\operatorname{ran} R)$

Informally, we can argue as follows. Suppose we are given $R : X \leftrightarrow Y$. We must show $R \in (\operatorname{dom} R) \leftrightarrow (\operatorname{ran} R)$. Now

1. By the definition of $\_ \leftrightarrow \_$, it is enough to show $R \in \mathbb{P}((\operatorname{dom} R) \times (\operatorname{ran} R))$.
2. By theorem *inPower*, this is equivalent to $\forall e : R \bullet e \in (\operatorname{dom} R) \times (\operatorname{ran} R)$.
3. So, let $e \in R$ be arbitrary; we need to show $e \in (\operatorname{dom} R) \times (\operatorname{ran} R)$.
4. Since $e \in R$ and $R \in X \leftrightarrow Y$, we have $e \in X \times Y$.
5. Therefore, there exists $x : X$ and $y : Y$ with $e = (x, y)$ (by theorem *inCross*2).
6. $e \in R$ and $e = (x, y)$. So, $x \in \operatorname{dom} R$.
7. Similarly, $y \in \operatorname{ran} R$.
8. Thus, $(x, y) \in (\operatorname{dom} R) \times (\operatorname{ran} R)$, and, since $e = (x, y)$, the proof is complete.

Three steps in this chain (4, 6, and 7) need filling in, and we will make a lemma for each of them:

theorem thm1 $[X, Y]$
$\forall R : X \leftrightarrow Y \bullet e \in R \Rightarrow e \in X \times Y$

theorem thm2 $[X, Y]$
$\forall R : X \leftrightarrow Y \bullet (x, y) \in R \Rightarrow x \in \operatorname{dom} R$

theorem thm3 $[X, Y]$
$\forall R : X \leftrightarrow Y \bullet (x, y) \in R \Rightarrow y \in \operatorname{ran} R$

With these three lemmas in place, we can turn to the main theorem.

theorem relationInDomCrossRan $[X, Y]$
$\forall R : X \leftrightarrow Y \bullet R \in (\operatorname{dom} R) \leftrightarrow (\operatorname{ran} R)$

The first step in our argument used the definition of $\_ \leftrightarrow \_$. There are two occurrences in the goal, so we direct Z/EVES to expand the one we want by the command `invoke \dom[X, Y] R \rel \ran[X, Y] R` which gives the goal

$$R \in X \leftrightarrow Y \Rightarrow R \in \mathbb{P}(\operatorname{dom}[X, Y]R \times \operatorname{ran}[X, Y]R).$$

The second step was to use theorem *inPower*. As this is a disabled rewrite rule, it can be "applied" by selecting the predicate, right clicking, and selection "inPower" from the "Apply to predicate" submenu of the pop-up menu. This results in the goal

$$R \in X \leftrightarrow Y \Rightarrow (\forall e : R \bullet e \in \operatorname{dom}[X, Y]R \times \operatorname{ran}[X, Y]R).$$

We next wrote "let $e \in R$ be arbitrary"; this corresponds to a `prenex` command (available in the "Quantifiers" menu), which results in the goal

$$\begin{array}{l} R \in X \leftrightarrow Y \\ \land\ e \in R \\ \Rightarrow e \in \operatorname{dom}[X, Y]R \times \operatorname{ran}[X, Y]R. \end{array}$$

Step 4 was captured in lemma *thm*1, which we will bring in with the command `use thm1[X, Y]`, which must be added as a new command and typed in to the editor. This results in the goal

$$\begin{array}{l} \quad (R \in X \leftrightarrow Y \land e \in R \Rightarrow e \in X \times Y) \\ \quad \land\ R \in X \leftrightarrow Y \\ \quad \land\ e \in R \\ \Rightarrow e \in \operatorname{dom}[X, Y]R \times \operatorname{ran}[X, Y]R \end{array}$$

Normally, after a `use` command we apply a `prove` command to clean up the goal. If we try that here, the added lemma is eliminated and we are left with the goal we had before the `use` command. So, before cleaning up, we will move on to step 5, where $e \in X \times Y$ is used.[1] This step is to infer $\exists x : X;\ y : Y \bullet e = (x, y)$ from $e \in X \times Y$. This is the gist of theorem *InCross*2 in the Toolkit. We can `apply` the theorem, as it is a rewrite rule, by selecting $X \times Y$ in the goal predicate, right clicking, and selecting "InCross2" from the "Apply to expression" submenu. This results in the goal

$$\begin{array}{l} \quad (R \in X \leftrightarrow Y \land e \in R \Rightarrow (\exists x : X;\ y : Y \bullet e = (x, y))) \\ \quad \land\ R \in X \leftrightarrow Y \\ \quad \land\ e \in R \\ \Rightarrow e \in \operatorname{dom}[X, Y]R \times \operatorname{ran}[X, Y]R \end{array}$$

[1] It seems strange that even though this fact is so obvious to Z/EVES, we need to bring it in. There is, in fact, a good reason to add $e \in X \times Y$; once it is explicitly in the formula, other commands can be applied to it, as seen the the remainder of this proof.

We might also have made a different lemma:

theorem thm1a $[X, Y]$
$\forall R : X \leftrightarrow Y \bullet e \in R \Rightarrow (\exists x : X;\ y : Y \bullet e = (x, y))$

which has a less trivial proof than the original form; which adds a fact that is not immediately obvious to Z/EVES (and so which is not eliminated in a following `prove` step; and which saves a step in the main proof. This is easier to see in hindsight than when first planning the proof!

which can be cleaned up with the `prove` command. The resulting goal is

$$\begin{aligned}&R \in X \leftrightarrow Y\\ &\land x \in X\\ &\land y \in Y\\ &\land e = (x, y)\\ &\land (x, y) \in R\\ \Rightarrow\ & x \in \mathrm{dom}[X, Y]R \land y \in \mathrm{ran}[X, Y]R\end{aligned}$$

Z/EVES has jumped ahead a bit in the argument, applying the argument of step 8, but has left us with the remaining two interesting steps. We bring in the two lemmas defined for these steps, and clean up with a prove command

proof
  *use thm*2$[X,\ Y]$;
  *use thm*3$[X,\ Y]$;
  prove;
■

and the proof is complete.

We are, however, not done. We stated and used three lemmas; now we need to prove them. These proofs are relatively straightforward using a proof exploration, and so are not discussed here.

Theorem *thm*1 can be proved by a `prove` command.

Theorem *thm*2 requires a few detailed steps:

proof
  with enabled (*inDom*) *rewrite*;
  instantiate $y\_0\ ==\ y$;
  prove;
  *use thm*1$[X,\ Y][e\ :=\ (x,\ y)]$;
  prove;
■

Theorem *thm*3 can be proved by using theorem *thm*2 on the inverse relation:

proof
  *use thm*2$[Y, X][R\ :=\ R^{\sim}[X, Y],\ x\ :=\ y,\ y\ :=\ x]$;
  prove;
■

Chapter 6

Theory Development

Many specifications need to introduce new concepts. When some new name is introduced, one or more theorems are added (usually in the axiom of generic box introducing the name, but sometimes as separate theorem or predicate paragraphs). This collection of theorems forms the "theory" of the new name.

In order to be successful in getting proofs through Z/EVES, it is necessary to have a sufficient stock of theorems. Numerous simple facts are needed in even the easiest of proofs, and the power of the reduction commands depends critically on the available theorems. The Toolkit defines a fairly rich set of theorems about its functions, so that rewriting and reduction can automatically prove many simple facts.

When a specification introduces new functions, the automatic commands will be unable to prove anything nontrivial unless some theorems about these functions are added. Even worse, proofs of simple facts involving references to the new functions might fail as well, because some simple subgoal about the new function cannot be discharged.

In this chapter, we will describe some of the approaches we use in developing a theory.

6.1 Objectives

The goal of theory development is threefold:

- *logical adequacy*: The given theorems should determine the meaning of the new name.
- *practical adequacy*: There should be enough "support" theorems about a concept to make it possible to prove typical theorems arising from the analysis of specifications using that concept.
- *automation*: When practical, there should be enough rewrite rules, forward rules, and assumption rules so that trivial facts can be proved automatically.

6.1.1 Logical adequacy

A collection of theorems is *logically adequate* if it describes the new concept exactly. There is no general way to test the adequacy of a set of theorems. It is possible to test if the theorems determine exactly one concept. For example, given a definition

$$f : \mathbb{Z} \rightarrow \mathbb{N}$$

$$\forall x : \mathbb{N} \bullet f(x) = 2 * x$$
$$\forall x : \mathbb{Z} \mid x \leq 0 \bullet f(x) = -x,$$

we can try to prove

$$\exists_1 f : \mathbb{Z} \to \mathbb{N} \bullet (\forall x : \mathbb{N} \bullet f(x) = 2 * x) \land (\forall x : \mathbb{Z} \mid x \leq 0 \bullet f(x) = -x),$$

thus showing that the two constraints in the axiom box determine exactly one possible value for f. However, this still does not show that these constraints determine the *right* function (the one the specifier had in mind). Furthermore, it is sometimes the case that a specifier intentionally leaves some cases unspecified. For example, the size of a hash table may be specified as some arbitrary large prime:

$$\begin{array}{|l} hashTableSize : Prime \\ \hline hashTableSize > 1000 \end{array}$$

In cases like this, the reader needs to apply judgment in determining whether the specification is logically adequate.

6.1.2 Practical adequacy

A theory is *practically adequate* if it provides theorems that are convenient in proofs. A theorem can be logically adequate, and yet express its theorems in a way that is hard to use, or that makes proofs of even simple facts difficult. The practical adequacy of a theory is generally determined by doing proofs; these will expose any weaknesses or awkward areas.

For example, it is possible to define the concept of a partial function as

$$X \to\!\!\!\!\!\!+\; Y == \{ R : X \leftrightarrow Y \mid R^\sim \mathbin{⨾} R \subseteq \mathrm{id}\, Y \}.$$

This definition is certainly logically adequate for the definition of partial functions. However, this definition is not necessarily the most convenient for use in showing that a particular relation is a function. The usual line of argument in showing that a relation Q is a function is to show $\forall x : X;\ y, y' : Y \mid (x, y) \in Q \land (x, y') \in Q \bullet y = y'$. It is possible to get a subgoal of this form from the above definition of partial function, but it is not particularly easy. So, a theory of partial functions consisting of just the above definition is not practically adequate. Either some other definition should be used, or (even better) additional theorems should be given, so that a specifier has a choice of different ways to prove that a relation is a theorem.

6.1.3 Automation

A good collection of rewrite rules, forward rules, and assumption rules for a basic theory makes it easier to prove theorems involving its concepts.

New notions are introduced because they are needed in a specification, and these notions will appear in the theorems generated while analysing the specification. While proving a domain condition, the satisfiability of some schema, or some other fact about a specification, it can be an annoyance and a distraction to spend time working on a trivial fact about the basic concepts used. A well-automated theory allows Z/EVES to prove these trivial facts by itself, freeing the specifier to concentrate on the more interesting and illuminating aspects of the proof.

We certainly do not advise a huge investment in setting up a theory so that all the interesting proofs are fully automatic. This is an ambitious undertaking, and likely not worth the effort. It is worthwhile, however, spending enough time to make the trivial steps automatic.

6.2 Guidelines

Several guidelines can be applied in the development of a theory. We can illustrate some of them using the following definition of the complement of a set with respect to its type:

$$\begin{array}{|l}
\hline [X] \\
\sim\ : \mathbb{P}\,X \rightarrow \mathbb{P}\,X \\
\hline \forall\, S : \mathbb{P}\,X \bullet\ \sim S = X \setminus S \\
\hline
\end{array}$$

1. Use plain facts. There can be a temptation to express properties in a compact form, using various notions from the Toolkit. A theory is more likely to be practically adequate if its facts are stated in a simple and direct form. For example, instead of a theorem asserting disjoint $\langle S,\ \sim S\rangle$, it is probably more useful to assert $x \in\ \sim S \Rightarrow \neg\, x \in S$.

2. Look for rewrite rules. Rewriting is by far the easiest of the prover's mechanisms to understand and control, and many mathematical facts make natural rewrite rules if they are expressed properly. Rewrite rules that help reduce the size and complexity of a goal are particularly valuable, as are rules that apply to predicates that commonly occur as side conditions in a proof.

   A common type of rewrite rule replaces an expression or predicate by something simpler. For example, the fact $\forall\, S : \mathbb{P}\,X \bullet\ \sim (\ \sim S) = S$ is an obvious candidate for a rewrite rule; the replacement expression S is always simpler than the pattern $\sim (\ \sim S)$. Similarly, the fact $\sim [X]\{\} = X$ is suitable as a rule.

3. Not all theorems make good rules. While it is useful to identify theorems that can be rewrite rules, one should be judicious. For example, the Z/EVES mathematical Toolkit contains a number of distribution laws, including $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$. This could possibly be a rewrite rule, but it is not obvious which form of expression is superior. Similarly, the equality $R^{\sim *} = R^{* \sim}$ is a possible rewrite rule, but it is not obvious that one form should be preferred to the other. These facts are disabled rewrite rules in the Toolkit, so they can be applied if desired but are not normally applied.

   A fact might be a poor rewrite rule because it introduces complexity. For example, $p \in X \times Y \Leftrightarrow (\exists\, x : X;\ y : Y \bullet p = (x, y)$ can be added as a rewrite rule, but doing so makes rewriting introduce many quantifiers into a predicate. Again, it is better to state such facts as disabled rules, so they can be applied when, and if, they are appropriate in a proof.

4. Make theorems strong. Suppose we have defined the notion of increasing and decreasing sequences. We could then state a theorem

   theorem rule decreasingReverse
   $\forall\, s : \operatorname{seq} \mathbb{Z} \mid s \in \mathit{increasing} \bullet \mathit{rev}(s) \in \mathit{decreasing}$

   This can be made into a rewrite rule, so that if a predicate $\mathit{rev}(t) \in \mathit{decreasing}$ is encountered for some sequence t, Z/EVES will consider the predicate $t \in \mathit{increasing}$. If this subgoal can be proved, the original predicate will be replaced by *true*. But suppose Z/EVES fails to prove the subgoal in some case where a user expected it to succeed. In this case, it is not so easy to find out what happened.

   A stronger fact can be stated:

   theorem rule strongDecreasingReverse
   $\forall\, s : \operatorname{seq} \mathbb{Z} \bullet \mathit{rev}(s) \in \mathit{decreasing} \Leftrightarrow s \in \mathit{increasing}$

   Now, whenever a predicate $\mathit{rev}(t) \in \mathit{decreasing}$ is encountered, it is replaced with $t \in \mathit{increasing}$ (which might then be further rewritten or simplified).

This stronger rule has another advantage: it can be used to refute predicates of the form $\mathit{rev}(s) \in \mathit{decreasing}$ as well as to prove them. For example, the predicate

$$x \notin \mathit{increasing} \land \mathit{rev}(x) \in \mathit{decreasing}$$

can be reduced to *false* using the strong rule, but is unaffected by the weak rule.

A simple fact about complementation is $x \in \ \sim S \Rightarrow \neg\ x \in S$. This could be stated as a theorem, but the equivalence

$$\forall S : \mathbb{P}\,X;\ x : X \bullet x \in \ \sim S \Leftrightarrow \neg\ x \in S$$

is stronger. In fact, and even stronger form is possible:

$$\forall S : \mathbb{P}\,X \bullet x \in \ \sim S \Leftrightarrow x \in X \land \neg\ x \in S.$$

This last form is useful in those (hopefully rare) cases where non-maximal sets are used as generic actuals, and allows us to show, for example, $\neg\ S[\mathbb{N}]$ given the definition

$$S[X] \mathrel{\widehat=} [A : \mathbb{P}\,X \mid -1 \in \ \sim A]$$

In this case, the generic actual for $\sim$ is $\mathbb{N}$ (rather than the maximal set $\mathbb{Z}$).[1]

Even in those cases where the theorem is not suitable as a rewrite rule, a strong form of the theorem can be more powerful, and thus more useful, than a weaker form.

5. Avoid complex conditions. Generally, it is advisable to ensure that any conditions or a rewrite rule are not more complex than the pattern. In particular, it is probably not a good idea to have a rule that introduces some new concept in its conditions. For example, a rule

   theorem rule largePrimesAreOdd
   $\forall x : \mathit{Prime} \mid x > 2 \bullet x \in \mathit{Odd}$

   is silly; any time a predicate $n \in \mathit{Odd}$ is rewritten, the primality of n will be considered. This is likely to be a complete waste of time.

6. Make rules work together. It is wise to have some overall strategy in mind when writing rewrite rules. For example, a good strategy for complementation might be to push complementation inside other set connectives—that is, to rewrite $\sim(A \cup B)$ to $(\sim A) \cap (\sim B)$ and so on. A collection of such rules expressing defines a kind of "normal form" for set expressions.

   In the absence of a coherent strategy for the rules about complementation, it is easy to run into trouble. For example, if we have the three rules

   theorem rule complementCap $[X]$
   $\forall S, T : \mathbb{P}\,X \bullet \ \sim(S \cap T) = (\sim S) \cup (\sim T)$

   theorem rule diffSetminus $[X]$
   $\forall S, T : \mathbb{P}\,X \bullet S \setminus \sim T = S \cap T$

   theorem rule complementSetminus $[X]$
   $\forall S, T : \mathbb{P}\,X \bullet (\sim S) \cup T = \ \sim(S \setminus T)$

[1] This case is debatable; there is not yet complete agreement among Z authorities about the correct generic actuals to associate with the type of generic formal parameters. The approach taken in Z/EVES is to use generic formal parameters as though they are maximal. Thus, for example, in the definition $n[X] == X \cup X$, the generic actual inferred for $\_ \cup \_$ is simply X.

(where in the third rule we chose the opposite direction of rewriting than that suggested by the "push in" strategy), then there is a possibility of conflict between the rules. For example, if a term of the form $\sim (A \cap B)$ is rewritten, it is transformed by rule *complementCap* to $(\sim A) \cup (\sim B)$. This can be rewritten by rule *complementSetminus* to $\sim (A \setminus \sim B)$. Rule *diffSetminus* then applies to the parenthesised term, resulting in $\sim (A \cap B)$, which is exactly where we started. Such rule loops should be avoided.

7. Work with the Toolkit. The Z/EVES Mathematical Toolkit [10] contains many facts about the concepts introduced in the standard Z Mathematical Toolkit, and most of these facts are expressed as rewrite rules. Any rules added in a theory need to work together with these existing rules.[2] For example, to avoid an ordering conflict with rule *emptyDefinition*, a rule's pattern should contain $\{\}$ instead of $\emptyset$.

8. Add assumption rules (grules) for abbreviation definitions, as directed by the Z/EVES Reference Manual. This is really part of working with the Toolkit, but deserves special emphasis.

6.3 A Theory Development Example

One way to develop a theory is to be guided by the needs of its users, identifying where support is needed from attempts to prove theorems involving the new concepts. We will illustrate this approach using the concept of disjoint sets:

```
[X]
_ dj _ : ℙ X ↔ ℙ X
─────────
∀ A, B : ℙ X • A dj B ⇔ A ∩ B = ∅
```

The logical adequacy of this definition is obvious; the predicate determines the truth of A dj B for every pair A, B in the domain of the disjointness relation.

We will use this concept in the specification of a simple resource management system. This gives us several simple conjectures involving disjointness that are used to analyse the specification.

6.3.1 A resource manager

We will first present the specification of the resource manager, and state a few conjectures. Later sections will show the proofs.

The exact nature of these resources is immaterial; we just use a given set *Resources*.

$[Resource]$

The system records which resources are in use and which are available.

```
Manager
used, free : ℙ Resource
─────────
used dj free
```

Initially, there are no used or free resources.

```
InitManager
Manager
─────────
used = free = ∅
```

[2]The exception is a section that does not have the Toolkit as a parent.

As explained in Section 3.1.3, it is useful to ensure that this initial state is satisfiable:

theorem InitManagerIsOK
$\exists \mathit{Manager} \bullet \mathit{InitManager}$

The *AddResource* operation introduces a new resource, which is initially available.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{AddResource}} \\
\Delta \mathit{Manager} \\
r? : \mathit{Resource} \\
\hline
r? \notin \mathit{free} \cup \mathit{used} \\
\mathit{used}' = \mathit{used} \\
\mathit{free}' = \mathit{free} \cup \{r?\} \\
\hline
\end{array}$$

As explained in Section 3.2.2, it is good to examine the precondition of the operation:

theorem AddResourceIsHonest
$\forall \mathit{Manager};\ r? : \mathit{Resource} \mid r? \notin \mathit{free} \cup \mathit{used} \bullet \mathrm{pre}\ \mathit{AddResource}$

The *Alloc* operation is used to obtain a resource.

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Alloc}} \\
\Delta \mathit{Manager} \\
r! : \mathit{Resource} \\
\hline
r! \in \mathit{free} \\
\mathit{free}' = \mathit{free} \setminus \{r!\} \\
\mathit{used}' = \mathit{used} \cup \{r!\} \\
\hline
\end{array}$$

We will also examine the precondition of this operation:

theorem AllocIsHonest
$\forall \mathit{Manager} \mid \mathit{free} \neq \{\} \bullet \mathrm{pre}\ \mathit{Alloc}$

Obviously, we need more operations to make a realistic specification; however, the three conjectures above are all we need to illustrate the issues in theory development.

6.3.2 A first proof attempt

For our first attempt, we will just roll up our sleeves and start proving. All three theorems depend on the schema definitions, so we begin the proofs with a `prove by reduce` command and see what happens.

For *InitManagerIsOK*, `prove by reduce` results in the goal $\{\}$ dj $\{\}$. This is exactly the sort of trivial fact that Z/EVES should be able to prove automatically. For now, we will forge on, proving this fact by using the definition of $\_$ dj $\_$. When the definition was introduced, Z/EVES adds two theorems, which can be seen in the Theorem window. A theorem called *axiom*\$4 contains the predicate from the defining paragraph.

theorem axiom\$4 $[X]$
$A \in \mathbb{P}\,X \land B \in \mathbb{P}\,X \Rightarrow (A \mathrm{\ dj\ } [X] B \Leftrightarrow A \cap [X] B = \emptyset[X]$

(As we will see, it is possible to attach a chosen name to this theorem, by labelling the predicate.) We can appeal to a suitable instance of this fact, and see where that gets us.

proof
use axiom$4[*Resource*][*A* := {}, *B* := {}];
■

Z/EVES rejects this command.[3] We will leave this unfinished and move on to the next theorem, *AddResourceIsHonest*. The command `prove by reduce` results in a goal

$$\begin{array}{l} \quad \mathit{used} \in \mathbb{P}\, \mathit{Resource} \\ \quad \land\ \mathit{free} \in \mathbb{P}\, \mathit{Resource} \\ \quad \land\ r? \in \mathit{Resource} \\ \quad \land\ \mathit{used} \ \mathrm{dj}\ \mathit{free} \\ \quad \land\ \neg\ r? \in \mathit{free} \\ \quad \land\ \neg\ r? \in \mathit{used} \\ \Rightarrow \mathit{used}\ \mathrm{dj}\ \mathit{free} \cup \{r?\}. \end{array}$$

We can use *axiom*$4 to eliminate the two instances of _ dj _:

proof
use axiom$4[*Resource*][*A* := *used*, *B* := *free*];
use axiom$4[*Resource*][*A* := *used*, *B* := *free* ∪ { *r*? }];
prove;
■

which results in the goal

$$\begin{array}{l} \quad \mathit{used} \in \mathbb{P}\, \mathit{Resource} \\ \quad \land\ \mathit{free} \in \mathbb{P}\, \mathit{Resource} \\ \quad \land\ r? \in \mathit{Resource} \\ \quad \land\ \mathit{used} \ \mathrm{dj}\ \mathit{free} \\ \quad \land\ \neg\ r? \in \mathit{free} \\ \quad \land\ \neg\ r? \in \mathit{used} \\ \quad \land\ (\mathit{used}\ \mathrm{dj}\ \mathit{free} \cup \{r?\} \Leftrightarrow \mathit{used} \cap (\mathit{free} \cup \{r?\}) = \{\}) \\ \quad \land\ \mathit{free} \cap \mathit{used} = \{\} \\ \Rightarrow \mathit{used} \cap (\mathit{free} \cup \{r?\}) = \{\} \end{array}$$

This looks pretty messy, and seem to have little to do with our original problem of the resource manager! We will turn to the third goal, *AllocIsHonest*. Command `prove by reduce` produces the goal

$$\begin{array}{l} \quad \mathit{used} \in \mathbb{P}\, \mathit{Resource} \\ \quad \land\ \mathit{free} \in \mathbb{P}\, \mathit{Resource} \\ \quad \land\ \mathit{used} \ \mathrm{dj}\ \mathit{free} \\ \quad \land\ \neg\ \mathit{free} = \{\} \\ \Rightarrow (\exists\, r! : \mathit{Resource} \bullet r! \in \mathit{free} \land \mathit{used} \cup \{r!\}\ \mathrm{dj}\ \mathit{free} \setminus \{r!\}) \end{array}$$

Since *free* is nonempty, it has a member which could be used for *r*!. The Toolkit contains the theorem

theorem nonEmptySetHasMember
$S = \{\} \lor (\exists\, x : S \bullet \mathit{true})$

[3]There is a slight flaw in the way that the current version of Z/EVES checks `use` commands, so, in this version, it is simply impossible to bring in the needed instance of the axiom.

This theorem lets us bring in the fact $\exists\, x : \mathit{free} \bullet \mathit{true}$; prenexing eliminates the quantifier on x; then $r!$ can be instantiated. Following this, we use the `prove` command to clean up the goal:

proof
use nonEmptySetHasMember$[S := \mathit{free}]$;
prove;
instantiate $r! == x$;
prove;
■

The result of these steps is the new goal

$$\begin{array}{l}
\quad \mathit{used} \in \mathbb{P}\, \mathit{Resource} \\
\quad \land \mathit{free} \in \mathbb{P}\, \mathit{Resource} \\
\quad \land x \in \mathit{free} \\
\quad \land \mathit{used} \mathrel{\mathrm{dj}} \mathit{free} \\
\quad \land \neg\, \mathit{free} = \{\} \\
\quad \land \neg\, \mathit{used} \cup \{x\} \mathrel{\mathrm{dj}} \mathit{free} \setminus \{x\} \\
\Rightarrow (\exists\, r! : \mathit{Resource} \bullet r! \in \mathit{free} \land \mathit{used} \cup \{r!\} \mathrel{\mathrm{dj}} \mathit{free} \setminus \{r!\})
\end{array}$$

in which the two disjointness hypotheses are contradictory. We can eliminate function $\_ \mathrel{\mathrm{dj}} \_$, clean up, and see where we are:

proof
use axiom\$4$[\mathit{Resource}][A := \mathit{used} \cup \{x\},\ B := \mathit{free} \setminus \{x\}]$;
use axiom\$4$[\mathit{Resource}][A := \mathit{used},\ B := \mathit{free}]$;
prove;
■

Now the goal is

$$\begin{array}{l}
\quad \mathit{used} \in \mathbb{P}\, \mathit{Resource} \\
\quad \land \mathit{free} \in \mathbb{P}\, \mathit{Resource} \\
\quad \land x \in \mathit{free} \\
\quad \land \mathit{used} \mathrel{\mathrm{dj}} \mathit{free} \\
\quad \land \neg\, \mathit{free} = \{\} \\
\quad \land \neg\, \mathit{used} \cup \{x\} \mathrel{\mathrm{dj}} \mathit{free} \setminus \{x\} \\
\quad \land \mathit{free} \cap \mathit{used} = \{\} \\
\quad \land \neg\, (\mathit{used} \cup \{x\}) \cap (\mathit{free} \setminus \{x\}) = \{\} \\
\Rightarrow (\exists\, r! : \mathit{Resource} \bullet r! \in \mathit{free} \land \mathit{used} \cup \{r!\} \mathrel{\mathrm{dj}} \mathit{free} \setminus \{r!\})
\end{array}$$

The hypotheses are still contradictory, but further proof steps are required to show it.

As can be seen from these three proof attempts, it is rather awkward to work with disjointness. The following two sections show how the theory can be developed to improve the situation.

6.3.3 Second attempt: elimination

We may be able to simplify matters somewhat by eliminating occurrences of function $\_ \mathrel{\mathrm{dj}} \_$, and replacing them by the definition in terms of intersection. We can do this by labelling the defining predicate as a rewrite rule. This will make the reduction commands replace any predicate $X \mathrel{\mathrm{dj}} Y$ by $X \cap Y = \emptyset$.

$$[X]$$
$$\_\ \mathrm{dj}\ \_ : \mathbb{P}\,X \leftrightarrow \mathbb{P}\,X$$
$$\langle\!\langle\, \text{rule djDef} \,\rangle\!\rangle$$
$$\forall A, B : \mathbb{P}\,X \bullet A \ \mathrm{dj}\ B \Leftrightarrow A \cap B = \emptyset$$

With this revised definition, we can recheck the specification and try the example theorems again. For *InitManagerIsOK*, the command `prove by reduce` completes the proof.

For *AddResourceIsHonest*, the `prove by reduce` command results in the goal

$$\begin{array}{l} \quad \mathit{used} \in \mathbb{P}\,\mathit{Resource} \\ \quad \land\ \mathit{free} \in \mathbb{P}\,\mathit{Resource} \\ \quad \land\ r? \in \mathit{Resource} \\ \quad \land\ \mathit{free} \cap \mathit{used} = \{\} \\ \quad \land\ \neg\ r? \in \mathit{free} \\ \quad \land\ \neg\ r? \in \mathit{used} \\ \Rightarrow \mathit{used} \cap (\mathit{free} \cup \{r?\}) = \{\} \end{array}$$

which we saw in our earlier proof attempt. We can complete the proof using some set theory: rule *distributeCapOverCupRight* transforms the conclusion to $(\mathit{user} \cap \mathit{free}) \cup (\mathit{used} \cap \{r?\}) = \{\})$, which can be proved automatically by rewriting. The complete proof is then

proof
  prove by reduce;
  apply distributeCapOverCupRight;
  prove;
  ∎

The proof has been successful. However, the proof is still not as simple as we might like; an obvious fact like this should be proved immediately.

The proof for *AllocIsHonest* is not much different from the previous attempt; the appeals to *axiom*\$4 are gone, but nothing else has changed.

This approach has not been a big improvement. We could follow two paths. One is to stick with the elimination idea, and prove some additional facts about intersection, so that the above proofs can be completed more easily. The second path, which we explore in the next section, is to abandon the elimination approach and add some useful facts about disjointness.

6.3.4 Third attempt: developing a theory

Eliminating occurrences of $\_\ \mathrm{dj}\ \_$ was not a completely successful approach. There is a further drawback: we defined this notion as a convenient and useful concept for use the specification. So, it is rather annoying to have it eliminated in proofs!

In this attempt, we will not eliminate the disjointness relation from theorems; instead we will prove a number of simple facts about disjointness. We can be guided in our selection of theorems by our previous proof attempts.

We first redefine disjointness, this time making the defining predicate a *disabled* rewrite rule. This lets us rewrite with it when we choose to eliminate disjointness (as will be necessary in several proofs below).

$$[X]$$
$$\_\ \mathrm{dj}\ \_ : \mathbb{P}\,X \leftrightarrow \mathbb{P}\,X$$
$$\langle\!\langle\, \text{disabled rule djDef} \,\rangle\!\rangle$$
$$\forall A, B : \mathbb{P}\,X \bullet A \ \mathrm{dj}\ B \Leftrightarrow A \cap B = \emptyset$$

From the proof of theorem *InitSIsOK*, we know that the fact $\emptyset \text{ dj } \emptyset$ is needed. In fact, *any* set is disjoint from the empty set, so we will prove the more general fact $\forall A : \mathbb{P}\, X \bullet A \text{ dj } \emptyset$. This fact can be a rewrite rule, and it looks like a good one: the replacement (in this case *true*) is far simpler than the pattern $A \text{ dj } \emptyset$.

To make this a useful rewrite rule, one other small detail needs to be observed. The Toolkit rules replace $\emptyset$ by $\{\}$. Thus, it is better to use $\{\}$ in the rule's pattern.

theorem rule djNullRight $[X]$
$\forall A : \mathbb{P}\, X \bullet A \text{ dj } \{\}$

The above rule is fine, but it will not apply to a predicate of the form $\{\} \text{ dj } A$, which is also true. In fact, disjointness is a symmetric relation: if $A \text{ dj } B$ then $B \text{ dj } A$. We can easily state and prove this fact, but it is not so easily automated. The equivalence $A \text{ dj } B \Leftrightarrow B \text{ dj } A$ can be made a rewrite rule. Instead of leading to an infinite sequence of rewrite steps, a rule like this is recognized by Z/EVES and is treated specially. Z/EVES applies it at most once (to a given instance) and determines heuristically which permutation of the arguments is preferred. However, there is no guarantee that $\{\} \text{ dj } A$ would be permuted into $A \text{ dj } \{\}$ so that rule *djNullRight* would apply.

It seems we have no option but to state two versions of every interesting rewrite rule, so that is what we will do.

theorem rule djNullLeft $[X]$
$\forall A : \mathbb{P}\, X \bullet \{\} \text{ dj } A$

Although it does not help us in stating our main theorems here, the symmetry of the disjointness relation is a useful fact.

theorem djIsSymmetric $[X]$
$\forall A, B : \mathbb{P}\, X \bullet A \text{ dj } B \Leftrightarrow B \text{ dj } A$

Our attempt to prove theorem *AddResourceIsHonest* without eliminating the disjointness predicate resulted in the goal

$$\begin{array}{l}
\quad used \in \mathbb{P}\, Resource \\
\quad \land\ free \in \mathbb{P}\, Resource \\
\quad \land\ r? \in Resource \\
\quad \land\ used \text{ dj } free \\
\quad \land\ \neg\ r? \in free \\
\quad \land\ \neg\ r? \in used \\
\Rightarrow used \text{ dj } free \cup \{r?\}
\end{array}$$

Two general facts about disjointness are applicable here. First, a set is disjoint with the union of two others if and only if it is disjoint with each of the two other sets.

theorem rule djCupLeft $[X]$
$\forall A, B, C : \mathbb{P}\, X \bullet A \cup B \text{ dj } C \Leftrightarrow A \text{ dj } C \land B \text{ dj } C$

This rewrite rule also needs a "right hand" variant:

theorem rule djCupRight $[X]$
$\forall A, B, C : \mathbb{P}\, X \bullet A \text{ dj } B \cup C \Leftrightarrow A \text{ dj } B \land A \text{ dj } C.$

Secondly, a set A is disjoint with a unit set $\{x\}$ if and only if x is not a member of A.

theorem rule djUnitLeft $[X]$
$\forall x : X;\ A : \mathbb{P}\, X \bullet \{x\} \text{ dj } A \Leftrightarrow \neg\ x \in A$

We again want a "right hand" variant:

theorem rule djUnitRight $[X]$
$\forall x : X;\ A : \mathbb{P}\, X \bullet A \text{ dj } \{x\} \Leftrightarrow \neg\, x \in A$

Finally, when we tried to prove *AllocIsHonest*, we had the contradictory hypotheses*used* dj *free* and $\neg$ *used* dj *free* $\setminus \{x\}$. That these are contradictory is a consequence of the general fact that subsets of disjoint sets are also disjoint:

theorem djMonotone $[X]$
$\forall A, A', B, B' : \mathbb{P}\, X \mid A \text{ dj } B \wedge A' \subseteq A \wedge B' \subseteq B \bullet A' \text{ dj } B'$

There seems to be no straightforward way to automate the use of this theorem, so we leave it as a plain fact.

With these theorems in hand, we can retry the three proofs. Both *InitManagerIsOK* and *AddResourceIsHonest* are proved by the `prove by reduce` command. For *AllocIsHonest*, more work is required:

proof
prove by reduce;
use nonEmptySetHasMember[S := *free*];
prenex;
instantiate $r!$ == x;
prove;
■

We are left with the goal

$$\begin{array}{l} \quad used \in \mathbb{P}\, Resource \\ \quad \wedge\ free \in \mathbb{P}\, Resource \\ \quad \wedge\ x \in free \\ \quad \wedge\ used \text{ dj } free \\ \quad \wedge\ \neg\, free = \{\} \\ \quad \wedge\ \neg\, used \text{ dj } free \setminus \{x\} \\ \Rightarrow (\exists\, r! : Resource \bullet r! \in free \wedge used \text{ dj } free \setminus \{r!\}) \end{array}$$

Bringing in the appropriate instance of *djMonotone* completes the proof:

proof
use djMonotone[*Resource*][A := *used*, A' := *used*, B := *free*, B' := *free* $\setminus$ $\{x\}$];
prove;
■

These proof scripts seem reasonable; the obvious facts about disjointness were proved automatically, and little of the proof of *AllocIsHonest* had to do with disjointness.

Of course, this single "test drive" of the theory of disjointness does not show that it is finished, and it is quite likely that a more realistic resource manager would require additional properties.

Another way to test the theory is with small examples. In this case, the rules we have defined so far are enough to let rewriting show $\{1, 2, 3\}$ dj $\{4\}$ and $\{1, 2, 3\}$ dj $10 \mathrel{..} 20$ are true, and $\{1, 2, 3\}$ dj $2 \mathrel{..} 20$ is false. So, the theory seems to be reasonably well developed.

There is a danger in developing a theory like this—what if the axioms are incorrect? It is easy to overlook some small detail in stating such theorems, and a missing detail can make a supposed theorem unsound. So, these theorems should all be proved. In fact, we already proved most of these theorems in Section 5.1, and the remainder are similar; we will therefore not show the proofs here.

Bibliography

[1] R. D. Arthan. On free type definitions in Z. In Nicholls [8], pages 40–58.

[2] R. Barden, S. Stepney, and D. Cooper. *Z in Practice*. BCS Practitioner Series. Prentice Hall, 1994.

[3] Dan Craigen, Irwin Meisels, and Mark Saaltink. Analysing Z Specifications with Z/EVES. In J. P. Bowen and M. Hinchey (eds.) *Industrial Strength Formal Methods* Springer Verlag FACIT series, 1999.

[4] I. J. Hayes, editor. *Specification Case Studies*. Prentice Hall International Series in Computer Science, 2nd edition, 1993.

[5] ISO. *Z notation.* Technical report, ISO/IEC JTC1/SC22 N1970, 1995. ISO CD 13568; Committee Draft of the proposed Z Standard.

[6] Jonathan Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.

[7] Irwin Meisels and Mark Saaltink. The Z/EVES Reference Manual. Technical Report TR-99-5493-03d, ORA Canada, September 1997.

[8] J. E. Nicholls, editor. *Z User Workshop, York 1991*, Workshops in Computing. Springer-Verlag, 1992.

[9] B. F. Potter, J. E. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Prentice Hall International Series in Computer Science, 2nd edition, 1996.

[10] Mark Saaltink. *The Z/EVES 2.0 Mathematical Toolkit*. Technical Report TR-99-5493-05b, ORA Canada, October 1999.

[11] Mark Saaltink. *The Z/EVES User's Guide*. Technical Report TR-97-5493-06, ORA Canada, October 1999.

[12] A. Smith. On recursive free types in Z. In Nicholls [8], pages 3–39.

[13] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[14] Mike Spivey. *A Guide to the* `zed` *style option.* December 1990. Available by ftp from host `ftp.comlab.ox.ac.uk` in directory `pub/Zforum` as file `zguide.ps.Z`. Also accessible from `http://www.comlab.ox.ac.uk/archive/z.html`.

[15] J. C. P. Woodcock and J. Davies. *Using Z: Specification, Proof and Refinement*. Prentice Hall International Series in Computer Science, 1996.