The Z Notation:

A Reference Manual

Second Edition

J. M. Spivey

Programming Research Group
University of Oxford

Based on the work of

J. R. Abrial, I. J. Hayes, C. A. R. Hoare,
He Jifeng, C. C. Morgan, J. W. Sanders,
I. H. Sørensen, J. M. Spivey, B. A. Sufrin

This edition first published 1992 by
Prentice Hall International (UK) Ltd

Published 1998 by
J. M. Spivey
Oriel College, Oxford, OX1 4EW, England

Contents

Preface

JACK: You're quite perfect, Miss Fairfax.
GWENDOLEN: Oh! I hope I am not that. It would leave no room for developments, and I intend to develop in many directions.
Oscar Wilde, *The Importance of Being Earnest*

The Z notation for specifying and designing software has evolved over the best part of a decade, and it is now possible to identify a standard set of notations which, although simple, capture the essential features of the method. This is the aim of the reference manual in front of you, and it is written with the everyday needs of readers and writers of Z specifications in mind. It is not a tutorial, for a concise statement of general rules is often given rather than a presentation of illustrative examples; nor is it a formal definition of the notation, for an informal but rigorous style of presentation will be more accessible to Z users, who may not be familiar with the special techniques of formal language definition.

It is perhaps worth recording here the causes which led to even this modest step towards standardization of Z. The first of these is the growing trend towards computer assistance in the writing and manipulation of Z specifications. While the specifier's tools amounted to little more than word-processing facilities, they had enough inherent flexibility to make small differences in notation unimportant. But tools are now being built which depend on syntactic analysis, and to some extent on semantic analysis, of specifications. For these tools – syntax checkers, structure editors, type checkers, and so on – to be useful and reliable, there must be agreement on the grammatical rules of the language they support.

Communication between people is also helped by an agreed common notation, and here I expect the part of this manual devoted to the standard 'mathematical tool-kit' to be especially useful. In this part, I have given a formal definition of each mathematical symbol, together with an informal description and a collection of useful algebraic laws relating the symbol to others.

A third reason for standardization is the need to define a syllabus for training courses in the use of Z. Whilst there is an important difference between learning the Z language and learning to be effective in reading and writing Z specifications, just as learning to program is much more than learning a programming language, I hope that this description of the language will provide a useful check-list of topics to be covered in courses.

Finally, as the use of Z increases, there will be a need for a reference point for contracts and research proposals which call for a specification to be written in Z, and this manual is intended to fill that need also.

In selecting the language features and the mathematical symbols to be included, I have tried to maintain a balance between comprehensiveness and simplicity. On one hand, there is a need to promote common notations for as many important concepts as possible; but on the other hand, there is little point in including notations which are used so rarely that they will be forgotten before they are needed. This observation principally affects the choice of symbols to be included in the 'mathematical tool-kit'.

Because one of the aims is increased stability of Z, I have felt obliged to omit from the account certain aspects of Z which still appear to be tentative. I found it difficult to reconcile the idea of *overloading* – that is, the possibility that two distinct variables in the same scope might have identical names – with the idea that common components are identified when schemas are joined, so overloading is forbidden in the language described. The relative weakness of the Z type system would, in any case, make overloading less useful than it is in other languages.

More importantly, I have also felt unable to include a system of formal inference rules for deriving theorems about specifications. The principles on which such a system might be based are clear enough, at least for the parts of Z which mirror ordinary mathematical notation; but the practical usefulness of inference rules seems to depend crucially on making them interact smoothly, and we have not yet gained enough experience to do this.

How to use this book

Here is a brief summary of the contents of each chapter:

Chapter 1 is an overview of the Z notation and its use in specifying and developing programs. The chapter begins with a simple example of a Z specification; this is followed by examples of the use of the schema calculus to modularize a specification and the use of data refinement to relate specifications and designs.

Chapter 2 explains the concepts behind the Z language, such as *schemas* and *types*. It contains definitions of the terms which are used later to explain the constructs of Z. Although the presentation is informal, it assumes a basic knowledge of naive set theory and predicate calculus.

Chapter 3 contains a description of the Z language itself. It is organized according to the syntactic categories of the language, with separate sections on declarations, predicates, expressions, and so on. Some more advanced features of the language, *generics* and *free types*, are given their own sections at the end of the chapter.

Chapter 4 describes a standard collection of mathematical symbols which are useful in specifying information systems. It is divided into six sections, each dealing with a small mathematical theory such as sets, relations or sequences. The chapter starts with a classified list of the symbols it defines, on pages 86 to 88.

Chapter 5 explains the conventions used in describing sequential programs with Z specifications, including the processes of operation and data refinement, by which abstract specifications can be developed into more concrete designs.

Chapter 6 contains a summary of the syntax of Z. It is here that the fine details of Z syntax are presented, such as the relative binding powers of operators, connectives and quantifiers.

Large parts of Chapters 3 and 4 are organized into 'manual pages' with a fixed layout. Each manual page deals with a single construct or symbol, or a small group of related ones. In Chapter 3, the pages may contain the following items:

Name The constructs defined on the page are listed, and a short descriptive title is given for each of them.

Syntax The syntax rules for each construct are given in Backus–Naur Form (BNF).

Scope rules If variables are introduced by a construct, this item identifies the region of text in the specification where they are visible. If the meaning of a construct depends implicitly on the values of certain variables, these variables are listed.

Type rules The type of each kind of expression is described in terms of the types of its sub-expressions. Restrictions on the types of sub-expressions are stated.

Description The meaning of each construct is explained informally.

Laws Some mathematical properties of the constructs and relationships with other constructs are listed.

In Chapter 4, the format is a little different: each mathematical symbol is defined formally in an item headed '**Definition**', using the Z notation itself. Particular emphasis is laid on the collection of mathematical laws obeyed by the symbols. For brevity, the variables used in these laws are not declared explicitly if their types are clear from the context. An item headed '**Notation**' sometimes explains special-purpose notations designed to make the symbols easier to use.

Several special pages in Chapter 4 consist entirely of laws of a certain kind: for example, the laws which express the monotonicity with respect to $\subseteq$ of various operations on sets and relations are collected on page 104 under the title 'Monotonic operations'.

As well as the usual entries under descriptive terms, the general index at the back of the book contains entries for each syntactic class of the language such as Expression or Paragraph. These entries appear in sans-serif type, and refer to the syntax rules for the class. Each symbol defined as part of the mathematical tool-kit has an entry, either under the symbol itself, if it is a word such as *head*, or under a descriptive name if it is a special symbol such as $\oplus$. These special symbols also appear in the one-page 'Index of symbols'.

The glossary at the back of the book contains concise definitions of the technical terms used in describing Z. Each term defined in the glossary is set in *italic* type the first time it appears in the text.

Acknowledgements

It gives me great pleasure to end this preface by thanking my present and former colleagues for allowing me to contribute to the work of theirs reported in this book; many of the ideas are theirs, and I am happy that their names appear with mine on the title page. I owe a special debt of thanks to Bernard Sufrin, whose *Z Handbook* was the starting point for this manual, and whose constant advice and encouragement have helped me greatly. I should like to thank all those who have pointed out errors and suggested possible improvements, and especially the following, who have helped me with detailed comments on the manuscript: Tim Clement (University of Manchester), Anthony Hall (Praxis Systems), Nigel Haigh (Seer Management), Ian Hayes (University of Queensland), Steve King (University of Oxford), Ruaridh MacDonald (Royal Signals and Radar Establishment), Sebastian Masso (University of Oxford), Dan Simpson (Brighton Polytechnic), Sam Valentine (Logica).

I am grateful to Katharine Whitehorn for permission to quote from her book *Cooking in a Bedsitter*. Chapter 1 is adapted from a paper which first appeared in *Software Engineering Journal* under the title 'An introduction to Z and formal specifications', and is reproduced with the permission of the Institute of Electrical Engineers.

My final thanks go to my wife Petronella, who contributed large helpings of the two most vital ingredients, patience and food, even when I seemed to spend more time with the *TEXbook* than I did with her.

Oriel College, Oxford J. M. S.
September, 1988

Preface to the second edition

This second edition remedies a number of defects. There are several language constructs that I had omitted from the first edition as being of marginal use, but turn out to be far more widely used than I had imagined. The most significant of these is notation for the renaming of schema components, but there are many other smaller changes. I have also made some additions to the library of mathematical notation following suggestions from many people. Obviously, this process of extension could go on for ever, and I have only adopted new notations when they seem to be widely needed and to have a close relationship with the notation that was already there. The purpose of the library is not to be an exhaustive list of concepts that are used in specifications, but to provide a basic vocabulary that readers and writers of Z specifications can have in common. All the substantive changes to the language and library are listed in an appendix.

The new edition has also provided an opportunity to improve the exposition in many small ways, and I am grateful to the many people who have written with suggestions, or with questions that they could not answer from the account of Z contained in the first edition. The biggest change is the introduction of an explicit notation for bindings, the objects that inhabit schema types, and its use in explaining the language constructs that involve schemas. I am grateful to Paul Gardiner for persuading me that an explanation of non-generic schemas could be given in this way.

Both the LaTeX style option that was used to print the Z specifications in the book and a type-checking program that enforces the syntax, scope, and type rules may be obtained from the author. For details, write to Mrs. A. Spivey, 34, Westlands Grove, Stockton Lane, York, YO3 0EF.

Wolfson College, Oxford J. M. S.
September, 2001

You probably cannot afford elaborate equipment, and you certainly have no room for it: but the *right* simple tools will stop you longing for the other, complicated ones.

Katharine Whitehorn, *Cooking in a Bedsitter*

CHAPTER 1

Tutorial Introduction

This chapter is an introduction to some of the features of the Z notation, and to its use in specifying information systems and developing rigorously checked designs. The first part introduces the idea of a formal specification using a simple example: that of a 'birthday book', in which people's birthdays can be recorded, and which is able to issue reminders on the appropriate day. The behaviour of this system for correct input is specified first, then the schema calculus is used to strengthen the specification into one requiring error reports for incorrect input.

The second part of the chapter introduces the idea of data refinement as a means of constructing designs which achieve a formal specification. Refinement is presented through the medium of two examples: the first is a direct implementation of the birthday book from part one, and the second is a simple checkpoint facility, which allows the current state of a database to be saved and later restored. A Pascal-like programming language is used to show the code for some of the operations in the examples.

1.1 What is a formal specification?

Formal specifications use mathematical notation to describe in a precise way the properties which an information system must have, without unduly constraining the way in which these properties are achieved. They describe *what* the system must do without saying *how* it is to be done. This *abstraction* makes formal specifications useful in the process of developing a computer system, because they allow questions about what the system does to be answered confidently, without the need to disentangle the information from a mass of detailed program code, or to speculate about the meaning of phrases in an imprecisely-worded prose description.

A formal specification can serve as a single, reliable reference point for those who investigate the customer's needs, those who implement programs to satisfy those needs, those who test the results, and those who write instruction manuals for the system. Because it is independent of the program code, a formal specification of a system can be completed early in its development. Although it might need to be changed as the design team gains in understanding and the perceived needs of the customer evolve, it can be a valuable means of promoting a common understanding among all those concerned with the system.

One way in which mathematical notation can help to achieve these goals is through the use of *mathematical data types* to model the data in a system. These data types are not oriented towards computer representation, but they obey a rich collection of mathematical laws which make it possible to reason effectively about the way a specified system will behave. We use the notation of *predicate logic* to describe abstractly the effect of each operation of our system, again in a way that enables us to reason about its behaviour.

The other main ingredient in Z is a way of decomposing a specification into small pieces called *schemas*. By splitting the specification into schemas, we can present it piece by piece. Each piece can be linked with a commentary which explains informally the significance of the formal mathematics. In Z, schemas are used to describe both static and dynamic aspects of a system. The static aspects include:

- the states it can occupy;
- the invariant relationships that are maintained as the system moves from state to state.

The dynamic aspects include:

- the operations that are possible;
- the relationship between their inputs and outputs;
- the changes of state that happen.

Later, we shall see how the schema language allows different facets of a system to be described separately, then related and combined. For example, the operation of a system when it receives valid input may be described first, then the description may be extended to show how errors in the input are handled. Or the evolution of a single process in a complete system may be described in isolation, then related to the evolution of the system as a whole.

We shall also see how schemas can be used to describe a transformation from one view of a system to another, and so explain why an abstract specification is correctly implemented by another containing more details of a concrete design. By constructing a sequence of specifications, each containing more details than the last, we can eventually arrive at a program with confidence that it satisfies the specification.

1.2 The birthday book

The best way to see how these ideas work out is to look at a small example. For a first example, it is important to choose something simple, and I have chosen a system so simple that it is usually implemented with a notebook and pencil rather than a computer. It is a system which records people's birthdays, and is able to issue a reminder when the day comes round.

In our account of the system, we shall need to deal with people's names and with dates. For present purposes, it will not matter what form these names and dates take, so we introduce the set of all names and the set of all dates as *basic types* of the specification:

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

This allows us to name the sets without saying what kind of objects they contain. The first aspect of the system to describe is its *state space*, and we do this with a schema:

$$
\begin{array}{|l}
\hline
\mathit{BirthdayBook} \\
\mathit{known} : \mathbb{P}\, \mathit{NAME} \\
\mathit{birthday} : \mathit{NAME} \mathrel{+\!\!\!\!\rightarrow} \mathit{DATE} \\
\hline
\mathit{known} = \operatorname{dom} \mathit{birthday} \\
\hline
\end{array}
$$

Like most schemas, this consists of a part above the central dividing line, in which some variables are declared, and a part below the line which gives a relationship between the values of the variables. In this case we are describing the state space of a system, and the two variables represent important *observations* which we can make of the state:

- *known* is the set of names with birthdays recorded;
- *birthday* is a function which, when applied to certain names, gives the birthdays associated with them.

The part of the schema below the line gives a relationship which is true in every state of the system and is maintained by every operation on it: in this case, it says that the set *known* is the same as the domain of the function *birthday* – the set of names to which it can be validly applied. This relationship is an *invariant* of the system.

In this example, the invariant allows the value of the variable *known* to be derived from the value of *birthday*: *known* is a *derived* component of the state, and it would be possible to specify the system without mentioning *known* at all. However, giving names to important concepts helps to make specifications more readable; because we are describing an abstract view of the state space of the birthday book, we can do this without making a commitment to represent *known* explicitly in an implementation.

One possible state of the system has three people in the set *known*, with their birthdays recorded by the function *birthday*:

$$known = \{\, \mathrm{John}, \mathrm{Mike}, \mathrm{Susan} \,\}$$

$$\begin{aligned} birthday = \{\, & \mathrm{John} \mapsto \text{25–Mar}, \\ & \mathrm{Mike} \mapsto \text{20–Dec}, \\ & \mathrm{Susan} \mapsto \text{20–Dec} \,\}. \end{aligned}$$

The invariant is satisfied, because *birthday* records a date for exactly the three names in *known*.

Notice that in this description of the state space of the system, we have not been forced to place a limit on the number of birthdays recorded in the birthday book, nor to say that the entries will be stored in a particular order. We have also avoided making a premature decision about the format of names and dates. On the other hand, we have concisely captured the information that each person can have only one birthday, because the variable *birthday* is a function, and that two people can share the same birthday as in our example.

So much for the state space; we can now start on some *operations* on the system. The first of these is to add a new birthday, and we describe it with a schema:

```
── AddBirthday ─────────────────────────────
│ ΔBirthdayBook
│ name? : NAME
│ date? : DATE
├──────────
│ name? ∉ known
│
│ birthday′ = birthday ∪ {name? ↦ date?}
└───────────────────────────────────────────
```

The declaration $\Delta BirthdayBook$ alerts us to the fact that the schema is describing a *state change*: it introduces four variables *known*, *birthday*, $known'$ and $birthday'$. The first two are observations of the state before the change, and the last two are observations of the state after the change. Each pair of variables is implicitly constrained to satisfy the invariant, so it must hold both before and after the operation. Next come the declarations of the two inputs to the operation. By convention, the names of inputs end in a question mark.

The part of the schema below the line first of all gives a *pre-condition* for the success of the operation: the name to be added must not already be one of those known to the system. This is reasonable, since each person can only have one birthday. This specification does not say what happens if the pre-condition is not satisfied: we shall see later how to extend the specification to say that an error message is to be produced. If the pre-condition is satisfied, however, the second line says that the birthday function is extended to map the new name to the given date.

We expect that the set of names known to the system will be augmented with the new name:

$$known' = known \cup \{name?\}.$$

In fact we can *prove* this from the specification of *AddBirthday*, using the invariants on the state before and after the operation:

$$\begin{aligned}
known' &= \operatorname{dom} birthday' && \text{[invariant after]}\\
&= \operatorname{dom}(birthday \cup \{name? \mapsto date?\}) && \text{[spec. of } AddBirthday\text{]}\\
&= \operatorname{dom} birthday \cup \operatorname{dom}\{name? \mapsto date?\} && \text{[fact about `dom']}\\
&= \operatorname{dom} birthday \cup \{name?\} && \text{[fact about `dom']}\\
&= known \cup \{name?\}. && \text{[invariant before]}
\end{aligned}$$

Stating and proving properties like this one is a good way of making sure the specification is accurate; reasoning from the specification allows us to explore the behaviour of the system without going to the trouble and expense of implementing it. The two facts about 'dom' used in this proof are examples of the laws obeyed by mathematical data types:

$$\operatorname{dom}(f \cup g) = (\operatorname{dom} f) \cup (\operatorname{dom} g)$$
$$\operatorname{dom}\{a \mapsto b\} = \{a\}.$$

Chapter 4 contains many laws like these.

Another operation might be to find the birthday of a person known to the system. Again we describe the operation with a schema:

$$\begin{array}{|l}
\hline
\textit{FindBirthday}\\
\Xi BirthdayBook\\
name? : NAME\\
date! : DATE\\
\hline
name? \in known\\
date! = birthday(name?)\\
\hline
\end{array}$$

This schema illustrates two new notations. The declaration $\Xi BirthdayBook$ indicates that this is an operation in which the state does not change: the values $known'$ and $birthday'$ of the observations after the operation are equal to their values *known* and *birthday* beforehand. Including $\Xi BirthdayBook$ above the line has the same effect as including $\Delta BirthdayBook$ above the line and the two equations

$$known' = known$$
$$birthday' = birthday$$

below it. The other notation is the use of a name ending in an exclamation mark for an output: the *FindBirthday* operation takes a name as input and yields the corresponding birthday as output. The pre-condition for success of the operation is that *name*? is one of the names known to the system; if this is so, the output *date*! is the value of the birthday function at argument *name*?.

The most useful operation on the system is the one to find which people have birthdays on a given date. The operation has one input *today*?, and one output, *cards*!, which is a *set* of names: there may be zero, one, or more people with birthdays on a particular day, to whom birthday cards should be sent.

```
Remind
  ΞBirthdayBook
  today? : DATE
  cards! : ℙ NAME
  ─────────────
  cards! = { n : known | birthday(n) = today? }
```

Again the Ξ convention is used to indicate that the state does not change. This time there is no pre-condition. The output *cards*! is specified to be equal to the set of all values n drawn from the set *known* such that the value of the birthday function at n is *today*?. In general, y is a member of the set $\{ x : S \mid \ldots x \ldots \}$ exactly if y is a member of S and the condition $\ldots y \ldots$, obtained by replacing x with y, is satisfied:

$$y \in \{ x : S \mid \ldots x \ldots \} \Leftrightarrow y \in S \land (\ldots y \ldots).$$

So, in our case,

$$m \in \{ n : known \mid birthday(n) = today? \} \\ \quad \Leftrightarrow m \in known \land birthday(m) = today?\,.$$

A name m is in the output set *cards*! exactly if it is known to the system and the birthday recorded for it is *today*?.

To finish the specification, we must say what state the system is in when it is first started. This is the *initial state* of the system, and it also is specified by a schema:

```
InitBirthdayBook
  BirthdayBook
  ─────────────
  known = ∅
```

This schema describes a birthday book in which the set *known* is empty: in consequence, the function *birthday* is empty too.

What have we achieved in this specification? We have described in the same mathematical framework both the state space of our birthday-book system and the operations which can be performed on it. The data objects which appear in the system were described in terms of mathematical data types such as sets and functions. The description of the state space included an invariant relationship between the parts of the state – information which would not be part of a program implementing the system, but which is vital to understanding it.

The effects of the operations are described in terms of the relationship which must hold between the input and the output, rather than by giving a recipe to be followed. This is particularly striking in the case of the *Remind* operation, where we simply documented the conditions under which a name should appear in the output. An implementation would probably have to examine the known names one at a time, printing the ones with today's date as it found them, but this complexity has been avoided in the specification. The implementor is free to use this technique, or any other one, as he or she chooses.

1.3 Strengthening the specification

A correct implementation of our specification will faithfully record birthdays and display them, so long as there are no mistakes in the input. But the specification has a serious flaw: as soon as the user tries to add a birthday for someone already known to the system, or tries to find the birthday of someone not known, it says nothing about what happens next. The action of the system may be perfectly reasonable: it may simply ignore the incorrect input. On the other hand, the system may break down: it may start to display rubbish, or perhaps worst of all, it may appear to operate normally for several months, until one day it simply forgets the birthday of a rich and elderly relation.

Does this mean that we should scrap the specification and begin a new one? That would be a shame, because the specification we have describes clearly and concisely the behaviour for correct input, and modifying it to describe the handling of incorrect input could only make it obscure. Luckily there is a better solution: we can describe, separately from the first specification, the errors which might be detected and the desired responses to them, then use the operations of the Z *schema calculus* to combine the two descriptions into a stronger specification.

We shall add an extra output $result!$ to each operation on the system. When an operation is successful, this output will take the value ok, but it may take the other values $already\_known$ and not_known when an error is detected. The following *free type definition* defines $REPORT$ to be a set containing exactly these three values:

$$REPORT ::= ok \mid already\_known \mid not\_known.$$

We can define a schema *Success* which just specifies that the result should be ok, without saying how the state changes:

```
┌─ Success ───────────────────────────────
│ result! : REPORT
├──────────
│ result! = ok
└─────────────────────────────────────────
```

The conjunction operator $\land$ of the schema calculus allows us to combine this description with our previous description of $AddBirthday$:

$$AddBirthday \land Success.$$

This describes an operation which, for correct input, both acts as described by $AddBirthday$ and produces the result ok.

For each error that might be detected in the input, we define a schema which describes the conditions under which the error occurs and specifies that the appropriate report is produced. Here is a schema which specifies that the report $already\_known$ should be produced when the input $name?$ is already a member of $known$:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AlreadyKnown}} \\
\Xi BirthdayBook \\
name? : NAME \\
result! : REPORT \\
\hline
name? \in known \\
result! = already\_known \\
\hline
\end{array}$$

The declaration $\Xi BirthdayBook$ specifies that if the error occurs, the state of the system should not change.

We can combine this description with the previous one to give a specification for a robust version of $AddBirthday$:

$$RAddBirthday \mathrel{\widehat{=}} (AddBirthday \land Success) \lor AlreadyKnown.$$

This definition introduces a new schema called $RAddBirthday$, obtained by combining the three schemas on the right-hand side. The operation $RAddBirthday$ must terminate whatever its input. If the input $name?$ is already known, the state of the system does not change, and the result $already\_known$ is returned; otherwise, the new birthday is added to the database as described by $AddBirthday$, and the result ok is returned.

We have specified the various requirements for this operation separately, and then combined them into a single specification of the whole behaviour of the operation. This does not mean that each requirement must be implemented separately, and the implementations combined somehow. In fact, an implementation might search for a place to store the new birthday, and at the same time check that the name is not already known; the code for normal operation and error handling might be thoroughly mingled. This is an example of the abstraction which is possible when we use a specification language free from the constraints necessary in a programming language. The operators $\land$ and $\lor$ cannot (in general) be implemented efficiently as ways of combining programs, but this should not stop us from using them to combine specifications if that is a convenient thing to do.

The operation *RAddBirthday* could be specified directly by writing a single schema which combines the predicate parts of the three schemas *AddBirthday*, *Success* and *AlreadyKnown*. The effect of the schema $\lor$ operator is to make a schema in which the predicate part is the result of joining the predicate parts of its two arguments with the logical connective $\lor$. Similarly, the effect of the schema $\land$ operator is to take the conjunction of the two predicate parts. Any common variables of the two schemas are merged: in this example, the input *name*?, the output *result*!, and the four observations of the state before and after the operation are shared by the two arguments of $\lor$.

$$\begin{array}{|l}
\textit{RAddBirthday} \\
\hline
\Delta \textit{BirthdayBook} \\
\textit{name}? : \textit{NAME} \\
\textit{date}? : \textit{DATE} \\
\textit{result}! : \textit{REPORT} \\
\hline
(\textit{name}? \notin \textit{known} \land \\
\qquad \textit{birthday}' = \textit{birthday} \cup \{\textit{name}? \mapsto \textit{date}?\} \land \\
\qquad \textit{result}! = \textit{ok}) \lor \\
(\textit{name}? \in \textit{known} \land \\
\qquad \textit{birthday}' = \textit{birthday} \land \\
\qquad \textit{result}! = \textit{already\_known}) \\
\hline
\end{array}$$

In order to write *RAddBirthday* as a single schema, it has been necessary to write out explicitly that the state doesn't change when an error is detected, a fact that was implicitly part of the declaration Ξ*BirthdayBook* before.

A robust version of the *FindBirthday* operation must be able to report if the input name is not known:

$$\begin{array}{|l}
\textit{NotKnown} \\
\hline
\Xi \textit{BirthdayBook} \\
\textit{name}? : \textit{NAME} \\
\textit{result}! : \textit{REPORT} \\
\hline
\textit{name}? \notin \textit{known} \\
\textit{result}! = \textit{not\_known} \\
\hline
\end{array}$$

The robust operation either behaves as described by *FindBirthday* and reports success, or reports that the name was not known:

$$\textit{RFindBirthday} \mathrel{\widehat{=}} (\textit{FindBirthday} \land \textit{Success}) \lor \textit{NotKnown}.$$

The *Remind* operation can be called at any time: it never results in an error, so the robust version need only add the reporting of success:

$$\textit{RRemind} \mathrel{\widehat{=}} \textit{Remind} \land \textit{Success}.$$

The separation of normal operation from error-handling which we have seen here is the simplest but also the most common kind of modularization possible with the schema calculus. More complex modularizations include *promotion* or *framing*, where operations on a single entity – for example, a file – are made into operations on a named entity in a larger system – for example, a named file in a directory. The operations of reading and writing a file might be described by schemas. Separately, another schema might describe the way a file can be accessed in a directory under its name. Putting these two parts together would then result in a specification of operations for reading and writing named files.

Other modularizations are possible: for example, the specification of a system with access restrictions might separate the description of who may call an operation from the description of what the operation actually does. There are also facilities for generic definitions in Z which allow, for example, the notion of resource management to be specified in general, then applied to various aspects of a complex system.

1.4 From specifications to designs

We have seen how the Z notation can be used to specify software modules, and how the schema calculus allows us to put together the specification of a module from pieces which describe various facets of its function. Now we turn our attention to the techniques used in Z to document the design of a program which implements the specification.

The central idea is to describe the concrete data structures which the program will use to represent the abstract data in the specification, and to derive descriptions of the operations in terms of the concrete data structures. We call this process *data refinement*, and it is fully explained in Chapter 5. Often, a data refinement will allow some of the control structure of the program to be made explicit, and this is achieved by one or more steps of *operation refinement* or *algorithm development*.

For simple systems, it is possible to go from the abstract specification to the final program in one step, a method sometimes called *direct refinement*. In more complex systems, however, there are too many design decisions for them all to be recorded clearly in a single refinement step, and the technique of *deferred refinement* is appropriate. Instead of a finished program, the first refinement step results in a new specification, and this is then subjected to further steps of refinement until a program is at last reached. The result is a sequence of design documents, each describing a small collection of related design decisions. As the details of the data structures are filled in step by step, so more of the control structure can be filled in, leaving certain sub-tasks to be implemented in subsequent refinement steps. These sub-tasks can be made into subroutines in the final program, so the step-wise structure of the development leads to a modular structure in the program.

Program developments are often documented by giving an idealized account of the path from specification to program. In these accounts, the ideas all appear miraculously at the right time, one after another. There are no mistakes, no false starts, no decisions taken which are later revised. Of course, real program developments do not happen like that, and the earlier stages of a development are often revised many times as later stages cast new light on the system. In any case, specifications are seldom written without at least a rough idea of how they might be implemented, and it is very rare to find that something similar has not been implemented before. This does not mean that the idealized accounts are worthless, however. They are often the best way of presenting the decisions which have been made and the relationships between them, and such an account can be a valuable piece of documentation.

The rest of this chapter concentrates on data refinement in Z, although the results of the operation refinement which might follow it are shown. Two examples of data refinement are presented. The first shows direct refinement; the birthday book we specified in Section 1.2 is implemented using a pair of arrays. In the second example, deferred refinement is used to show the implementation of a simple checkpoint–restart mechanism. The implementation uses two submodules for which specifications in Z are derived as part of the refinement step. This demonstrates the way in which mathematics can help us to explore design decisions at a high level of abstraction.

1.5 Implementing the birthday book

The specification of the birthday book worked with abstract data structures chosen for their expressive clarity rather than their ability to be directly represented in a computer. In the implementation, the data structures must be chosen with an opposite set of criteria, but they can still be modelled with mathematical data types and documented with schemas.

In our implementation, we choose to represent the birthday book with two arrays, which might be declared by

names : **array** [1 . .] **of** *NAME*;
dates : **array** [1 . .] **of** *DATE*;

I have made these arrays 'infinite' for the sake of simplicity. In a real system development, we would use the schema calculus to specify a limit on the number of entries, with appropriate error reports if the limit is exceeded. Finite arrays could then be used in a more realistic implementation; but for now, this would just be a distraction, so let us pretend that potentially infinite arrays are part of our programming language. We shall, in any case, only use a finite part of them at any time. These arrays can be modelled mathematically by functions from the set $\mathbb{N}_1$ of strictly positive integers to *NAME* or *DATE*:

$$names : \mathbb{N}_1 \longrightarrow NAME$$
$$dates : \mathbb{N}_1 \longrightarrow DATE.$$

The element $names[i]$ of the array is simply the value $names(i)$ of the function, and the assignment $names[i] := v$ is exactly described by the specification

$$names' = names \oplus \{i \mapsto v\}.$$

The right-hand side of this equation is a function which takes the same value as *names* everywhere except at the argument i, where it takes the value v.

We describe the state space of the program as a schema. There is another variable *hwm* (for 'high water mark'); it shows how much of the arrays is in use.

```
── BirthdayBook1 ──────────────────────────
│ names : ℕ₁ ⟶ NAME
│ dates : ℕ₁ ⟶ DATE
│ hwm : ℕ
├──────────
│ ∀ i, j : 1 .. hwm •
│       i ≠ j ⇒ names(i) ≠ names(j)
└──────────────────────────────────────────
```

The predicate part of this schema says that there are no repetitions among the elements $names(1), \ldots, names(hwm)$.

The idea of this representation is that each name is linked with the date in the corresponding element of the array *dates*. We can document this with a schema *Abs* that defines the *abstraction relation* between the abstract state space *BirthdayBook* and the concrete state space *BirthdayBook*1:

```
── Abs ────────────────────────────────────
│ BirthdayBook
│ BirthdayBook1
├──────────
│ known = { i : 1 .. hwm • names(i) }
│
│ ∀ i : 1 .. hwm •
│       birthday(names(i)) = dates(i)
└──────────────────────────────────────────
```

This schema relates two points of view on the state of the system. The observations involved are both those of the abstract state – *known* and *birthday* – and those of the concrete state – *names*, *dates* and *hwm*. The first predicate says that the set *known* consists of just those names which occur somewhere among $names(1), \ldots, names(hwm)$. The set $\{ y : S \bullet \ldots y \ldots \}$ contains those values taken by the expression $\ldots y \ldots$ as y takes values in the set S, so *known* contains a name n exactly if $n = names(i)$ for some value of i such that $1 \leq i \leq hwm$. We can write this in symbols with an existential quantifier:

$$n \in known \Leftrightarrow (\exists\, i : 1 \mathrel{..} hwm \bullet n = names(i)).$$

The second predicate says that the birthday for $names(i)$ is the corresponding element $dates(i)$ of the array $dates$.

Several concrete states may represent the same abstract state: in the example, the order of the names and dates in the arrays does not matter, so long as names and dates correspond properly. The order is not used in determining which abstract state is represented by a concrete state, so two states which have the same names and dates in different orders will represent the same abstract state. This is quite usual in data refinement, because efficient representations of data often cannot avoid including superfluous information.

On the other hand, each concrete state represents only one abstract state. This is usual, because we don't expect to find superfluous information in the abstract state that does not need to be represented in the concrete state. It does sometimes happen that one concrete state represents several abstract states, but this is often a sign of a badly-written specification that has a bias towards a particular implementation.

Having explained what the concrete state space is, and how concrete states are related to abstract states, we can begin to implement the operations of the specification. To add a new name, we increase hwm by one, and fill in the name and date in the arrays:

$$
\begin{array}{|l}
\multicolumn{1}{l}{AddBirthday1} \\
\hline
\Delta BirthdayBook1 \\
name? : NAME \\
date? : DATE \\
\hline
\forall i : 1 \mathinner{..} hwm \bullet name? \neq names(i) \\[1ex]
hwm' = hwm + 1 \\
names' = names \oplus \{hwm' \mapsto name?\} \\
dates' = dates \oplus \{hwm' \mapsto date?\} \\
\hline
\end{array}
$$

This schema describes an operation which has the same inputs and outputs as *AddBirthday*, but operates on the concrete instead of the abstract state. It is a correct implementation of *AddBirthday*, because of the following two facts:

1. Whenever *AddBirthday* is legal in some abstract state, the implementation *AddBirthday*1 is legal in any corresponding concrete state.
2. The final state which results from *AddBirthday*1 represents an abstract state which *AddBirthday* could produce.

Why are these two statements true? The operation *AddBirthday* is legal exactly if its pre-condition $name? \notin known$ is satisfied. If this is so, the predicate

$$known = \{\, i : 1 \mathinner{..} hwm \bullet names(i) \,\}$$

from Abs tells us that $name?$ is not one of the elements $names(i)$:

$$\forall i : 1 \mathinner{..} hwm \bullet name? \neq names(i).$$

This is the pre-condition of *AddBirthday*1.

To prove the second fact, we need to think about the concrete states before and after an execution of *AddBirthday*1, and the abstract states they represent according to *Abs*. The two concrete states are related by *AddBirthday*1, and we must show that the two abstract states are related as prescribed by *AddBirthday*:

$$birthday' = birthday \cup \{name? \mapsto date?\}.$$

The domains of these two functions are the same, because

$$\begin{aligned}
\operatorname{dom} birthday' &= known' && \text{[invariant after]}\\
&= \{\, i : 1 \mathrel{..} hwm' \bullet names'(i) \,\} && \text{[from } Abs'\text{]}\\
&= \{\, i : 1 \mathrel{..} hwm \bullet names'(i) \,\} \cup \{names'(hwm')\} && [hwm' = hwm + 1]\\
&= \{\, i : 1 \mathrel{..} hwm \bullet names(i) \,\} \cup \{name?\}\\
& && [names' = names \oplus \{hwm' \mapsto name?\}]\\
&= known \cup \{name?\} && \text{[from } Abs\text{]}\\
&= \operatorname{dom} birthday \cup \{name?\}. && \text{[invariant before]}
\end{aligned}$$

There is no change in the part of the arrays which was in use before the operation, so for all i in the range $1 \mathrel{..} hwm$,

$$names'(i) = names(i) \land dates'(i) = dates(i).$$

For any i in this range,

$$\begin{aligned}
&birthday'(names'(i))\\
&\quad = dates'(i) && \text{[from } Abs'\text{]}\\
&\quad = dates(i) && [dates \text{ unchanged]}\\
&\quad = birthday(names(i)). && \text{[from } Abs\text{]}
\end{aligned}$$

For the new name, stored at index $hwm' = hwm + 1$,

$$\begin{aligned}
&birthday'(name?)\\
&\quad = birthday'(names'(hwm')) && [names'(hwm') = name?]\\
&\quad = dates'(hwm') && \text{[from } Abs'\text{]}\\
&\quad = date?\,. && \text{[spec. of } AddBirthday1\text{]}
\end{aligned}$$

So the two functions $birthday'$ and $birthday \cup \{name? \mapsto date?\}$ are equal, and the abstract states before and after the operation are guaranteed to be related as described by *AddBirthday*.

The description of the concrete operation uses only notation which has a direct counterpart in our programming language, so we can translate it directly into a subroutine to perform the operation:

```
**procedure** $AddBirthday(name : NAME;\ date : DATE)$;
**begin**
    $hwm := hwm + 1$;
    $names[hwm] := name$;
    $dates[hwm] := date$
**end**;
```

The second operation, *FindBirthday*, is implemented by the following operation, again described in terms of the concrete state:

```
── FindBirthday1 ──────────────────────────
│ $\Xi BirthdayBook1$
│ $name? : NAME$
│ $date! : DATE$
├──────────
│ $\exists i : 1 \mathrel{..} hwm \bullet$
│     $name? = names(i) \land date! = dates(i)$
└───────────────────────────────────────
```

The predicate says that there is an index i at which the *names* array contains the input $name?$, and the output $date!$ is the corresponding element of the array *dates*. For this to be possible, $name?$ must in fact appear somewhere in the array *names*: this is the pre-condition of the operation.

Since neither the abstract nor the concrete operation changes the state, there is no need to check that the final concrete state is acceptable, but we need to check that the pre-condition of *FindBirthday1* is sufficiently liberal, and that the output $date!$ is correct. The pre-conditions of the abstract and concrete operations are in fact the same: that the input $name?$ is known. The output is correct because for some i, $name? = names(i)$ and $date! = dates(i)$, so

$$\begin{aligned} date! &= dates(i) && \text{[spec. of } \mathit{FindBirthday1}\text{]} \\ &= birthday(names(i)) && \text{[from } \mathit{Abs}\text{]} \\ &= birthday(name?). && \text{[spec. of } \mathit{FindBirthday1}\text{]} \end{aligned}$$

The existential quantifier in the description of *FindBirthday1* leads to a loop in the program code, searching for a suitable value of i:

```
**procedure** $FindBirthday(name : NAME;$ **var** $date : DATE)$;
    **var** $i : INTEGER$;
**begin**
    $i := 1$;
    **while** $names[i] \neq name$ **do** $i := i + 1$;
    $date := dates[i]$
**end**;
```

The operation *Remind* poses a new problem, because its output *cards* is a *set* of names, and cannot be directly represented in the programming language. We can deal with it by introducing a new abstraction relation, showing how it can be represented by an array and an integer. Since this decision about representation affects the interface between the birthday book module we are developing and a program that uses it, this abstraction relation will form part of the documentation of that interface. Here is a schema *AbsCards* that defines the abstraction relation:

$$
\begin{array}{|l}
\textit{AbsCards} \\
\hline
\textit{cards} : \mathbb{P}\,\textit{NAME} \\
\textit{cardlist} : \mathbb{N}_1 \longrightarrow \textit{NAME} \\
\textit{ncards} : \mathbb{N} \\
\hline
\textit{cards} = \{\, i : 1 \mathrel{..} \textit{ncards} \bullet \textit{cardlist}(i) \,\} \\
\hline
\end{array}
$$

The concrete operation can now be described: it produces as outputs *cardlist* and *ncards*:

$$
\begin{array}{|l}
\textit{Remind}1 \\
\hline
\Xi \textit{BirthdayBook}1 \\
\textit{today}? : \textit{DATE} \\
\textit{cardlist}! : \mathbb{N}_1 \longrightarrow \textit{NAME} \\
\textit{ncards}! : \mathbb{N} \\
\hline
\{\, i : 1 \mathrel{..} \textit{ncards}! \bullet \textit{cardlist}!(i) \,\} \\
\qquad = \{\, j : 1 \mathrel{..} \textit{hwm} \mid \textit{dates}(j) = \textit{today}? \bullet \textit{names}(j) \,\} \\
\hline
\end{array}
$$

The set on the right-hand side of the equation contains all the names in the *names* array for which the corresponding entry in the *dates* array is *today*?. The program code for *Remind* uses a loop to examine the entries one by one:

```
procedure Remind(today : DATE;
                          var cardlist : array [1 . . ] of NAME;
                          var ncards : INTEGER);
    var j : INTEGER;
begin
    ncards := 0; j := 0;
    while j < hwm do begin
        j := j + 1;
        if dates[j] = today then begin
            ncards := ncards + 1;
            cardlist[ncards] := names[j]
        end
    end
end;
```

The initial state of the program has $hwm = 0$:

$$\begin{array}{|l}\hline \textit{InitBirthdayBook1} \\ \textit{BirthdayBook1} \\ \hline \textit{hwm} = 0 \\ \hline\end{array}$$

Nothing is said about the initial values of the arrays *names* and *dates*, because they do not matter. If the initial concrete state satisfies this description, and it is related to the initial abstract state by the abstraction schema *Abs*, then

$$\begin{aligned} \textit{known} &= \{\, i : 1 \mathinner{..} \textit{hwm} \bullet \textit{names}(i) \,\} && [\text{from } \textit{Abs}] \\ &= \{\, i : 1 \mathinner{..} 0 \bullet \textit{names}(i) \,\} && [\text{from } \textit{InitBirthdayBook1}] \\ &= \varnothing, && [1 \mathinner{..} 0 = \varnothing] \end{aligned}$$

so the initial abstract state is as described by *InitBirthdayBook*. This description of the initial concrete state can be used to write a subroutine to initialize our program module:

```
procedure InitBirthdayBook;
begin
    hwm := 0
end;
```

In this direct refinement, we have taken the birthday book specification and in a single step produced a program module which implements it. The relationship between the state of the book as described in the specification and the values of the program variables which represent that state was documented with an abstraction schema, and this allowed descriptions of the operations in terms of the program variables to be derived. These operations were simple enough to implement immediately, but in a more complex example, rules of operation refinement could be used to check the code against the concrete operation descriptions.

1.6 A simple checkpointing scheme

This example shows how refinement techniques can be used at a high level in the design of systems, as well as in detailed programming. What we shall call a *database* is simply a function from addresses to pages of data. We first introduce *ADDR* and *PAGE* as basic types:

$$[\textit{ADDR}, \textit{PAGE}].$$

We define *DATABASE* as an abbreviation for the set of all functions from *ADDR* to *PAGE*:

$$\textit{DATABASE} == \textit{ADDR} \longrightarrow \textit{PAGE}.$$

We shall be looking at a system which – from the user's point of view – contains two versions of a database. Here is a schema describing the state space:

CheckSys

$$\begin{array}{|l} working : DATABASE \\ backup : DATABASE \\ \hline \end{array}$$

This schema has no predicate part: it specifies that the two observations *working* and *backup* may be any databases at all, and need not be related in any way.

Most operations affect only the working database. For example, it is possible to access the page at a specified address:

Access

$$\begin{array}{|l} \Xi CheckSys \\ a? : ADDR \\ p! : PAGE \\ \hline p! = working(a?) \\ \hline \end{array}$$

This operation takes an address $a?$ as input, and produces as its output $p!$ the page stored in the working database at that address. Neither version of the database changes in the operation.

It is also possible to update the working database with a new page:

Update

$$\begin{array}{|l} \Delta CheckSys \\ a? : ADDR \\ p? : PAGE \\ \hline working' = working \oplus \{a? \mapsto p?\} \\ backup' = backup \\ \hline \end{array}$$

In this operation, both an address $a?$ and a page $p?$ are supplied as input, and the working database is updated so that the page $p?$ is now stored at address $a?$. The page previously stored at address $a?$ is lost.

There are two operations involving the back-up database. We can take a copy of the working database: this is the *CheckPoint* operation:

CheckPoint

$$\begin{array}{|l} \Delta CheckSys \\ \hline working' = working \\ backup' = working \\ \hline \end{array}$$

We can also restore the working database to the state it had at the last checkpoint:

$$\begin{array}{|l}
\hline
\textit{Restart} \\
\Delta \textit{CheckSys} \\
\hline
\textit{working}' = \textit{backup} \\
\textit{backup}' = \textit{backup} \\
\hline
\end{array}$$

This completes the specification of our system, and we can begin to think of how we might implement it. A first idea might be really to keep two copies of the database, so implementing the specification directly. But experience tells us that copying the entire database is an expensive operation, and that if checkpoints are taken frequently, then the computer will spend much more time copying than it does accessing and updating the working database.

A better idea for an implementation might be to keep only one complete copy of the database, together with a record of the changes made since creation of this master copy. The master copy consists of a single database:

$$\begin{array}{|l}
\hline
\textit{Master} \\
\textit{master} : \textit{DATABASE} \\
\hline
\end{array}$$

The record of changes made since the last checkpoint is a *partial function* from addresses to pages: it is partial because we expect that not every page will have been updated since the last checkpoint.

$$\begin{array}{|l}
\hline
\textit{Changes} \\
\textit{changes} : \textit{ADDR} \nrightarrow \textit{PAGE} \\
\hline
\end{array}$$

The concrete state space is described by putting these two parts together:

$$\begin{array}{|l}
\hline
\textit{CheckSys}1 \\
\textit{Master} \\
\textit{Changes} \\
\hline
\end{array}$$

How does this concrete state space mirror our original abstract view? The master database is what we described as the back-up, and the working database is $\textit{master} \oplus \textit{changes}$, the result of updating the master copy with the recorded changes. We can record this relationship with an abstraction schema:

$$\begin{array}{|l}
\hline
\textit{Abs} \\
\textit{CheckSys} \\
\textit{CheckSys}1 \\
\hline
\textit{backup} = \textit{master} \\
\textit{working} = \textit{master} \oplus \textit{changes} \\
\hline
\end{array}$$

The notation $\textit{master} \oplus \textit{changes}$ denotes a function which agrees with *master* everywhere except in the domain of *changes*, where it agrees with *changes*.

How can we implement the four operations? Accessing a page at address $a?$ should return a page from the working copy of the database, and according to the abstraction relation,

$$working(a?) = (master \oplus changes)(a?),$$

so a valid specification of $Access1$ is as follows:

```
┌─ Access1 ─────────────────────────────
│ $\Xi CheckSys1$
│ $a? : ADDR$
│ $p! : PAGE$
├───────────
│ $p! = (master \oplus changes)(a?)$
└───────────────────────────────────────
```

But we can do a little better than this: if $a? \in \mathrm{dom}\, changes$, then

$$(master \oplus changes)(a?)$$

is equal to $changes(a?)$ and if $a? \notin \mathrm{dom}\, changes$, then it is equal to $master(a?)$. So we can use operation refinement to develop the operation further; it is implemented by

```
**procedure** $Access(a : ADDR;$ **var** $p : PAGE);$
    **var** $r : REPORT;$
**begin**
    $GetChange(a, p, r);$
    **if** $r \neq ok$ **then**
        $ReadMaster(a, p)$
**end**;
```

What are the operations $GetChange$ and $ReadMaster$? We need give only their specifications here, and can leave their implementation to a later stage in the development. $GetChange$ operates only on the $changes$ part of the state; it checks whether a given page is present, returning a report and, if possible, the page itself:

```
┌─ GetChange ───────────────────────────
│ $\Xi Changes$
│ $a? : ADDR$
│ $p! : PAGE$
│ $r! : REPORT$
├───────────
│ $(a? \in \mathrm{dom}\, changes \land$
│     $p! = changes(a?) \land$
│     $r! = ok) \lor$
│ $(a? \notin \mathrm{dom}\, changes \land$
│     $r! = not\_present)$
└───────────────────────────────────────
```

As you will see, this is a specification which could be structured nicely with the schema $\lor$ operator. The *ReadMaster* operation simply returns a page from the *master* database:

$$\begin{array}{|l}
\hline \textit{ReadMaster} \\
\Xi \textit{Master} \\
a? : \textit{ADDR} \\
p! : \textit{PAGE} \\
\hline
p! = \textit{master}(a?) \\
\hline
\end{array}$$

For the *Update* operation, we want $\textit{backup}' = \textit{backup}$, so

$$\textit{master}' = \textit{backup}' = \textit{backup} = \textit{master}.$$

Also $\textit{working}' = \textit{working} \oplus \{a? \mapsto p?\}$, so we want

$$\textit{master}' \oplus \textit{changes}' = (\textit{master} \oplus \textit{changes}) \oplus \{a? \mapsto p?\}.$$

Luckily, the overriding operator $\oplus$ is associative: it satisfies the law

$$(f \oplus g) \oplus h = f \oplus (g \oplus h).$$

If we let $\textit{changes}' = \textit{changes} \oplus \{a? \mapsto p?\}$, then

$$\begin{aligned}
\textit{working}' &= \textit{working} \oplus \{a? \mapsto p?\} && [\text{spec. of } \textit{Update}] \\
&= (\textit{master} \oplus \textit{changes}) \oplus \{a? \mapsto p?\} && [\text{from } \textit{Abs}] \\
&= \textit{master} \oplus (\textit{changes} \oplus \{a? \mapsto p?\}) && [\text{associativity of } \oplus] \\
&= \textit{master}' \oplus \textit{changes}', && [\text{spec. of } \textit{Update}1]
\end{aligned}$$

and the abstraction relation is maintained. So the specification for *Update*1 is

$$\begin{array}{|l}
\hline \textit{Update}1 \\
\Delta \textit{CheckSys}1 \\
a? : \textit{ADDR} \\
p? : \textit{PAGE} \\
\hline
\textit{master}' = \textit{master} \\
\textit{changes}' = \textit{changes} \oplus \{a? \mapsto p?\} \\
\hline
\end{array}$$

This is implemented by an operation *MakeChange* which has the same effect as described here, but operates only on the *Changes* part of the state.

For the *CheckPoint* operation, we want $\textit{backup}' = \textit{working}$, so we immediately see that

$$\textit{master}' = \textit{backup}' = \textit{working} = \textit{master} \oplus \textit{changes}.$$

We also want $\textit{working}' = \textit{working}$, so

$$\textit{master}' \oplus \textit{changes}' = \textit{master} \oplus \textit{changes} = \textit{master}'.$$

This equation is solved by setting $changes' = \varnothing$, since the empty function $\varnothing$ is a right identity for $\oplus$, as expressed by the law

$$f \oplus \varnothing = f.$$

So a specification for $CheckPoint1$ is

$$\begin{array}{|l}
\hline
CheckPoint1 \\
\Delta CheckSys1 \\
\hline
master' = master \oplus changes \\
changes' = \varnothing \\
\hline
\end{array}$$

This can be refined to the code

$$MultiWrite(changes);\ ResetChanges$$

where *MultiWrite* updates the *master* database, and *ResetChanges* sets *changes* to $\varnothing$.

Finally, for the operation $Restart1$, we have $backup' = backup$, so we need $master' = master$, as for *Update*. Again, we want

$$master' \oplus changes' = master',$$

this time because $working' = backup$, so we choose $changes' = \varnothing$ as before:

$$\begin{array}{|l}
\hline
Restart1 \\
\Delta CheckSys1 \\
\hline
master' = master \\
changes' = \varnothing \\
\hline
\end{array}$$

This can be refined to a simple call to *ResetChanges*.

Now we have found implementations for all the operations of our original specification. In these implementations, we have used two new sets of operations, which we have specified with schemas but not yet implemented. One set, *ReadMaster* and *MultiWrite*, operates on the *master* part of the concrete state, and the other, containing *MakeChange*, *GetChange*, and *ResetChanges*, operates only on the *changes* part of the state. The result is two new specifications for what are in effect modules of the system, and in later stages they can be developed independently. Perhaps the *master* function would be represented by an array of pages stored on a disk, and *changes* by a hash table held in main store.

In mathematics, we can describe data structures with equal ease, whether they are held in primary or secondary storage. Operations are described in terms of their function, and it makes no difference whether their execution takes microseconds or hours to finish. Of course, the designer must be very closely concerned with the capabilities of the equipment to be used, and it is vital to distinguish primary storage, which though fast has limited capacity, from the slower but larger secondary storage. But we regard it as a strength and not a weakness of the mathematical method that it does not reflect this distinction. By modelling only the functional characteristics of a software module, a mathematical specification technique encourages a healthy *separation of concerns*: it helps the designer to focus his or her attention on functional aspects, and to compare different designs, even if they differ widely in performance.

The rest of this book is a reference manual for the notation and ideas used in the examples we have looked at here. In Chapter 2, an outline is given of the mathematical world of sets, relations and functions in which Z operates, and the way Z specifications describe objects in this world. These concepts are applied in Chapter 3, where an account of the Z language is given. The language is made usable by the library of definitions which is implicitly a part of every Z specification, described in Chapter 4 on 'the mathematical tool-kit'. This chapter contains many laws of the kind we have used in reasoning about the examples. Chapter 5 covers the conventions by which Z specifications are used to describe sequential programs, and the rules for developing concrete representations of data types from their mathematical specifications. The final chapter contains a summary of the syntax of the Z language described in the manual.