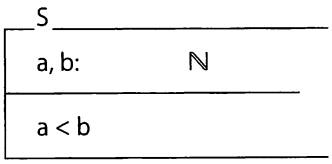
CHAPTER 6

Schemas

6.1 Schemas

A specification document in Z consists of narrative text written in a natural language such as English, interspersed with formal descriptions written in the Z notation. As a way of making a clear separation between these two components a graphical format called the *schema* (plural *schemas*) was devised. The schema also has various useful mathematical properties.

Here is an example of a schema:



The schema is referred to by the name *S* and it declares two *variables*, *a* and *b*. It also contains a *constraining predicate* which states that *a* must be less than *b*.

A schema can also be written in an equivalent *linear* form, which is more convenient for small schemas:

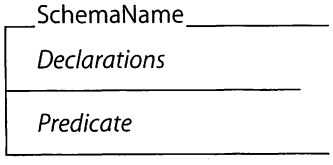
$$S == [a, b: \mathbb{N} \mid a < b]$$

The operator

$$==$$

means 'stands for'.

The general form of a schema is:



and the form of the linear schema is

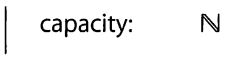
$$SchemaName == [\ declarations \mid predicate\ ]$$

It is possible to have an *anonymous* schema, in which case the schema name would be omitted. Furthermore, it is possible to have a schema with

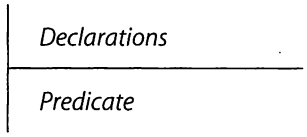
no predicate part. In this case the schema would simply declare a new variable or variables without applying a constraining predicate.

A variable introduced by a schema is *local* to that schema and may only be referenced in another schema by explicitly *including* the variable's defining schema. This is sometimes inconvenient and it is also possible to introduce variables which are available throughout the specification. These are known as *global* variables and are introduced by an *axiomatic* definition. Such values cannot be changed by operations of the specification.

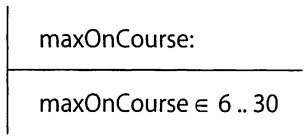
For example, the fixed capacity of an aircraft is introduced as a global variable by:



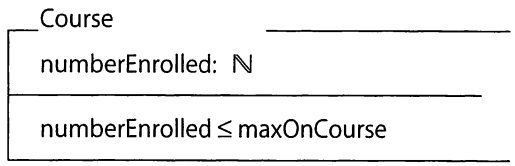
If you wish to add a constraining predicate to the variable you can use the general form:



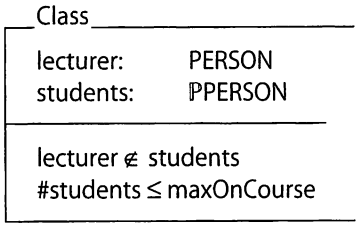
For example, to introduce a limit to the number of participants who may enrol on a course, *maxOnCourse*, where this limit must be in the range 6 to 30, you could use the following:



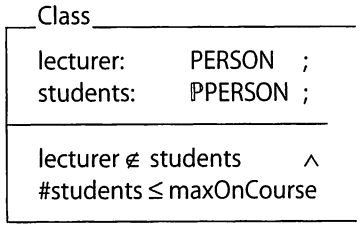
Schemas can make reference to *capacity* and *maxOnCourse* without explicitly including their defining schemas:



If a schema contains several lines of declarations then each line is regarded as being terminated by a semicolon. Furthermore, if the predicate part consists of more than one line then the lines are regarded as being joined by *and* operators. For example:



is an abbreviation of

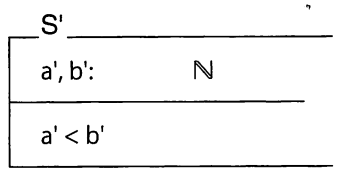


6.2 Schema calculus

Schemas can be regarded as units and manipulated by various operators that are analogous to the logical operators.

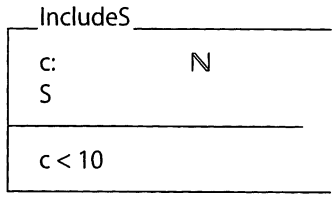
6.2.1 Decoration

The schema name *S* decorated with a prime, *S'*, is defined to be the same as the schema *S* with all its *variables* decorated with a prime. It is used to signify the value of a schema *after* some operation has been carried out. It is as if the schema *S'* had been defined:

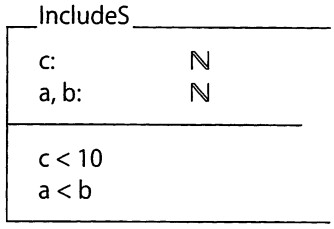


6.2.2 Inclusion

The name of a schema can be included in the declarations of another schema. The effect is for the included schema to be *textually imported*: its declarations are merged with those of the including schema and its predicate part is conjoined ('*and*ed') with that of the including schema. (It follows that any variables that have the same name in each schema must be of the same type. If not then the schema conjunction is illegal.)

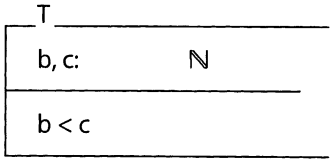


means



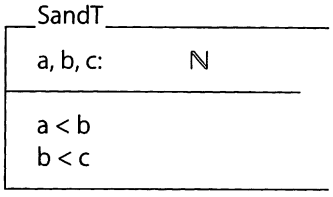
6.2.3 Schema conjunction

Two schemas can be joined by a *schema conjunction* operator, written like the *logical* conjunction operator. The effect is to make a new schema with the declarations of the two component schemas merged and their predicates conjoined ('*and*ed'). Given *S* as before and *T*:



SandT == S ∧ T

means



Variables with the same name, such as the two *b*'s here, are merged if they have the same type. If not then the schema conjunction is illegal.

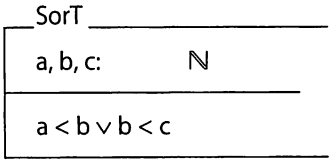
6.2.4 Schema disjunction

Two schemas can be joined by a *schema disjunction* operator, written just like the logical disjunction. The effect is to make a new schema with the

declarations of the two component schemas merged and their predicates disjoined ('*or*ed'). (It follows that any variables that have the same name in each schema must be of the same type. If not then the schema conjunction is illegal.) Given S and T as before:

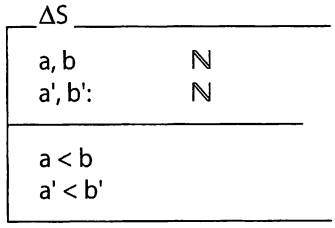
$$SorT == S \vee T$$

means



6.2.5 Delta convention

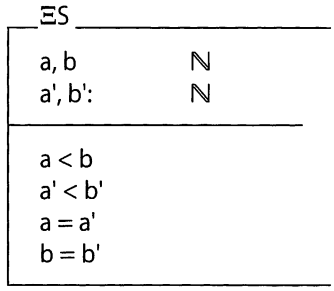
The convention that the value of a variable before an operation is denoted by the undecorated name of the variable, and the value after an operation by the name decorated by a prime (') character, is used in the *delta* naming convention. By convention, a schema with the Greek character capital delta ('Δ') as the first character of its name, such as ΔS, is defined to be:



The Greek delta character, in ΔS, is used, as in other areas of mathematics, to signify a change in S.

6.2.6 Xi convention

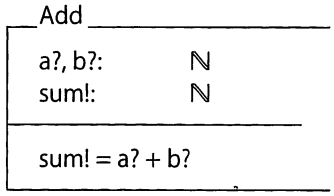
By convention, a schema with the Greek character capital *xi* (Ξ) as the first character of its name, such as ΞS, is defined to be the same as ΔS but with the constraint that the new value of every variable is the same as the old. This Greek symbol is chosen for its visual similarity to the *equivalence* symbol, $\equiv$, showing that the new state is equivalent to the old. It is as if the schema ΞS were written:



6.3 Decoration of input and output variables

A convention is used to decorate the variables of a schema which specifies an operation. Finishing the variable's name with a question mark (?) indicates that the variable is an *input* to the schema. Finishing the variable's name with an exclamation mark (!) indicates that the variable is an *output* from the schema. Note that the question mark and exclamation mark are simply characters in the variable's name.

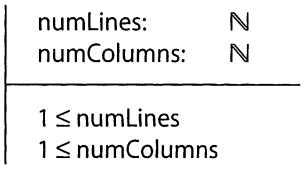
6.4 Simple example of schema with input and output



6.5 Example of schemas with input

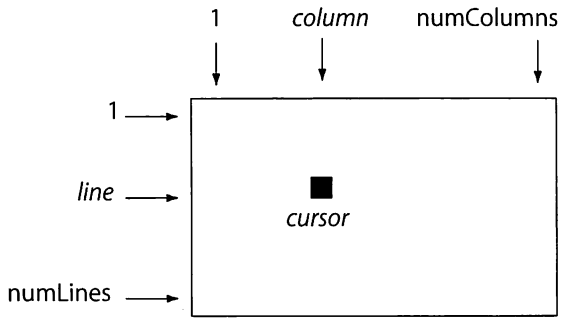
The display of a computer terminal shows lines of characters with each line consisting of a fixed number of columns containing a character in a fixed-width typeface. A *cursor* marks a current position of interest on the display. The user can press cursor-control *keys* on the keyboard, some of which directly control the position of the cursor.

KEY ::= home | return | left | right | up | down



The lines are numbered from 1 to *numLines* down the display and the columns are numbered 1 to *numColumns* across the display. (A display with fewer than on lines, or fewer than one column is useless.)

Figure 6.1



6.5.1 The state

At any time the cursor is within the bounds of the display. The state of the cursor can be described by the schema *Cursor:*

Cursor

line: ℕ
column: ℕ

---

line ∈ 1 .. numLines
column ∈ 1 .. numColumns

6.5.2 Home key

The operations for moving the cursor can be built up one at a time. The simplest is the response to the *home* key. It causes the cursor to move to the top-left corner of the display.

HomeKey

ΔCursor
key?: KEY

---

key? = home
line' = 1
column' = 1

Reminder: We do not need to declare the schema Δ*Cursor* since it is automatically defined to be:

$$
\begin{array}{|l}
\Delta Cursor \\
\hline
line, line' : \mathbb{N} \\
column, column' : \mathbb{N} \\
\hline
line \in 1 \,..\, numLines \\
line' \in 1 \,..\, numLines \\
column \in 1 \,..\, numColumns \\
column' \in 1 \,..\, numColumns \\
\hline
\end{array}
$$

6.5.3 Down key

The action for responding to the *down* key is given next. It causes the cursor to move to the same column position on the next line down. It is easiest to start by dealing with what happens if the cursor is *not* on the bottom line of the display.

$$
\begin{array}{|l}
DownKeyNormal \\
\hline
\Delta Cursor \\
key? : KEY \\
\hline
key? = down \\
line < numLines \\
line' = line + 1 \\
column' = column \\
\hline
\end{array}
$$

The next schema deals with what happens when the cursor *is* on the bottom line of the display.

$$
\begin{array}{|l}
DownKeyAtBottom \\
\hline
\Delta Cursor \\
key? : KEY \\
\hline
key? = down \\
line = numLines \\
line' = 1 \\
column' = column \\
\hline
\end{array}
$$

Note that the cursor has been defined to *wrap round* to the top line of the display.

The full behaviour is given by:

$$DownKey == DownKeyNormal \lor DownKeyAtBottom$$

The behaviour of the down key is defined by '*or*ing' the 'normal' behaviour of the down key with the behaviour of the down key when at the bottom.

6.5.4 Return key

The response to the *return* key is to move to the leftmost column of the next line down or the top line if the cursor was on the bottom line. For contrast this is given as a single schema:

ReturnKey

$$\Delta Cursor$$
$$key? : KEY$$

$$key? = return$$
$$column' = 1$$
$$((line < numLines \land line' = line + 1)$$
$$\lor$$
$$(line = numLines \land line' = 1))$$

6.5.5 Right key

Next, the operation for moving right is given. It is easiest to deal first with what happens when the cursor is *not* at the far right of the display:

RightKeyNormal

$$\Delta Cursor$$
$$key? : KEY$$

$$key? = right$$
$$column < numColumns$$
$$column' = column + 1$$
$$line' = line$$

The next schema deals with the situation when the cursor *is* at the end of a line (other than the *bottom* line of the display). Note that the cursor wraps round to the start of the next line:

RightKeyAtEnd

$$\Delta Cursor$$
$$key? : KEY$$

$$key? = right$$
$$column = numColumns$$
$$column' = 1$$
$$line < numLines$$
$$line' = line + 1$$

Finally, a separate schema deals with the situation where the cursor *is* at the end of the *bottom* line. The cursor wraps round to the left of the top line:

RightKeyAtBottom

ΔCursor
key?: KEY

key? = right
column = numColumns
column' = 1
line = numLines
line' = 1

These schemas can be combined to form one schema that defines the response of the cursor to the right key in all initial positions of the cursor:

RightKey ==
RightKeyNormal ∨ RightKeyAtEnd ∨ RightKeyAtBottom

For the sake of illustration here is the expansion of *RightKey*:

RightKey

ΔCursor
key?: KEY

(key? = right ∧
column < numColumns ∧
column' = column + 1 ∧
line' = line)
∨
(key? = right ∧
column = numColumns ∧
column' = 1 ∧
line < numLines ∧
line' = line + 1)
∨
(key? = right ∧
column = numColumns ∧
column' = 1 ∧
line = numLines ∧
line' = 1)

RightKey can be simplified to

RightKey

$$
\begin{array}{|l}
\Delta Cursor \\
key?: \quad KEY \\
\hline
key? = right \land \\
((column < numColumns \land \\
\quad column' = column + 1 \land \\
\quad line' = line) \\
\lor \\
\quad (column = numColumns \land \\
\quad column' = 1 \land \\
\qquad (line < numLines \land \\
\qquad line' = line + 1) \\
\lor \\
\quad (line = numLines \land \\
\quad line' = 1))) \\
\hline
\end{array}
$$

Of course, the behaviour of the cursor at the end of the line and at the bottom-right of the display need not be defined as here. The style used here of defining separate schemas to describe the behaviour in these situations makes it easier to understand what happens in these cases.

6.5.6 Cursor-control key action

The action of the cursor on the pressing of any of these *cursor-control* keys can be defined by

$$CursorControlKey == HomeKey \lor ReturnKey \lor UpKey \lor DownKey \lor LeftKey \lor RightKey$$

6.6 Further operations on schemas

6.6.1 Renaming

The observations in a schema may be *renamed* by the following form:

schemaName [newName / oldName]

For example, given:

Aircraft

$$
\begin{array}{|l}
onboard: \quad \mathbb{P}PERSON \\
\hline
\#onboard \leq capacity \\
\hline
\end{array}
$$

then

Ship == Aircraft [passengers / onboard]

gives:

Ship

$$\text{passengers}: \mathbb{P}\,\text{PERSON}$$

$$\#\text{passengers} \leq \text{capacity}$$

6.6.2 Hiding

The schema *hiding* operator hides specified variables so that they are no longer variables of the schema and simply become local variables of existential operators in the predicate part of the schema. For example, given:

S

$$a: \mathbb{N}$$
$$b: \mathbb{N}$$

$$a < b$$

then

BHidden == S \ (b)

gives the schema:

BHidden

$$a: \mathbb{N}$$

$$\exists\, b: \mathbb{N} \bullet a < b$$

Several variable names can be given in the brackets.

6.6.3 Projection

Schema *projection* is similar to hiding except *all but* the named variables are hidden. Given S as above then

AProjected == S ↾ (a)

gives the schema:

AProjected

$$a: \mathbb{N}$$

$$\exists\, b: \mathbb{N} \bullet a < b$$

6.6.4 Schema composition

The *composition* of schema S with schema T is written:

$$S \fatsemi T$$

and signifies the effect of doing S, then doing T. It is equivalent to renaming the variables describing the after state of S to some temporary names and the equivalent variables describing the before state of T with the same temporary names and then hiding the temporary names.

For example, to show the effect of pressing the right-key and then the left-key on a visual display, using the definition of *CursorControlKey*:

$$\mathit{PressRight} == \mathit{CursorControlKey} \land [key?: KEY \mid key? = right]$$
$$\mathit{PressLeft} == \mathit{CursorControlKey} \land [key?: KEY \mid key? = left]$$

$$\mathit{PressRight} \fatsemi \mathit{PressLeft}$$
$$==$$

$$\mathit{PressRight}[tempCol / column', tempLine / line'] \land$$
$$\mathit{PressLeft}[tempCol / column, tempLine / line]$$
$$\setminus (tempCol, tempLine)$$

6.7 Overall structure of a Z specification document

A Z specification document consists of mathematical text in the Z notation, interleaved with explanatory text in a natural language. The explanatory text should be expressed in terms of the problem and should not refer directly to the mathematical formulation. This rule is broken only in the case of documents intended as tutorials on Z.

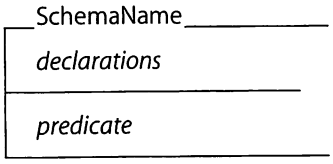
6.7.1 Sections of a Z document

The sections of a Z document are as follows:

- Introduction.
- The types used in the specification (their introduction is sometimes deferred until they are needed).
- The state and its invariant properties.
- An operation to set the variables to some initial state.
- Operations and enquiries.
- Error handling.
- Final versions of operations and enquiries.

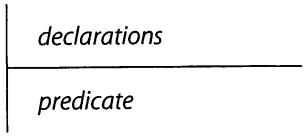
Some simple examples of Z specification documents appear in the next chapter.

6.8 Summary of notation

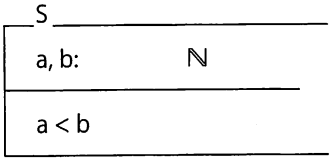


SchemaName == [declarations | predicate]

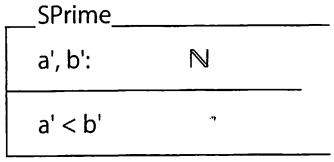
6.8.1 Axiomatic definition



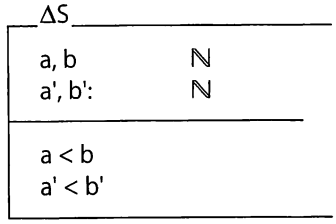
6.8.2 Decoration



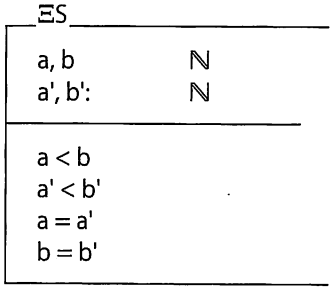
SPrime == S'



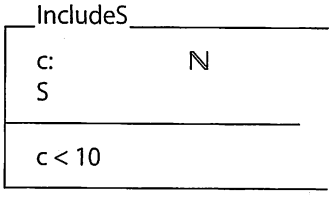
6.8.3 Delta convention



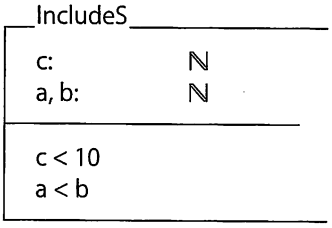
6.8.4 Xi convention



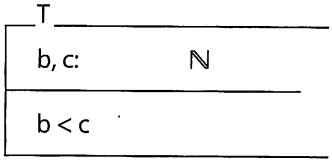
6.8.5 Inclusion



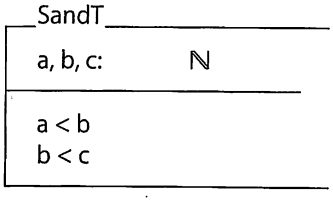
==



6.8.6 Conjunction

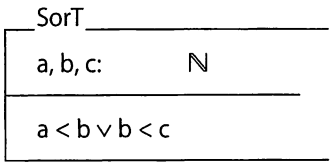


SandT == S ∧ T
==



6.8.7 Disjunction

SorT == S ∨ T
==



S[new / old,...]	schema renaming
$S \setminus (x_1, x_2, \ldots, x_n)$	schema hiding
$S \upharpoonright (x_1, x_2, \ldots, x_n)$	schema projection
pre S	precondition of S
S ⨟ T	schema composition: S, then T

EXERCISES

1. Define a schema *LinesRemaining* which delivers the number of lines below the cursor as an output parameter.

   Make use of schemas from the examples in this chapter.
2. Define a schema *UpKey* to define the operation of pressing the *up* key.
3. Define a schema *LeftKey* to define the operation of pressing the *left* key.
4. Devise a schema to define pressing the *down* key where the cursor does not move at all if it is already on the bottom line of the screen.

   Make use of schemas from the examples in this chapter and from your solutions to the previous exercises.
5. Devise a schema to define pressing the *right* key where the cursor does not move at all if it is already on the last column of the screen.

   Make use of schemas from the examples in this chapter and from your solutions to the previous exercises.
6. The Houses of Parliament (*HP*) consist of Members of Parliament (*MP*s), some of whom are in the *Cabinet*. There is an MP called the Prime Minister (*PM*) who is in the Cabinet.

   The only type involved is:

   [PERSON] the set of all persons

   The *state* of the Houses of Parliament is described by the schema *HP*:

HP

$$\begin{array}{l} MPs: \quad \mathbb{P}PERSON \\ Cabinet: \quad \mathbb{P}PERSON \\ PM: \quad PERSON \\ \hline Cabinet \subseteq MPs \\ PM \in Cabinet \end{array}$$

(a) Does the specification state that the Prime Minister has to be a Member of Parliament? If so, how?
(b) What would you add to the schema *HP* to have a Deputy Prime Minister (*DPM*), who is also in the Cabinet?
(c) An operation *ReplacePM* replaces the Prime Minister with a person *newPM*?.

ReplacePM

$$\begin{array}{l} \Delta HP \\ newPM?: PERSON \\ \hline newPM? \in MPs \\ newPM? \neq PM \\ Cabinet = Cabinet \cup \{newPM?\} \\ PM' = newPM? \\ MPs' = MPs \end{array}$$

Why does the schema include the line:

$$newPM? \neq PM$$

(d) Does the new Prime Minister have to be chosen from the Cabinet?
(e) Does the outgoing Prime Minister have to leave the Cabinet?
(f) May the outgoing Prime Minister leave the Cabinet?

7. Two operations, *ChangeCabinet1* and *ChangeCabinet2*, are proposed:

ChangeCabinet1

$$\begin{array}{l} \Delta HP \\ newCab?: \mathbb{P}PERSON \\ \hline newCab? \subseteq MPs \\ Cabinet' = newCab? \\ PM' = PM \\ MPs' = MPs \end{array}$$

ChangeCabinet2

ΔHP
$newCab?: \mathbb{P} PERSON$

$newCab? \subseteq MPs$
$newCab? \cap Cabinet = \varnothing$
$Cabinet' = newCab?$
$PM' = PM$
$MPs' = MPs$

(a) Explain why the line:

$$newCab? \subseteq MPs$$

is included.

(b) Explain the difference between the effect of *ChangeCabinet1* and *ChangeCabinet2*.

(c) Explain the logical error in *ChangeCabinet2*.