Introductory Notes on Specification with Z

Michael Butler
Dept. of Electronics and Computer Science
University of Southampton

March 12, 2001

1 Introduction

Z is a formal specification language for computer systems which is based on set theory and predicate logic.

There are several textbooks on Z in the library, in particular:

- *The Mathematics of Software Construction*. A. Norcliffe & G. Slater. Ellis Horwood, 1991.
- *Z User Manual*. M.A. McMorran & J.E. Nicholls. IBM Technical Report, 1989.
- *The Z Notation - A Reference Manual*. J.M. Spivey. Prentice–Hall, 1989.
- *An Introduction to Formal Specification and Z*. B. Potter, J. Sinclair & D. Till. Prentice–Hall, 1996.

The basic until of specification in Z is a *schema*. A Z schema consists of a name, a declaration of variables, and a predicate:

SchemaName

$x : X$

Predicate

Here, variable x is declared to be of type X (see section 2.2). Note that the declaration part may declare more than one variable. The predicate part is a predicate (see section 2.3) whose free variables are those of the declaration plus any constants.

A system specification in Z consists of some state variables, an initialisation, and a set of operations on the state variables. The state variables will also have some invariants associated with them representing "healthiness conditions" which must always be satisfied. Usually all of these are specified using schemas. For example, the state variables of a counter system may be specified using the following schema:

Counter

$ctr : \mathbb{N}$

$0 \leq ctr \leq max$

Here, *ctr* is declared to be a natural number and the predicate part describes an invariant that must be satisfied by *ctr*, the state variable of the system.

An initialisation may be specified as follows:

InitCounter

$$Counter$$

$$ctr = 0$$

An operation is specified in Z with a predicate relating the state before and after the invocation of that operation. For example, an operation to increment the counter may be specified as follows:

Increment

$$\Delta Counter$$

$$ctr < max$$
$$ctr' = ctr + 1$$

The declaration $\Delta Counter$ means that the state *Counter* is changed by the operation. In the predicate, the new value of a variable is primed (ctr'), while the old value is unprimed. So the above predicate states that the new value of the counter, ctr', is the old value plus one. Note that there is an implicit conjunction (logical-and) between successive lines of the predicate part of a schema.

As well as changing the state variables, an operation may also have input and output parameters. Input parameter names are usually suffixed with '?', while output parameter names are suffixed with '!'. For example, the following operation for decrementing the counter has as an input parameter, the amount by which the counter should be decremented:

Decrement

$$\Delta Counter$$
$$d? : \mathbb{N}$$

$$ctr \geq d?$$
$$ctr' = ctr - d?$$

The following operation has an output parameter which is the value of the counter:

Display

$$\Xi Counter$$
$$c! : \mathbb{N}$$

$$c! = ctr$$

Here, the declaration $\Xi Counter$ means that the operation cannot change the state of *Counter*, so $ctr' = ctr$.

2 Sets and Logic

2.1 Sets

Sets are the most basic types in Z. Examples of sets include:

$\{ 3, 6, 7 \}$
$\{ windows, unix, mac \}$
$\{ false, true \}$
$\mathbb{N}$ (the set of natural numbers)
$\mathbb{Z}$ (the set of integers)
$\mathbb{R}$ (the set of real numbers)
$\{\}$ (the empty set)

Set Membership:

$mac \in \{ windows, unix, mac \}$
$linux \notin \{ windows, unix, mac \}$
$10 \in \mathbb{N}$
$10.5 \notin \mathbb{N}$
$10.5 \in \mathbb{R}$

Set Equality:

$$\{ 3, 6, 7 \} = \{ 7, 6, 3, 6 \}$$

The following operators may be applied to sets:

Union: $S \cup T$

Intersection: $S \cap T$

Difference: $S \setminus T$

Subset: $S \subseteq T$
E.g., $\{c, b\} \subseteq \{a, b, c\}$.

Power Set: $\mathbb{P} S$ (set of subsets of S).
E.g.,

$$\begin{aligned} \mathbb{P}\{a, b, c\} = \{ & \{\}, \{a\}, \{b\}, \{c\}, \\ & \{a, b\}, \{b, c\}, \{a, c\}, \\ & \{a, b, c\} \} \end{aligned}$$

2.2 Types

Types are used to differentiate the various forms of data present in a specification. Advantages of using types are that they

- help to structure specifications by differentiating objects;
- help to prevent errors by not allowing us to write meaningless things;
- they can be checked by computer.

The declaration $x : T$ says that x is of type T, where T is a set. This is like saying $x \in T$.

$x : \mathbb{N}$
$z : \mathbb{R}$
$unix : \{ windows, unix, mac \}$
$7 : \mathbb{N}$
$(3 + 5) : \mathbb{N}$

What are the types of the following expressions?

mac
log y
sin $(\pi/2)$
$(a + b) \times (3!)$

A new *basic type* T is introduced to a specification by putting its name in square brackets:

$[T]$

This allows us to name the types of a specification without saying what kind of objects they contain. For example, a specification of an address book might introduce the basic types *Name* and *Address* without worrying about the structure of these types:

$$[\, \mathit{Name},\ \mathit{Address} \,]$$

If we know the exact values of a type we use an *enumerated type* declaration:

$$\mathit{Direction} == \mathit{north} \mid \mathit{south} \mid \mathit{east} \mid \mathit{west}$$

Sets have types too. The type of the set $\{3, 4, 5\}$ is "*set of* $\mathbb{N}$". More precisely, this is written:

$$\{3, 4, 5\} : \mathbb{P}\,\mathbb{N}$$

Assume S and T have type $\mathbb{P}\,M$. What are the types of:

$$S \cup T$$
$$S \cap T$$

What about $\{\}$? The type of $\{\}$ is given explicitly: $\{\}_M : \mathbb{P}\,M$

What is the type of $\{\ \{3, 4\},\ \{\}_{\mathbb{N}},\ \{7\}\ \}$?

Expressions which are incorrectly typed are meaningless:

$$\{4, 6, \mathit{unix}\}$$
$$\{\mathit{windows}, \mathit{mac}\} \cup \{\mathit{bwm}, \mathit{rover}, \mathit{ford}\}$$

2.3 Predicates

Predicates are used to state truth properties of values in a specification. Examples of simple predicates include:

false	*true*	$1 < (a/2)$
$(x + 1) = 7$	*even* 6	$\pi \in \mathbb{R}$

Compound predicates are formed using the following logical operators:

And	$\mathcal{A} \wedge \mathcal{B}$	(*Conjunction*)
Or	$\mathcal{A} \vee \mathcal{B}$	(*Disjunction*)
Implies	$\mathcal{A} \Rightarrow \mathcal{B}$	
Not	$\neg\, \mathcal{A}$	

Logical-and is sometimes called *conjunction* and logical-or is sometimes called *disjunction*.

Examples of compound predicates include:

$$(x \geq y) \wedge (y \geq 0)$$
$$(x > 20)\ \vee\ (x = 4)$$
$$(x > 0)\ \Rightarrow\ x/x = 1$$
$$(\neg\, (a \in S)) \vee (a \in T)$$

$x \notin S$ is short for $\neg\, (x \in S)$. $x \neq y$ is short for $\neg\, (x = y)$.

Universal Quantification is written as follows:

$$(\forall\, x : T \bullet \mathcal{A})$$

This is true when $\mathcal{A}$ holds for all values x of type T. Here x is a said to be a *quantified* or *bound* variable. Example: $(\forall x : \mathbb{Z} \bullet x - x = 0)$.

Existential Quantification is written as follows:

$$(\exists x : T \bullet \mathcal{A})$$

This is true when $\mathcal{A}$ holds for some value x of type T.

Example: $(\exists x : \mathbb{Z} \bullet x * x = 16)$.

2.4 Set Comprehension

A set comprehension is written as follows:

$$\{ x : T \mid \mathcal{A} \}$$

This stands for the set of objects x of type T satisfying predicate $\mathcal{A}$.

Examples:

$$\begin{array}{rcl} \mathbb{N} & = & \{ n : \mathbb{Z} \mid n \geq 0 \} \\ S, T : \mathbb{P} M & & \\ S \cup T & = & \{ x : M \mid x \in S \ \lor \ x \in T \} \\ & & \\ S \cap T & = & ? \\ & & \\ S \setminus T & = & ? \\ & & \\ \{\}_M & = & \{ x : M \mid \mathit{false} \} \\ M & = & \{ x : M \mid \mathit{true} \} \end{array}$$

3 Example Specification: Check-In/Check-Out

We consider a specification of a system used to check staff members in and out of a building. Since we will be dealing with elements of type staff, we introduce the type *Staff* as a basic type:

$$[\mathit{Staff}]$$

The state of the system is described by the following schema

Log

$$\mathit{users}, \mathit{in}, \mathit{out} : \mathbb{P}\, \mathit{Staff}$$

$$\begin{array}{l} \mathit{in} \cap \mathit{out} \ = \ \{\} \ \land \\ \mathit{in} \cup \mathit{out} \ = \ \mathit{users} \end{array}$$

The state consists of three components modelling

- the set of users of the system,
- the set of staff members who are currently in and
- the set of staff members who are currently out.

The predicate part of the state schema describes an *Invariant* of the system. The invariant says that

- No staff member is simultaneously in and out.
- The set of users of the system is exactly the union of those who are in and those who are out.

An operation to check a staff member into the building is specified as follows:

CheckIn

$$\Delta Log$$
$$name? : Staff$$

$$name? \in out$$
$$in' = in \cup \{name?\}$$
$$out' = out \setminus \{name?\}$$
$$users' = users$$

This has an input parameter representing the member of staff to be checked in. The predicate part says that:

- The staff member to be checked in must currently be out. This is a *pre-condition* on the operation.
- The staff member is added to the set *in*.
- The staff member is removed from the set *out*.
- The overall set of users remains unchanged.

Similarly, an operation to check a staff member out of the building may be specified as follows:

CheckOut

$$\Delta Log$$
$$name? : Staff$$

$$name? \in in$$
$$out' = out \cup \{name?\}$$
$$in' = in \setminus \{name?\}$$
$$users' = users$$

A query operation to check whether a particular member of staff is in or out will give an output parameter of the following type:

$$QueryReply == is\_in \mid is\_out$$

The operation is then specified as:

StaffQuery

$$\Xi Log$$
$$name? : Staff$$
$$reply! : QueryReply$$

$$name? \in users$$
$$name? \in in \Rightarrow reply! = is\_in$$
$$name? \in out \Rightarrow reply! = is\_out$$

Here we used the declaration ΞLog to say that the operation makes no change to the state of the system.

3.1 Initialisation

Typically the system would be initialised so that all sets are empty.

InitLog

Log

$users = \{\}$
$in = \{\}$
$out = \{\}$

Just to recap, the specification contains:

- **State Schema:** Components/Objects of system.
- **Invariant:** Static relationship between state components.
- **Operation Schemas:**
  - Condition on Input parameters.
  - Relationship between before- and after-states.
  - Output parameters.
- **Initialisation**

3.2 More Operations

Here is an outline of an operation to register a new staff member:

Register

ΔLog
$name? : Staff$

Fill in the gaps.

Do the same for an operation to check which staff are currently checked-in:

QueryIn

ΞLog
$names! : \mathbb{P}\, Staff$

---

4 Combining Schemas

Schemas may be combined using conjunction and disjunction to form new schemas. Suppose we have the following two schemas:

*Schema*1

$$x : X; \quad y : Y$$

$$\mathcal{A}(x, y)$$

*Schema*2

$$z : Z; \quad x : X$$

$$\mathcal{B}(z, x)$$

We can now define *Schema*3 to be the conjunction of *Schema*1 and *Schema*2 and we can define *Schema*4 to be the disjunction of these:

$$\begin{aligned} \mathit{Schema}3 &== \mathit{Schema}1 \land \mathit{Schema}2 \\ \mathit{Schema}4 &== \mathit{Schema}1 \lor \mathit{Schema}2 \end{aligned}$$

*Schema*3 really stands for the following schema were the declaration parts of *Schema*1 and *Schema*2 are merged, and the predicate parts of both schemas are conjoined:

*Schema*3

$$x : X; \quad y : Y; \quad z : Z$$

$$\mathcal{A}(x, y) \ \land \ \mathcal{B}(z, x)$$

It is important when combining schemas that the types of any common variables are the same.

*Schema*4 may be expanded in a similar way except that this time the predicates of *Schema*1 and *Schema*2 are disjoined:

*Schema*4

$$x : X; \quad y : Y; \quad z : Z$$

$$\mathcal{A}(x, y) \ \lor \ \mathcal{B}(z, x)$$

4.1 Robust Operations

Schema composition is very useful for making a specification of an operation more robust, that is, able to deal with potential error cases. Consider the *StaffQuery* operation again:

StaffQuery

$$\begin{aligned} &\Xi \mathit{Log} \\ &\mathit{name}? : \mathit{Staff} \\ &\mathit{reply}! : \mathit{QueryReply} \end{aligned}$$

$$\begin{aligned} &\mathit{name}? \in \mathit{users} \\ &\mathit{name}? \in \mathit{in} \ \Rightarrow \ \mathit{reply}! = \mathit{is\_in} \\ &\mathit{name}? \in \mathit{out} \ \Rightarrow \ \mathit{reply}! = \mathit{is\_out} \end{aligned}$$

The predicate requires that $name? \in users$ must hold in order for the output to be valid. If $name? \in users$ does not hold when we try to execute the program, the specification says nothing about what the output should be. To deal with this error case, we define the following schema which gives the reply *not_registered* when $name? \notin users$:

BadStaffQuery

ΞLog
$name? : Staff$
$reply! : QueryReply$

$name? \notin users$
$reply! = not\_registered$

Here, we assume that the type *QueryReply* has three possible values:

$$QueryReply == is\_in \mid is\_out \mid not\_registered$$

A robust version of the query operation is then defined as follows:

$$\begin{aligned} RobustStaffQuery \quad == \quad & StaffQuery \\ & \lor\ BadStaffQuery \end{aligned}$$

RobustStaffQuery will always produce a valid outcome whatever the value of $name?$. The ability to combine schemas in this way means that we can deal with the normal behaviour of an operation first and then separately deal with the error cases.

The *CheckIn* operation was specified as:

CheckIn

ΔLog
$name? : Staff$

$name? \in out$
$in' = in \cup \{name?\}$
$out' = out \setminus \{name?\}$
$users' = users$

We can extend this so that it gives a success message using schema conjunction:

Success

$reply! : CheckInReply$

$reply! = ok$

$$GoodCheckIn == CheckIn \land Success$$

The negation of the precondition of *CheckIn* is $name? \notin out$, that is, $name?$ is not in the set *out*. Now this could be because $name?$ is already in ($name? \in in$) or because $name?$ is not a recognised user ($name? \notin users$). We introduce two separate schemas to deal with these cases, giving appropriate error messages in each case:

*BadCheckIn*1

ΞLog
$name? : Staff$
$reply! : CheckInReply$

$name? \in in$
$reply! = already\_in$

*BadCheckIn*2

Ξ*Log*
name? : *Staff*
reply! : *CheckInReply*

---

name? $\notin$ *users*
reply! $=$ *not_registered*

We have assumed that *CheckInReply* has three values:

$$\mathit{CheckInReply} == \mathit{ok} \mid \mathit{already\_in} \mid \mathit{not\_registered}$$

Now the robust check-in operation is simply the disjunction of three schemas:

$$\begin{aligned} \mathit{RobustCheckIn} \quad == \quad & \mathit{GoodCheckIn} \\ & \lor \mathit{BadCheckIn}1 \\ & \lor \mathit{BadCheckIn}2 \end{aligned}$$

5 Relations

An ordered pair of values is written:

$$(x, y)$$

Cartesian Product is the type for ordered pairs, written:

$$T_1 \times T_2$$

Given $x : T_1, \quad y : T_2, \quad$ we have

$$(x, y) : T_1 \times T_2$$

What are the types of the following expressions?

$$(4, 7) : \ ?$$

$$(\{5, 6, 3\},\ 4) : \ ?$$

$$\{\ (4, 8),\ (3, 0),\ (2, 9)\ \} : \ ?$$

A *Relation* is simply a set of order pairs. For example, a database relating names to telephone numbers can be modelled as a set of ordered pairs:

$$\begin{aligned} \mathit{directory} \quad = \quad & \{\ (\mathit{mary}, 287573), \\ & (\mathit{mary}, 398620), \\ & (\mathit{john}, 829483), \\ & (\mathit{jim}, 493028), \\ & (\mathit{jane}, 493028)\ \} \end{aligned}$$

The set *directory* has type,

$$\mathit{directory} : \mathbb{P}(\mathit{Person} \times \mathit{Number})$$

Note that it is possible for a name to be related to more than one number (*mary* may have a home number and a mobile number) and it is possible for two people to be related to the same number (*jim* and *jane* may live together).

Because relations are commonly used in specification, they have their own special symbol:

$$T \leftrightarrow S \;==\; \mathbb{P}(T \times S)$$

So we can write:

$$\mathit{directory} : \mathit{Person} \leftrightarrow \mathit{Number}$$

Maplets An ordered pair (x, y) can also be written

$$x \mapsto y$$

This is perhaps more suggestive of relating one value (e.g., a name) to another (e.g., a number). So $\mathit{directory} : \mathit{Person} \leftrightarrow \mathit{Number}$ can be written

$$\begin{aligned} \mathit{directory} \;=\; & \{\, \mathit{mary} \mapsto 287573, \\ & \mathit{mary} \mapsto 398620, \\ & \mathit{john} \mapsto 829483, \\ & \mathit{jim} \mapsto 493028, \\ & \mathit{jane} \mapsto 493028 \,\} \end{aligned}$$

Note: $\leftrightarrow$ combines 2 types to form a type. $\mapsto$ combines 2 values to form an ordered pair.

Domain and Range For any relation, the set of all the first components of its maplets is called its *domain*. For example the domain of *directory* is:

$$\{\mathit{mary}, \mathit{john}, \mathit{jim}, \mathit{jane}\}$$

This is written $\mathrm{dom}(\mathit{directory})$. Even if *mike* is of type name, it is not in $\mathrm{dom}(\mathit{directory})$ since there is no maplet in *directory* whose first component is *mike*.

The set of all the second components of a relation's maplets is called its *range* (ran). We have:

$$\mathrm{ran}(\mathit{directory}) = \{287573, 398620, 829483, 493028\}$$

5.1 Phone Directory Spec

Using relations, we specify a phone directory which relates people to their phone numbers. We assume the following basic types:

$$[\mathit{Person}, \mathit{Phone}]$$

The state of the directory is given by the following schema:

Directory

$$\mathit{dir} : \mathit{Person} \leftrightarrow \mathit{Phone}$$

Initially the directory is empty:

InitDirectory

$$\mathit{Directory}$$

$$\mathit{dir} = \{\}$$

We add an entry to the directory with the following operation:

AddEntry

$$\Delta Directory$$
$$name? : Person$$
$$number? : Phone$$
$$dir' = dir \cup \{name? \mapsto number?\}$$

An operation to get all the numbers associated with a name is specified as:

GetNumbers

$$\Xi Directory$$
$$name? : Person$$
$$numbers! : \mathbb{P}\, Phone$$
$$numbers! = \{ n : Phone \mid (name? \mapsto n) \in dir \}$$

Should this operation be made robust?

Equally we could specify an operation to get the names associated with a number:

GetNames

$$\Xi Directory$$
$$number? : Phone$$
$$names! : \mathbb{P}\, Person$$
$$names! = \{ p : Person \mid (p \mapsto number?) \in dir \}$$

The *RemoveEntry* operation removes an entry from the directory:

RemoveEntry

$$\Delta Directory$$
$$name? : Person$$
$$number? : Phone$$
$$dir' = dir \setminus \{ name? \mapsto number? \}$$

5.2 Domain Subtraction

Suppose we wish to remove all the entries associated with a name. This may be achieved using the *domain subtraction* operation ($\lhd$).

$S \lhd R$ represents the relation R with all mappings for domain elements in the set S removed. For example, if

$$\begin{aligned} directory \;=\; & \{ mary \mapsto 287573, \\ & mary \mapsto 398620, \\ & john \mapsto 829483, \\ & jim \mapsto 493028, \\ & jane \mapsto 493028 \; \} \end{aligned}$$

Then

$$\begin{aligned}\{mary\} \lhd directory \quad = \quad & \{\, john \mapsto 829483, \\ & jim \mapsto 493028, \\ & jane \mapsto 493028 \,\}\end{aligned}$$

Note that $S \lhd R \;=\; \{\, x \mapsto y \mid (x \mapsto y) \in R \land x \notin S \,\}$.

What is $\{john\} \lhd directory$?

What is $\{emma\} \lhd directory$?

An operation to remove all entries associated with a name is now specified as:

RemoveName

$\Delta Directory$

$name? : Person$

$dir' = \{name?\} \lhd dir$

An operation to remove all entries associated with a set of names is specified as:

RemoveNames

$\Delta Directory$

$names? : \mathbb{P}\, Person$

$dir' = names? \lhd dir$

6 Partial Functions

A *partial function* is a special kind of relation in which each domain element has at most one range element associated with it. To declare f as a partial function we write:

$$f : X \pfun Y$$

This stands for:

$$\begin{aligned}& f : X \rel Y \mid \\ & \quad \forall\, a : X;\quad b_1, b_2 : Y \cdot \\ & \quad (a \mapsto b_1) \in f \land (a \mapsto b_2) \in f \;\;\Rightarrow\;\; b_1 = b_2\end{aligned}$$

As with more general relations, we an write $\mathrm{dom}(f)$ and $\mathrm{ran}(f)$.

6.1 Function Application

If $a \in \mathrm{dom}(f)$, then we write

$$f(a)$$

for the unique range element associated with a in f.

If $a \notin \mathrm{dom}(f)$, then $f(a)$ is undefined.

For example, suppose $dir1$ is defined as follows:

$$\begin{array}{lll} dir1 & = & \{\ mary \mapsto 398620, \\ & & \ john \mapsto 829483, \\ & & \ jim \mapsto 493028, \\ & & \ jane \mapsto 493028\ \} \end{array}$$

Clearly $dir1$ is of type $Person \pfun Phone$. We have that:

$$dir1(jim) = 493028$$
$$dir1(john) = 829483$$
$dir1(sarah)$ *is undefined*

Now suppose $dir2$ is defined as:

$$\begin{array}{lll} dir2 & = & \{\ mary \mapsto 287573, \\ & & \ mary \mapsto 398620, \\ & & \ john \mapsto 829483, \\ & & \ jane \mapsto 493028\ \} \end{array}$$

This time $dir2$ is not a partial function since *mary* is related to more than one number and $dir2(mary)$ is undefined.

6.2 Function Operators

The normal set and relation operators may be applied to partial functions. For example, set union may be used to extend a function:

$$dir1 \cup \{emma \mapsto 483928\}$$

Note: $f \cup g$ is a partial function provided

$$\forall x \cdot \quad x \in \mathrm{dom}(f) \land x \in \mathrm{dom}(g) \quad \Rightarrow \quad f(x) = g(x)$$

Why?

Domain subtraction may be used to remove entries from a partial function:

$$\begin{array}{lll} \{mary, john\} \domsub dir1 & = & \{\ jim \mapsto 493028, \\ & & \ jane \mapsto 493028\ \} \end{array}$$

Function Overriding is an operator only used on partial functions. It is used to replace an existing entry with a new one. $f \oplus \{x \mapsto y\}$ represents the function f with the entry for x replaced by $x \mapsto y$. For example:

$$\begin{array}{lll} dir1 \oplus \{jim \mapsto 567325\} & = & \{\ mary \mapsto 398620, \\ & & \ john \mapsto 829483, \\ & & \ jim \mapsto 567325, \\ & & \ jane \mapsto 493028\ \} \end{array}$$

$f \oplus \{x \mapsto y\}$ is the same as $(\{x\} \domsub f) \cup \{x \mapsto y\}$, so if x is not already in the domain of f, then the new entry is simply added and there is no previous entry to override.

7 Birthday Book

We use partial functions to specify a database for recording people's birthdays. We assume some basic types:

$$[Person, Date]$$

Each person is associated with at most one birthday in the state schema:

BirthdayBook

$$bb : Person \pfun Date$$

The database is initially empty:

InitBB

$$BirthdayBook$$

$$bb = \{\}$$

We add an entry to the birthday book as follows:

Add

$$\Delta BirthdayBook$$
$$name? : Person$$
$$date? : Date$$

$$name? \notin \mathrm{dom}(bb)$$
$$bb' = bb \cup \{ name? \mapsto date? \}$$

Note that this is only valid if *name*? doesn't already have an entry associated with it in the database.

An operation to update an entry in the birthday book is specified as:

Update

$$\Delta BirthdayBook$$
$$name? : Person$$
$$date? : Date$$

$$bb' = bb \oplus \{ name? \mapsto date? \}$$

An operation to remove an entry from the birthday book is specified as:

Remove

$$\Delta BirthdayBook$$
$$name? : Person$$

$$bb' = \{ name? \} \rhd bb$$

What happens if $name? \notin \mathrm{dom}(bb)$, in the previous two operations?

To lookup a persons birthday in the book we use function application:

Lookup

$\Xi BirthdayBook$

$name? : Person$

$date! : Date$

$name? \in \mathrm{dom}(bb)$

$date! = bb(name?)$

Here $bb(name?)$ gives the birthday associated with $name?$ and is only valid because bb is a partial function and because $name? \in \mathrm{dom}(bb)$.

If $name? \notin \mathrm{dom}(bb)$, then we have an error case:

BadLookup

$\Xi BirthdayBook$

$name? : Person$

$r! : LookupReply$

$name? \notin \mathrm{dom}(bb)$

$r! = notknown$

$$LookupReply == ok \mid notknown$$

To make a robust version of the lookup operation we extend the *Lookup* operation so that it gives a success message and disjoin this with *BadLookup*:

Success

$r! : LookupReply$

$r! = ok$

$$RobustLookup == (Lookup \land Success) \lor BadLookup$$

An operation to lookup the set of names whose birthday is on a particular date is specified as:

Who

$\Xi BirthdayBook$

$date? : Date$

$names! : \mathbb{P}\, Person$

$$names! = \{ p : Person \mid p \in \mathrm{dom}(bb) \land bb(p) = date? \}$$

8 Total Functions

A total function is a special case of a partial function that is defined for all possible values of its argument type. The declaration

$$f : X \to Y$$

stands for

$$f : X \pfun Y \mid \mathrm{dom}(f) = X$$

This says that f is a total function, i.e., $f(a)$ is well defined for each $a : X$.

The *square* function that returns the square of its argument is an example of a total function since it is well defined for all integers. In Z, such a function is defined using a so-called *axiomatic definition* as follows:

$$\begin{array}{|l} \mathit{square} : \mathbb{Z} \to \mathbb{Z} \\ \hline \forall\, n : \mathbb{Z} \bullet \\ \quad \mathit{square}(n) \;=\; n * n \end{array}$$

The function and its type is declared above the line and a predicate defining the function is declared below the line.

Functions can also be defined recursively. For example, the factorial function is defined as follows:

$$\begin{array}{|l} \mathit{factorial} : \mathbb{N} \to \mathbb{N} \\ \hline \forall\, i : \mathbb{N} \bullet \\ \quad \mathit{factorial}(0) \;=\; 1 \\ \quad \mathit{factorial}(i+1) \;=\; (i+1) * \mathit{factorial}(i) \end{array}$$

This form of definition can also be used to introduce a constant:

$$\begin{array}{|l} c : T \\ \hline \mathcal{A} \end{array}$$

This says that c is a constant of type T satisfying predicate $\mathcal{A}$.

For example:

$$\begin{array}{|l} \mathit{min\_count}, \mathit{max\_count} : \mathbb{N} \\ \hline \mathit{max\_count} \;=\; 100 \\ 10 \leq \mathit{min\_count} < \mathit{max\_count} \end{array}$$

9 Glossary of Symbols

Logic

$\land$	logical-and (conjunction)
$\lor$	logical-or (disjunction)
$\neg$	negation
$\Rightarrow$	logical implication
$(\exists x \bullet P)$	exists an x such that P
$(\forall x \bullet P)$	forall x, P holds

Sets

$\{\cdots\}$	set delimiters
$\{ x \mid P \}$	set of x satisfying P
$\in, \notin$	set membership, non-membership
$\cup, \cap$	set union, set intersection
$\setminus$	set subtraction
$\mathbb{P} S$	powerset of S
$\mathbb{Z}, \mathbb{N}$	set of integers, set of natural numbers
$S \subseteq T$	S is a subset of T
$S \times T$	cartesian product of S and T

Relations and Functions

$S \leftrightarrow T$	set of relations from S to T
$S \nrightarrow T$	set of partial functions from S to T
$S \rightarrow T$	set of total functions from S to T
$x \mapsto y$	mapping of element x to element y
$f(x)$	application of function f to element x
$\mathrm{dom}\, f, \mathrm{ran}\, f$	domain of f, range of f
$f \oplus g$	function f overridden by function g
$S \lhd R$	relation (or function) R with all elements in S removed from its domain