

System Specification in Z

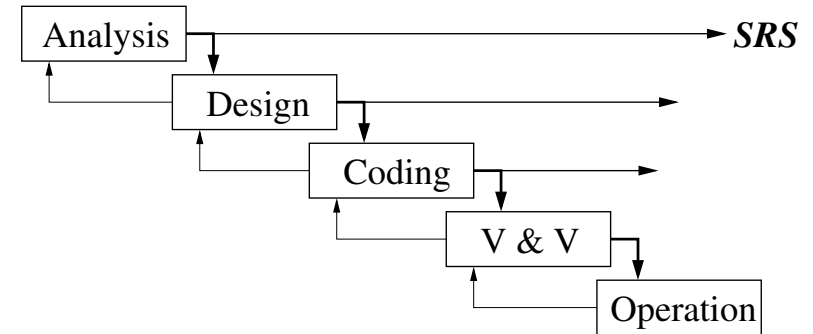
COMP2600 — Formal Methods for Software Engineering

Malcolm Newey

Australian National University
Semester 2, 2011

The Waterfall Model of System Development

- This is the simplest practical *System Development Life Cycle*.
- It assumes development is done in four phases .



- Note inputs to and outputs from the phases.

What is Z?

Z: A *mathematical language*, partly graphical, based on set theory and logic, that is used to formally specify **systems**.

- Note the use of the word **system**.
- The name comes from the first initial of Zermelo-Frankel set theory.
- Not a programming language but there are similarities.
- It is **declarative** and **strongly typed**.

The Role of Specification

- Requirements are specified in the System Requirements Specification
 - An SRS may be central to the contract with a software developer.
 - When validation fails and re-engineering is required, it can indicate who must bear the cost.
 - Clearly it must completely and unambiguously capture how the system is intended to perform.
- Designs also need to be specified – correctly, completely and unambiguously.
- The choice of language can make a difference.
- Programming languages can also be used for specification. They are formal notations.

Why be Formal in Specification?

Pros

- Precision of Mathematics
- Tool support
- Consistency may be checkable
- Completeness may be checkable

Cons

- Not customer-friendly
- Programmer/engineer training limited
- Phases of life cycle different

A short Z history

Inventor: Jean-Raymond Abrial - big system designer
(who since has gone on to develop the **B Method**.)

Evolution:

- 1980-1990 in Programming Research Group, Oxford
- Hayes, Morgan, Sufrin, Sorenson
- Australian Connection - UNSW, UQ
- Object Z

The Mismatch of Languages

When done informally,

- The Requirements Specification is English prose
- The Design is block diagrams, interface descriptions, module descriptions etc.
- The Code is in one or more programming languages
- The Manual is in English

This makes the use of tools that span the life cycle impossible.

.... and the hardest bit is coping with the natural language.

A Textbook List

- Hayes, *Specification Case Studies*, P-H, 1986
- Ince, *An Introduction to Discrete Mathematics and Formal System Specification*, OUP, 1988
- Woodcock & Loomes, *Software Engineering Mathematics*, Pitman, 1988
- Woodcock, *Using Z - ...*, PRG Oxford, 1990
- Imperato, *An Introduction to Z*, 1991
- Lightfoot, *Formal Specification Using Z*, McMillan, 1991
- Wordsworth, *Software Development in Z: ...*, A-W, 1993
- Spivey, *The Z Notation: A Reference Manual*, P-H, 1991
- Bowen, *Formal Specification and Documentation Using Z*, Thompson, 1995

Language Elements

- Some fragments look like declarations
- Other parts look like mathematical formulae
- Schemas (the building blocks of specifications) are usually presented graphically

Identifiers

- Basic identifiers are composed of letters, digits and underscores (starting with a letter)
- Case-sensitive
- Decorations - question mark, exclamation mark, prime

Given or Basic Types

Examples from a trip manager application

- Date and Time
- City and Country
- Flight and Airline
- Hotel
- Receipt

Even though the set of cities is not infinite it doesn't help to have it as an enumerated type.

Types

Built-in types

The set of the integers (\mathbb{Z}) is the only one!
(\mathbb{N} is then defined as a subset of \mathbb{Z}).

Given types

Given types are those that come from the system being modelled

Defined types

Sets defined by **enumeration**

Bool and Char are so defined

Reply = {yes, no}

Lights = {red, yellow, green}

Types defined by **abbreviation**

Data structures, functional types, relations are done in this way

Types in Z vs Types in a Prog. Lang.

In a programming language types are synthetic:

- Date would typically have fields for day, month, year.
- Receipt would have some structure to model the relevant attributes of receipts in the application.

In Z,

- What constitutes a date or a receipt is vague.
- What will be specified will be certain operations, not its structure or its representation.

Declarations and Decorations

Declarations

- Just as in a programming language
- Examples: boss: Person
 mine: Car

Decorations

- Names such as $v?$ are used for input variables;
- $v!$ is the form used for output variable names;
- v and v' give values of a variable before and after an operation

Set and Logic Notation in Z

Logic:

Standard operations: $\wedge \vee \neg \Rightarrow \equiv$

Set theory:

Basic operations: $\in \notin \subset \cup \cap \times \mathbb{P}$

Standard sets: $\mathbb{Z} \mathbb{N} \mathbb{R} \emptyset$

And also:

\rightarrow indicates a function

\leftrightarrow indicates a relation

$\#$ $\#S$ denotes the size of the set S . e.g. $\#\{2,4,6\}$ is 3.

Entities

Z focuses on 3 kinds of entities:

States The collection of values that constitute the mathematical structure that models the system at some point in time.

Events Occurrences, Operations, Transitions (wherein state changes)

Observations Examination of variables or aspects of the state before or after an event.

Example 1 - The Bus

This simple system is one where we have a bus of a certain capacity.

On the next slide we give, using some Z notation, the declarations of:

- the type *Person* appropriate for passengers
- the one global (state) variable – the set of people on the bus
- initial value of the state variables
- a crucial system invariant

Z declarations for *The Bus*

There is OneType:

$[Person]$

Global Declarations – a constant and a variable:

$busCapacity : \mathbb{N}$
 $passengers : \mathbb{P} Person$

Initial State of the one global variable:

$\#passengers = 0$

The System Invariant – true of every state:

$\#passengers \leq busCapacity$

The Operation of *boarding the bus*

Here are the assertions required to specify the operation of a person $p?$ boarding the bus:

Preconditions

$p? \notin passengers$
 $\#passengers < busCapacity$

that is: $p?$ cannot already be on the bus and there must be room.

Postcondition

$passengers' = passengers \cup \{p?\}$

that is: $p?$ has been added to the set of passengers

Specifying Operations for *The Bus*

On each of the next three slides a **collection of requirements**, expressed mathematically, specifies an operation in the system called *The Bus*.

1. A person boarding the bus (becoming a passenger)
2. A person leaving the bus
3. Asking if a person is on the bus

In each case **the operation is specified** by

- Precondition and Postcondition
- Declarations required for parameters of the operation

The Operation of *leaving the bus*

Here are the assertions required to specify the operation of a person alighting from the bus

Precondition

$p? \in passengers$

that is: $p?$ must initially be on the bus

Postcondition

$passengers' = passengers \setminus \{p?\}$

that is: $p?$ will have been removed from the set of passengers

Asking “Is that person on the bus”

Here are the assertions required to specify the operation of querying whether a person $p?$ is on the bus or not:

Preconditions - None

Postconditions

$$\begin{aligned} &(p? \in passengers \wedge reply! = yes) \\ &\vee (p? \notin passengers \wedge reply! = no) \\ &passengers' = passengers \end{aligned}$$

Note Declarations for the variables $p?$, $reply!$, $passengers'$ and the type $Reply$ are needed.

General Form of a Z document

1. Introduction in narrative form
2. Definitions of Types and Global Variables
3. Specification of the initial state
4. Schemas for operations and queries in the normal case
5. Schemas for error handling
6. Operations and queries with error handling

Is The Bus now specified in Z?

Not so far!

Although the preconditions and postconditions are formally expressed, the grouping of them into a structure is not formal.

There is still too much dependence on English and ad hoc layout.

We clearly need some structuring notation that packages all the preconditions and postconditions for each operation with appropriate declarations.

The basic building block in Z which does this is called the *Schema*.

Schemas

Z schemas are a rigorous shorthand and framework for the sort of specifications that we saw above.

- Schemas are quite formal
- They come in graphical and textual form with strict grammar
- They can be syntax checked and manipulated

This is what the graphical form looks like:



The Bus Example in Schema Form

<i>TheBus</i>
$passengers : \mathbb{P} Person$
$\#passengers \leq busCapacity$

<i>Initial</i>
<i>TheBus</i>
$passengers = \emptyset$

<i>TheBus'</i>
$passengers' : \mathbb{P} Person$
$\#passengers' \leq busCapacity$

Δ Schemas

The prefix Δ before a class name indicates it is concerned with states both before and after an **operation**.

Given schemas for *TheBus* and *TheBus'* above, the schema $\Delta TheBus$ is defined to be:

$\Delta TheBus$
<i>TheBus</i>
<i>TheBus'</i>

which is therefore equivalent to:

$\Delta TheBus$
$passengers : \mathbb{P} Person; passengers' : \mathbb{P} Person$
$\#passengers \leq busCapacity; \#passengers' \leq busCapacity$

'Importing' Schemas

In the schema *Initial* (on the last slide) the schema name (*TheBus*) appeared as one of the declarations.

The effect of having one schema name, *ThingA* say, in the declarations of another schema, *ThingB* say, is:

- to include all the declarations of *ThingA* among the declarations of *ThingB* and
- to include all the predicates of *ThingA* among the predicates of *ThingB* .

Operations on The Bus

<i>BoardBus_o</i>
$\Delta TheBus$
$p? : Person$
$p? \notin passengers$
$\#passengers < busCapacity$
$passengers' = passengers \cup \{p?\}$

<i>LeaveBus_o</i>
$\Delta TheBus$
$p? : Person$
$p? \in passengers$
$passengers' = passengers \setminus \{p?\}$

⊖ Schemas

The prefix \ominus before a class name indicates it can be used in the specifications of **enquiries** which are required to involve no state change.

Given schemas for $TheBus$ and $TheBus'$ above, the schema $\ominus TheBus$ is defined to be:

$\ominus TheBus$
$TheBus$
$TheBus'$
$passengers' = passengers$

The predicate part enforces the no-state-change characteristic.

Operations with Errors

If case of an error associated with an operation, some output value should say what it was. Typically the indication will be text – an error message.

If there are no errors the same result should indicate success.

$OkMessage$
$output! : PossibleMessages$
$output! = ok$

This schema relies on the prior type declaration:

$PossibleMessages$
 $::= ok \mid already_in_bus \mid full \mid not_in_bus \mid two_errors$

Queries about Bus Occupancy

$BusSize$
$\ominus TheBus$
$numberInBus! : \mathbb{N}$
$numberInBus! = \#passengers$

$PersonInBus$
$\ominus TheBus$
$response! : Reply$
$p? : Person$
$(p? \in passengers \wedge response! = yes)$
$\vee (p? \notin passengers \wedge response! = no)$

Operations with Errors – an Example

$BoardBusError$
$\ominus TheBus$
$p? : Person$
$output! : PossibleMessages$
$(p? \notin passengers \wedge \#passengers = busCapacity \wedge output! = full)$
$\vee (p? \in passengers \wedge \#passengers < busCapacity \wedge output! = already_in_bus)$
$\vee (p? \in passengers \wedge \#passengers = busCapacity \wedge output! = two_errors)$

Complete Specification – Combining the Usual and the Abnormal

The complete specification of adding a bus is the schema *BoardBus* which is defined in the non-graphical way:

$$BoardBus \hat{=} (BoardBus_o \wedge OkMessage) \vee BoardBusError$$

This illustrates:

- The **schema definition operator**, $\hat{=}$
- The fact that **schemas can be connected** by some logical operations - conjunction, disjunction and implication.
- We see now that the subscript on the name *BoardBus_o* indicates the error-free case of the operation.

Schema Expressions – Disjunction

If we have a schema *A* defined as:

$$A \hat{=} B \vee C$$

where *B* and *C* are previously defined schemas,

- The set of declarations of *A* is the union of the set of declarations of *B* and the set of declarations of *C*.
- The predicate part of *A* is $P \vee Q$ where:
 - *P* is the conjunction of the predicates in *B*,
 - *Q* is the conjunction of the predicates in *C*.

Schema Expressions – Conjunction

If we have a schema *A* defined as:

$$A \hat{=} B \wedge C$$

where *B* and *C* are previously defined schemas,

- Each declaration of *B* becomes a declaration in *A*.
- Each declaration of *C* becomes a declaration in *A*.
- The set of predicates of *A* is the union of the predicates in *B* and *C*.

The Complete *LeaveBus* Operation

$LeaveBusError$
$\exists TheBus$
$p? : Person$
$output! : PossibleMessages$
$p? \notin passengers \wedge output! = not_in_bus$

which allows us to define the final version of *LeaveBus* that copes with errors:

$$LeaveBus \hat{=} (LeaveBus_o \wedge OkMessage) \vee LeaveBusError$$

The convention of using a subscript *o* for the error free case is again illustrated.

General Form of a Z document (again)

1. Introduction in narrative form
2. Types and Global variables
3. The initial state
4. Operations and queries without error handling
5. Error handling
6. Operations and queries with error handling

Relations

The next example will illustrate the use of relations.

But, first some more Z notation:

- The notation $X \leftrightarrow Y$ indicates the type of all possible relations on X and Y .
- $X \leftrightarrow Y$ is the same type as $\mathbb{P}(X \times Y)$.
- The character \mapsto is called the *map symbol*.
- $x \mapsto y$ denotes the ordered pair (x, y) and called a *maplet*. A binary relation consists of a set of maplets.

Where to from Here?

Need to have more mathematics than basic set theory.

The text provide examples with more interesting mathematical models.

- Need **Relations**
- Need **Functions**
- Need **Sequences**

With these modelling techniques from discrete mathematics we can do a lot.

Functions – Recap.

- A **function** is a relation with the constraint that each element of its domain maps to at most one element of its range.
- A **total function** is one where *every* element of the domain is mapped to some element of its range.
When we write $f : X \rightarrow Y$ we mean to say that f is a total function.
- A **partial function** is not subject to that constraint. It is the general case of a function.
When we write $g : X \mapsto Y$ we mean to say that g is a partial function.
- **Examples:**
 - addition (on the integers) is a total function;
 - square root is partial.

Functions – Formal stuff

- A relation $f : X \leftrightarrow Y$ is a function if
 $\forall x : X \bullet \forall y_1, y_2 : Y \bullet (x \mapsto y_1) \in f \wedge (x \mapsto y_2) \in f \Rightarrow y_1 = y_2$
- The type $X \mapsto Y$ denotes the set of all *partial* functions from X to Y .
 Theorem: $(X \mapsto Y) \subset (X \leftrightarrow Y)$
- The type $X \rightarrow Y$ indicates the set of all *total* functions from X to Y .
 Theorem: $(X \rightarrow Y) \subset (X \mapsto Y) \subset (X \leftrightarrow Y)$
- If $f \in X \mapsto Y$ and $x \mapsto y \in f$ then we write $f(x)=y$

Note the use of the bit black ugly dot above. In Z we write $\forall x \bullet P(x)$ instead of the more traditional $\forall x.P(x)$.

Example 2 – Room Allocation

This example concerns the modelling of space allocation in a research building. We will refer to this system by the name **RoomAlloc**

The rooms (apart from service areas such as toilets) that make up the building are either offices (allocatable to either one or two staff members) or labs which are shared.

Further details will emerge as we see the Z specification unfold in:

- Type and constant declarations
- The state schema
- Operation and query schemas

Declaring (Constant) Functions

- In Example 1, we saw a constant, *busCapacity*, declared :

$$\left| \text{busCapacity} : \mathbb{N} \right.$$

- Constant functions can be declared in the same way. When so declared it is called an **axiomatic definition**.

For example, this is the way we would define the *factorial function* in a Z document:

$$\left| \begin{array}{l} \text{fact} : \mathbb{N} \rightarrow \mathbb{N} \\ \hline \forall n : \mathbb{N} \bullet \text{fact}(n) = (\text{if } n = 0 \text{ then } 1 \text{ else } n * \text{fact}(n - 1)) \end{array} \right.$$

Declarations of Types

- The set of uniquely identifiable people is a given type :

$$[Person]$$

- The set of uniquely identifiable rooms is a given type:

$$[Room]$$

- The sets of offices and labs are *constants* that we will use as types. They are declared axiomatically to be disjoint sets of rooms :

$$\left| \begin{array}{l} \text{Office} : \mathbb{P} \text{ Room} \\ \text{Lab} : \mathbb{P} \text{ Room} \\ \hline \text{Office} \cap \text{Lab} = \emptyset \end{array} \right.$$

Declarations of Constants

In the *RoomAlloc* system there are single and double offices. The function *capacity* gives the permitted number of occupants for an office. It is declared thus:

$$\mid \text{capacity} : \text{Office} \rightarrow \{1, 2\}$$

The use of the \rightarrow notation indicates that *capacity* is a total function and so:

- *capacity* is a set of maplets of the form $r \mapsto n$.
- It is defined for every office (either 1 or 2) and undefined for all other rooms.
- We could express the set of sharable offices as $\{x : \text{Office} \mid \text{capacity}(x) = 2\}$.

The RoomAlloc System's Initial State

<i>Initial</i>
<i>RoomAlloc</i>
$\text{staff} = \emptyset$
$\text{assignedTo} = \emptyset$

- In schemas that specify system initial state, initial values must be given for all state variables.
- The system variables *staff* and *assignedTo* become non-empty by operations affecting them.
- Note the system invariants are satisfied.

The State Schema for RoomAlloc

The following schema **characterises the state of the system**.
(Every system will have such a state schema.)

<i>RoomAlloc</i>
$\text{staff} : \mathbb{P} \text{Person}$
$\text{assignedTo} : \text{Room} \leftrightarrow \text{Person}$
$\text{dom } \text{assignedTo} \subseteq (\text{Office} \cup \text{Lab})$
$\text{ran } \text{assignedTo} \subseteq \text{staff}$

- *staff* is a set of people. *staff* is subject to change while *Person* is not.
- The keywords 'dom' and 'ran' denotes the domain and range operators;
- This system has two state variables and two invariants.

The Lab Assignment Operation

This schema specifies the error-free case of the operation of associating a lab with a staff member. There is no limit on how many staff can use the lab.

<i>AssignLab_o</i>
$\Delta \text{RoomAlloc}$
$p? : \text{Person}$
$r? : \text{Room}$
$p? \in \text{staff}$
$r? \in \text{Lab}$
$(r? \mapsto p?) \notin \text{assignedTo}$
$\text{staff}' = \text{staff}$
$\text{assignedTo}' = \text{assignedTo} \cup \{r? \mapsto p?\}$

Note the subscript in *AssignLab_o*.

Notes on the Lab Assignment Operation

- The declaration $\Delta RoomAlloc$ declares this to be an **operation**.
- It is an operation that takes two inputs – a person and a room.
- It is clear (from the absence of primed variables) that the first three predicates are **preconditions** for the operation. They must be satisfied in the error-free case.
- The last three predicates are clearly **postconditions**. (They involve primed variables.)
- Note that the post-condition $staff' = staff$ is **not optional**. It has to be present to specify that the set $staff$ is unchanged by the operation.

Notes on the Office Assignment Operation

- It is clear that the first four predicates are preconditions for the operation and the others are postconditions.
- The infix \triangleleft is the domain restriction operator. The term $S \triangleleft R$ restricts a relation R to contain just those maplets $x \mapsto y$ where $x \in S$.
- The term $\{r?\} \triangleleft assignedTo$ restricts the $assignedTo$ relation so that it contains just those maplets that involve $r?$. Thus $\#\{\{r?\} \triangleleft assignedTo\}$ gives the number of staff that are allocated to room $r?$.
- As before, the post-condition $staff' = staff$ is not optional.

The Office Assignment Operation

This schema specifies the error-free case of the operation of allocating office space to a staff member.

$AssignLab_o$ $\Delta RoomAlloc$ $p? : Person$ $r? : Room$
$p? \in staff$ $r? \in Office$ $(r? \mapsto p?) \notin assignedTo$ $\#\{\{r?\} \triangleleft assignedTo\} < capacity(r?)$ $staff' = staff$ $assignedTo' = assignedTo \cup \{r? \mapsto p?\}$

The Room Assignment Operation

Combining the operations defined above we can define the error-free case for room allocation.

$$RoomAlloc_o \hat{=} LabAlloc_o \vee OfficeAlloc_o$$

- Since the declarations for $LabAlloc_o$ and $OfficeAlloc_o$ match so they become the declarations for $RoomAlloc_o$.
- The predicate part of $RoomAlloc_o$ will be the disjunction of the predicate of $LabAlloc_o$ and $OfficeAlloc_o$.
- You can write it out and simplify it.

Room Assignment Errors - I

$\text{RoomAlreadyAssigned}$ $\exists \text{RoomAlloc}$ $p? : \text{Person}$ $r? : \text{Room}$ $\text{message!} : \text{PossibleMessages}$
$(r? \mapsto p?) \in \text{assignedTo}$ $\text{message!} = \text{this_assignment_exists}$

PossibleMessages applies to the whole system and is defined as:

$\text{PossibleMessages} ::= \text{okay} \mid \text{this_assignment_exists} \mid$
 $\text{not_on_staff} \mid \text{unallocatable_room} \mid$
 $\text{office_fully_occupied}$

Room Assignment Errors - III

OfficeFull $\exists \text{RoomAlloc}$ $r? : \text{Room}$ $\text{message!} : \text{PossibleMessages}$
$\#\{\{r?\} \triangleleft \text{assignedTo}\} = \text{capacity}(r?)$ $\text{message!} = \text{office_fully_occupied}$

Room Assignment Errors - II

NotStaff $\exists \text{RoomAlloc}$ $p? : \text{Person}$ $\text{message!} : \text{PossibleMessages}$
$p? \notin \text{staff}$ $\text{message!} = \text{not_on_staff}$

UnassignableRoom $\exists \text{RoomAlloc}$ $r? : \text{Room}$ $\text{message!} : \text{PossibleMessages}$
$(r? \notin \text{Lab}) \wedge (r? \notin \text{Office})$ $\text{message!} = \text{unallocatable_room}$

The Room Assignment with Error-handling

OK $\text{message!} : \text{PossibleMessages}$
$\text{message!} = \text{okay}$

$\text{AssignRoom} \hat{=} (\text{AssignRoom}_o \wedge \text{OK}) \vee$
 $\text{RoomAlreadyAssigned} \vee \text{NotStaff} \vee$
 $\text{UnassignableRoom} \vee \text{OfficeFull}$

Removing One Allocation

$Deallocate_o$ $\Delta RoomAlloc$ $p? : Person$ $r? : Room$
$p? \in staff$ $(r? \mapsto p?) \in assignedTo$ $staff' = staff$ $assignedTo' = assignedTo \setminus \{r? \mapsto p?\}$

Remember: $X \triangleleft R$ indicates relation R restricted to domain X .

Error specification for $Deallocate_o$ left as an exercise.

Staff Comings and Goings – II

When a staff member leaves, the *staff* list is also updated but by the *RetireStaff* operation.

$RetireStaff_o$ $\Delta RoomAlloc$ $p? : Person$
$p? \in staff$ $staff' = staff \setminus \{p?\}$ $\neg \exists r : Room \bullet (r \mapsto p?) \in assignedTo'$ $\forall r : Room, q : Person \bullet q \neq p? \Rightarrow$ $((r \mapsto q) \in assignedTo') = ((r \mapsto q) \in assignedTo)$

The third and fourth predicates specify that there are no allocations in the final state to person $p?$ but allocations to all other people remain the same.

Staff Comings and Goings

Recall that initially the set of staff members is empty. The variable *staff* is updated by the operation *AddStaff* which is specified as follows:

$AddStaff_o$ $\Delta RoomAlloc$ $p? : Person$
$p? \notin staff$ $staff' = staff \cup \{p?\}$ $assignedTo' = assignedTo$

The room assignments for a new staff member is done by the *Allocate* operation specified already.

A Slicker Specification for *RetireStaff*_o

The following schema, equivalent to the last, uses relation range restriction to delete all the allocations of rooms to $p?$.

$RetireStaff_o$ $\Delta RoomAlloc$ $p? : Person$
$p? \in staff$ $staff' = staff \setminus \{p?\}$ $assignedTo' = assignedTo \setminus (assignedTo \triangleright \{p?\})$

The term $R \triangleright S$ restricts a relation R to contain just those maplets $x \mapsto y$ where $y \in S$.

Exercise: Why can't we replace the second postcondition by

$$\forall r : Room \bullet assignedTo' = assignedTo \setminus \{r \mapsto p?\}$$

Query: Find Staff in an Office

$OfficeOccupants_o$ $\exists RoomAlloc$ $r? : Room$ $occupants! : \mathbb{P} Person$
$r? \in Office$ $occupants! = assignedTo (\{r?\})$

Notation: If $R : X \leftrightarrow Y$ is a relation and $S : \mathbb{P} X$ is a set, then $R (\{S\})$ is the subset of the range of R that is the image of S under R .

Query: Find Rooms

$FindRooms_o$ $\exists RoomAlloc$ $p? : Person$ $numbers! : \mathbb{P} Room$
$p? \in \text{ran } assignedTo$ $numbers! = assignedTo^{-1} (\{p?\})$

Notation: $assignedTo^{-1}$ is the inverse relation of $assignedTo$.
 $(n \mapsto s)$ is in $assignedTo^{-1}$ iff $(s \mapsto n)$ is in $assignedTo$.

Query: Find Staff Without an Office

$StaffWithoutOffice_o$ $\exists RoomAlloc$ $sList! : \mathbb{P} Person$
$sList! = \{s : Person \mid s \in staff \wedge$ $\#((Office \triangleleft assignedTo) \triangleright \{s\}) = 0\}$

Help!!

The term $((Office \triangleleft assignedTo) \triangleright \{s\})$ uses both domain and range restriction of the allocation relation, to give the set of allocations of any office to person s

Now Where?

All the textbooks provide examples with more interesting mathematical models.

- **Sequences** – we know about these!
- **Bags** – also called **Multisets**

A **bag** is a collection of objects where order doesn't matter (as with sets) but multiplicity does (unlike sets).

For example (using Z syntax):

$[[2, 3, 3, 4]]$ is the same bag as $[[3, 2, 4, 3]]$, but not the same as $[[2, 3, 4]]$.

Sequences

- $\langle 1, 4, 9, 16, 25 \rangle$ is an example, using Z syntax, of a **sequence**.
The **type** of this sequence is $\text{seq } \mathbb{N}$.
- Sequences in Z may be finite or infinite.
- $\text{seq } T$ is actually **syntactic sugar** for $\mathbb{N}_1 \rightarrow T$.
- So $\langle 1, 4, 9, 16, 25, 36 \rangle$ is syntactic sugar for
 $\{1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 9, 4 \mapsto 16, 5 \mapsto 25, 6 \mapsto 36\}$.
- Because a sequence is really a function, we can index a sequence using function application; for example $\langle 1, 4, 9, 16, 25, 36 \rangle(4) = 16$.

Summary

Z is a practical technique for formally modelling systems

- It uses powerful mathematics to express properties and so is rigorous.
- The schema notation is a good aid to the reader.
- It's focus on preconditions and postconditions suits specification of operations that change the state of a system.
- It's use is suggested in standards for safety-critical system design.
- It provides a basis for formal verification and model-directed development.
- There are lots of case studies, big and small, in the literature.

Bags

- The type 'bag T ' is the set of all bags of objects of type T .
- $[[]]$ denotes the empty bag.
- $[[1, 2, 2, 2, 3, 3, 4]]$ is the bag containing the numbers 1, 2, 3 and 4 with multiplicities 1, 3, 2 and 1, respectively.
- $[[1, 2, 2, 2, 3, 3, 4]]$ is the same bag as $[[4, 2, 2, 3, 3, 2, 1]]$.
- The type $\text{bag } T$ is actually syntactic sugar for the type $T \rightarrow \mathbb{N}_1$. As a consequence, we can get the multiplicity of element x in bag B by writing $B(x)$ (provided x is an element of B).
- The expression $x \in \text{dom } B$ says whether x is an element of B or not.