ANSWERS

Chapter 1

Ex 1.1 the collection of books in my study, the collection of software packages installed on my computer, the collection of cars that I own. You may have equally valid but different answers. The important point here is that each item in a collection has similar properties to each other item in that collection i.e. Z sets are typed sets.

Ex 1.2

1 { *Mon, Tue, Wed, Thu, Fri* }. You may be in college on different days or even on no days at all.
2 { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }. The order in which you write the elements is not important.
3 { 0, 1 }
4 { *red, orange, yellow, green, blue, indigo, violet* }. Some of us may remember the mnemonic Richard Of York Gave Battle In Vain where the first letter of each word represents a colour.
5 { *sales, accounting, production, personnel, management* }. You may think of other equally valid departments.

Ex 1.3

1 *BOOKTITLE* - ok - singular, capitals, no spaces, underscores or hyphens
2 *BOOK TITLE* - not ok - space not allowed
3 *BookTitle* - not ok - should be all capitals
4 *BOOK_TITLE* - not ok - underscore not allowed
5 *BOOKTITLES* - not ok - must be singular

Ex 1.4

1 $\mathbb{Z}$ - a notation that uses mathematics to describe computer systems, $\mathbb{Z}$- the set of all integers
2 $\varnothing$ and { } - no difference, both represent the empty set.

Ex 1.5

1 0..5 Note: just two dots.
2 0..23
3 0..125 125 is an arbitrary choice - you may have chosen an equally valid but different value
4 5..5 You may have chosen different values - the important point here is that both values are the same.
5 5..4 You may have chosen different values - the important point here is the first value is more than the second.

Ex 1.6

1 5
2 4
3 6
4 0
5 *undefined*

Ex 1.7

1 true - *end* is an element of { *if, else, while, repeat, until, end* }
2 false - *Key* is not an element of { *spaceKey, tabKey, returnKey, arrowKey, functionKey* }
3 true - the empty set has no elements, not even 0.
4 false - $\mathbb{Z}$ is the set of all integers and 3.142 is not an integer.
5 false - $\mathbb{N}$ is the set of all positive integers including zero.

Ex 1.8

1 false - the two sets do not contain exactly the same elements.
2 false - order does not matter.
3 true - order does not matter.
4 false - repeated elements are ignored.
5 true - order does not matter.

Chapter 2

Ex 2.1

1 abstraction - a simplification of reality deferring detail to a later stage. We may talk of STUDENT and ignore their name, date of birth, gender, course and location of study.
2 [*SUBJECT, TITLE*]
SUBJECT is the set of all possible subjects and *TITLE* is the set of all possible titles of books in libraries.
3 [*ROOMID*]
ROOMID the set of all possible hotel room names and numbers.
4 [*COURSE, SUBJECT*]
COURSE is the set of all courses and *SUBJECT* is the set of all possible subjects provided by all colleges, schools and universities.
5 [*ADDRESS, OWNER, REGISTRATIONMARK*]
REGISTRATIONMARK is the set of all vehicle registration marks, *OWNER* the set of all vehicle owners and *ADDRESS* is the set of all possible postal addresses.

Ex 2.2

1 *annualSalary* : $\mathbb{Z}$
2 *temperature* : $\mathbb{Z}$
3 We introduce CHARACTER, the set of all printable characters from all keyboards in all countries:
[*CHARACTER*]
Then:
printable : CHARACTER
4 We introduce CATALOGUENUMBER the set of all possible catalogue numbers and STATIONERYITEM the set of all possible stationery items:
[*CATALOGUENUMBER, STATIONERYITEM*]
Then:
item : *STATIONERYITEM*
number : *CATALOGUENUMBER*
5 *position* : 1..100

Ex 2.3

1 *numberOfPersons* : 0..5 ok
2 *NumberOfPersons* : 0..5 Not ok - must start with a lower case letter
3 *number Of Persons* : 0..5 Not ok - spaces not allowed
4 *NUMBEROFPERSONS* : 0..5 Not ok - must be written in lower case except for the first letter of each word (except the first word)
5 *n* : 0..5 Possible not ok - not descriptive

Ex 2.4

1 Not valid - *red, blue, green* are colours, 1, 2 and 3 are integers
2 Not valid - *ch* is a character and not an integer
3 Valid - *capacity* is an integer and so is 5.

Chapter 3

Ex 3.1

1a 16 div 3 = 5 **b** 20 div 4 = 5 **c** 5 div 5 = 1 **d** 4 div 5 = 0 **e** 7 div 5 = 1
2a 16 mod 3 = 1 **b** 20 mod 4 = 0 **c** 5 mod 5 = 0 **d** 4 mod 5 = 4 **e** 7 mod 5 = 2

Ex 3.2

1	5 + 7 * 9	=	5 + 63	[multiplication before addition]
		=	68	[arithmetic addition]
2	(5 + 7) * 9	=	12 * 9	[brackets first]
		=	108	[arithmetic multiplication]
3	5 div 9 * (212 - 32)	=	5 div 9 * 180	[brackets first]
		=	0 * 180	[integer division]
		=	0	[arithmetic multiplication]

4 *n* div 2 = 0 means *n* must be zero or 1.

n	n div 2	n div 2 = 0
0	0	true
1	0	true
2	1	false
3	1	false
4	2	false
5	2	false

5 (*n* mod 5) ∈ 0..4

n	n mod 5
0	0
1	1
2	2
3	3
4	4
5	0

Ex 3.3
1 7
2 -1
3 1
4 undefined or indeterminate

Ex 3.4
1a false **b** true **c** true
2a true **b** false **c** false
3 $n \in 0..4$
4 $n \in 10..19$
5 $n < 0$ OR $n > 9$ i.e. $n \notin 0..9$

Chapter 4

Ex 4.1
1 { 7 }
2 { 1, 2, 3 }
3 { 0, 1, 2, 3, 4, 5 }

Ex 4.2
1 { -1, 0, 1 }
2 { 1, 2, 3, 4, 5, 6, 7 }
3 { 0, 1 }
4 { 2, 4, 6, 8 }
5 { 1, 2, 3, 4 }

Chapter 5

Ex 5.1
1a empty - number of ten pound notes = 0,
1b full - machine cannot hold any more ten-pound notes
1c neither empty nor full

Ex 5.2
$\mathbb{N}_1 = = \{ n : \mathbb{Z} \mid n > 0 \}$

Ex 5.4

1 *Cancel* undoes a *Click* operation. We require that *count* is more than zero.

Cancel
$\Delta Counter$

$count > 0$
$count' = count - 1$

2 *Reset* sets count to zero.

Reset

$$\Delta Counter$$

$$count' = 0$$

3 The Car Park

We introduce capacity, the maximum number of cars that can be parked.

$$capacity : \mathbb{Z}$$

$$capacity \geqslant 0$$

The number of cars that can be parked is between zero and capacity inclusive.

CarPark

$$cars : \mathbb{Z}$$

$$cars \geqslant 0$$
$$cars \leqslant capacity$$

Initially, there are no cars in the car park.

InitCarPark

$$CarPark$$

$$cars = 0$$

A car may enter the car park provided its capacity has not been reached.

Enter

$$\Delta CarPark$$

$$cars < capacity$$
$$cars' = cars + 1$$

A car may depart from the car park provided there is at least one car in the car park.

Depart

$\Delta CarPark$

$cars > 0$

$cars' = cars - 1$

The number of available spaces is the difference between the car park capacity and the number of cars parked there.

QuerySpaces

$\Xi CarPark$

$spaces! : \mathbb{Z}$

$spaces! = capacity - cars$

Chapter 6

Ex 6.1

1 true **2** false - Key not an element **3** false - equality not allowed
4 true **5** true

Ex 6.2

1 { { }, { 0 }, { 1 }, { 0, 1 } }
2 Set A is a subset of set B if every element in A is also in B. With a proper subset equality is not allowed.
3a *a* is an element of { *a, b, c* }
b { *a* } is a subset of { *a, b, c* }
c { *a* } is an element of $\mathbb{F}$ { *a, b, c* }

Ex 6.3

1 $\in$ - is a member of, $\subseteq$ is a subset of
2 *p* - a single person, *q* a set of persons
3a *hotel* : *HOTEL* **b** *rooms* : $\mathbb{F}$*ROOM* **c** *reservation* : *RESERVATION*
d *guests* : $\mathbb{F}$*GUEST* **e** *date* : *DATE*

Chapter 7

Ex 7.1

1 { 1, 2, 3, 5, 8, 13, 7, 11 } Order does not matter
2 { *bigJ, littleJ, may, pat, alice, tom, denise* }
3 { *daisy, buttercup, rani* }
4 undefined
5 { *daisy, buttercup* }

Ex 7.2

1 { 3, 5, 13 } **2** { *may* } **3** { } **4** undefined
5 { *daisy, buttercup* }

Ex 7.3

1 { 1, 2, 8 } **2** { *bigJ, littleJ, pat, alice* } **3** { *daisy, buttercup* }
4 undefined **5** { } or ∅

Ex 7.4

1 $\{ 1, 2, 3 \} \cup \{ 2, 3, 4 \} \cap \{ 3, 4, 5 \} = \{ 1, 2, 3 \} \cup \{ 3, 4 \}$ [intersection first]
$= \{ 1, 2, 3, 4 \}$ [union]

2 $(\{ 1, 2, 3 \} \cup \{ 2, 3, 4 \}) \cap \{ 3, 4, 5 \} = \{ 1, 2, 3, 4 \} \cap \{ 3, 4, 5 \}$ [brackets first]
$= \{ 3, 4 \}$ [intersection]

3 $\{ 1, 2, 3 \} \cup \{ 2, 3, 4 \} \setminus \{ 3, 4, 5 \} = \{ 1, 2, 3, 4 \} \setminus \{ 3, 4, 5 \}$ [union first]
$= \{ 1, 2 \}$ [difference]

4 $\{ 1, 2, 3 \} \setminus \{ 2, 3, 4 \} \cup \{ 3, 4, 5 \} = \{ 1 \} \cup \{ 3, 4, 5 \}$ [difference first]
$= \{ 1, 3, 4, 5 \}$ [union]

5 $\{ 2, 3, 4 \} \setminus \{ 3, 4, 5 \} \cap \{ 1, 2, 3 \} = \{ 2, 3, 4 \} \setminus \{ 4, 5 \}$ [intersection first]
$= \{ 2, 3, 4 \}$ [difference]

Chapter 8

Ex 1

1

[*STUDENT*]	the set of all possible students everywhere
$maxClassSize : \mathbb{Z}$	*maxClassSize* is an integer - a whole number
$maxClassSize = 20$	*maxClassSize* is set to 20

Class	the system state schema is named *Class*
$enrolled : \mathbb{F}\ STUDENT$	*enrolled* is a finite set of *STUDENT* objects
$passed : \mathbb{F}\ STUDENT$	
$\#enrolled \leqslant maxClassSize$	number in enrolled cannot exceed *maxClassSize*
$passed \subseteq enrolled$	all those who have passed must also be enrolled

InitClass

Class	*InitClass* represents the beginning includes everything defined above in Class
enrolled = ∅	there are no students
passed = ∅	

Enrol

Δ*Class*	contents of enrolled may change
student? : *STUDENT*	the input is a student
#enrolled < *maxClassSize*	there must be enough space for the new student
student? ∉ *enrolled*	the new student cannot already be enrolled
enrolled' = *enrolled* ∪ { *student*? }	the new student is enrolled
passed' = *passed*	passed list remains unchanged

Complete

Δ*Class*	
student? : *STUDENT*	
student? ∈ *enrolled*	the student input must be enrolled and
student? ∉ *passed*	not already passed
passed' = *passed* ∪ *student*?	include the student input in the passed list
enrolled' = *enrolled*	

LeaveWithCertificate

Δ*Class*	
student? : *STUDENT*	
student? ∈ *passed*	the student input must have passed
passed' = *passed* \ { *student*? }	remove the student input from the passed list
enrolled' = *enrolled* \ { *student*? }	and from the enrolled list

2 The student has not passed and is removed from enrolled.

LeaveWithoutCertificate

Δ*Class*	
student? : *STUDENT*	
student? ∉ *passed*	
enrolled' = *enrolled* \ *student*?	

3 The number of enrolled students is output.

ReportNumberEnrolled

$\Xi Class$
$number? : \mathbb{Z}$

$number? = \#enrolled$

Ex 8.2

1a The pre-conditions for success: the student is enrolled and has not completed all the assignments.

b The conditions for failure: the student has not enrolled or the student has completed all their assignments.

2 alreadyPassed, notEnrolled

Ex 8.3

Success — the schema is named Success

$report! : REPORT$ — the report output can have any REPORT value

---

$report! = ok$ — the report output is ok

ClassFull

$\Xi Class$ — enrolled and passed lists remain unchanged
$report! : REPORT$

$\#enrolled \geqslant maxClassSize$ — enrolled students has reached *maxClassSize*
$report! = classFull$

AlreadyEnrolled

$\Xi Class$
$student? : STUDENT$ — the student input is ...
$report! : REPORT$

$student? \in enrolled$ — ... already enrolled
$report! = alreadyEnrolled$

NotEnrolled

$\Xi Class$
student? : *STUDENT*
report! : *REPORT*

---

student? $\notin$ *enrolled* — the student input is not enrolled
report! = *notEnrolled*

AlreadyPassed

$\Xi Class$
student? : *STUDENT*
report! : *REPORT*

---

student? $\in$ *passed* — the student input has already passed
report! = *alreadyPassed*

NotPassed

$\Xi Class$
student? : *STUDENT*
report! : *REPORT*

---

student? $\notin$ *passed* — the student input has not passed
report! = *notPassed*

Ex 8.4

1 Leave without certificate total is defined to be leave without certificate and success, or student not enrolled or student already passed.

$$LeaveWithoutCertificateTot \mathrel{\widehat=} (LeaveWithoutCertificate \land Success) \lor NotEnrolled \lor AlreadyPassed$$

2 We introduce ROOM, the set of all possible hotel rooms.

[*ROOM*]

The hotel has a set of rooms named accommodation; guests may occupy some (or all) of these rooms.

Hotel

accommodation, occupied : $\mathbb{F}$ *ROOM*

$occupied \subseteq accommodation$

In the beginning there are no rooms.

InitHotel

Hotel

$occupied = \varnothing$

$accommodation = \varnothing$

We add a new room to the accommodation system provided that has not already been done.

Commission

$\Delta Hotel$

$room? : ROOM$

$room? \notin accommodation$

$accommodation' = accommodation \cup \{ room? \}$

A guest occupies a room provided the room is not already occupied.

Occupy

$\Delta Hotel$

$room? : ROOM$

$room? \notin occupied$

$occupied' = occupied \cup \{ room? \}$

$accommodation' = accommodation$

A room is vacated provided it is occupied in the first place.

Vacate

$\Delta Hotel$

$room? : ROOM$

$room? \in occupied$

$occupied' = occupied \setminus \{ room? \}$

$accommodation' = accommodation$

We count the number of occupied rooms.

QueryOccupiedRooms

$\Xi Hotel$

$number! : \mathbb{Z}$

$number! = \#occupied$

We remove a room from the system provided it exists and is not occupied.

Decommission

$\Delta Hotel$

$room? : ROOM$

$room? \notin occupied$

$room? \in accommodation$

$accommodation' = accommodation \setminus \{ room? \}$

We define REPORT to consist of values that report on the success or failure of our methods.

$$REPORT ::= ok \mid alreadyAdded \mid notInSystem \mid roomOccupied \mid alreadyVacant$$

ok indicates success.

Success

$report! : REPORT$

$report! = ok$

We cannot add the same room twice to the system.

RoomAlreadyAdded

$\Xi Hotel$

$room? : ROOM$

$report! : REPORT$

$room? \in accommodation$

$report! = alreadyAdded$

We cannot put a guest in a room, or decommission a room, if the room is not in the system.

RoomNotInSystem

$\Xi Hotel$
$room? : ROOM$
$report! : REPORT$

$room? \notin accommodation$
$report! = notInSystem$

We cannot put a guest in a room, or remove the room from the system, if it is currently occupied.

RoomOccupied

$\Xi Hotel$
$room? : ROOM$
$report! : REPORT$

$room? \in occupied$
$report! = roomOccupied$

We cannot vacate a room if it is already vacant.

RoomAlreadyVacant

$\Xi Hotel$
$room? : ROOM$
$report! : REPORT$

$room? \in accommodation \setminus occupied$
$report! = alreadyVacant$

Now we can define the total commission process to be *Commission* and *Success*, or *RoomAlreadyAdded*.

$$CommissionTot \mathrel{\widehat=} (Commission \land Success) \lor RoomAlreadyAdded$$

We define the complete occupy process to be *Occupy* and *Success*, or *RoomNotInSystem*, or *RoomOccupied*.

$$OccupyTot \mathrel{\widehat=} (Occupy \land Success) \lor RoomNotInSystem \lor RoomOccupied$$

We define the complete vacate process to be *Vacate* and *Success*, or *RoomNotInSystem*, or *RoomAlreadyVacant*.

$$VacateTot \mathrel{\widehat=} (Vacate \land Success) \lor RoomNotInSystem \lor RoomAlreadyVacant$$

We define the complete decommission process to be *Decommission* and *Success*, or *RoomNotInSystem*, or *RoomOccupied*.

$DecommissionTot \triangleq (Decommission \land Success) \lor RoomNotInSystem \lor RoomOccupied$

Chapter 9

Ex 9.1

1 owners and their motor vehicles, managers and their football teams, doctors and their patients, website developers and their web sites, inventors and their inventions, debtors and their debts, ... You may well think of other equally valid binary relations.

2a { (*a, x*), (*b, y*), (*a, z*), (*b, w*) }
b { (*a, x*), (*b, y*) }
c { (b, y) }
d { (b, y) }
e 4

Ex 9.2

1 (*first*(*a*, 1), *second*(*a*, 1)) = (*a*, 1)

Ex 9.3

1a { *A*301, *A*302, *A*303, *A*304 }
b { *office, classRoom, lab* }
c { *terry, garry, kanti* }
d { *Access, ECDL, ALevel, HND* }
e { *richard, richardII, richardIII* }
f { 1921, 1952, 1976 }

Ex 9.4

1a debtors : CUSTOMER $\leftrightarrow \mathbb{Z}$ e.g (marris, 25000)
b instructs : INSTRUCTOR $\leftrightarrow$ PUPIL e.g. (michael, kimmi)
c ownes : OWNER $\leftrightarrow$ HORSE e.g. (queen, zebedee)
d studies : STUDENT $\leftrightarrow$ SUBJECT e.g. (marris, Z)
e hasWritten : AUTHOR $\leftrightarrow$ TITLE e.g. (marris, divingIntoC)

Ex 9.5

1a { $A \mapsto carrots$, $C \mapsto potatoes$, $C \mapsto tomatoes$ }
b { $green \mapsto 3$, $yellow \mapsto 4$, $blue \mapsto 5$ }
c { }
d { $rook \mapsto 5$, $queen \mapsto 9$ }

Ex 9.6

1a { *tom*, *jerry* }
b { *A*, *C*, *E* }
c { }
d { 12:20 }

e { 2, 3, 5, 7, 11 }

2 { 1 ↦ *a*, 2 ↦ *b*, 3 ↦ *c*, 4 ↦ *d*, 5 ↦ *e* } (2..4) = { *b*, *c*, *d* }
ran (2..4 ◁ { 1 ↦ *a*, 2 ↦ *b*, 3 ↦ *c*, 4 ↦ *d*, 5 ↦ *e* }) = ran { 2 ↦ *b*, 3 ↦ *c*, 4 ↦ *d* }
= { b, c, d }

They are identical.

Ex 9.7

1a { *tea* ↦ 55, *coffee* ↦ 75, *hotChocolate* ↦ 75 }
b { *smith* ↦ *beer*, *kline* ↦ *cider*, *french* ↦ *win*, *jack* ↦ *whisky* }
c { *andy* ↦ *mon*, *bettie* ↦ *tue*, *carol* ↦ *thu*, *fred* ↦ *wed* }

Ex 9.8

1a *alias*~ = { *dopy* ↦ *tom*, *dozy* ↦ *sam*, *dreamy* ↦ *stu*, *sleepy* ↦ *tom*, *titch* ↦ *sam* }
b { *sam* ↦ *dozy*, *stu* ↦ *dreamy* }
c { *dopy*, *dozy*, *sleepy*, *titch* }
d { }
e { *sam*, *stu* }

Ex 9.9

1 *riddenBy*~ = { *jones* ↦ *merryTom*, *jan* ↦ *ticTak*, *french* ↦ *jumpingJack*, *jan* ↦ *isAGas*, *fraser* ↦ *hissingSid* }
entered~ = { *merryTom* ↦ *sam*, *jumpingJack* ↦ *sam*, *hissingSid* ↦ *pam*, *ticTak* ↦ *jan*, *isAGas* ↦ *tel* }
riddenBy~ ⨾ *entered*~ = { *jones* ↦ *sam*, *jan* ↦ *jan*, *french* ↦ *sam*, *jan* ↦ *tel*, *fraser* ↦ *pam* }

Chapter 10

Ex 10.1

1 *URL* → *WEBSITE* because each URL (uniform resource locator e.g. www.tmarris.com) maps to just one unique web site - no two different web sites have the same URL.

ISBN → *BOOK* because each ISBN (International Standard Book Number) uniquely identifies each book - no two books have the same ISBN.

PAYROLL-NUMBER → (COMPANY ↔ EMPLOYEE) because no two employees in the same company have the same payroll number.

You may well have thought of different examples of functions, each equaly valid.

Ex 10.2

1a { 999, 11, 100, 155 }
b { *free*, *3pPerMin*, *8pPerMin* }

c { 064 ↦ *newZealand*, 061 ↦ *australia* }
d { *blair* ↦ *uk*, *howard* ↦ *australia* }
e { 3 ↦ *wed* }
f { *tom* ↦ 25, *dee* ↦ 25, *harry* ↦ 24, *may* ↦ 23 }
g { *A* ↦ 80, *B* ↦ 40, *C* ↦ 0, }
h { *australia* ↦ 061, *india* ↦ 091, *newZealand* ↦ 064, *uk* ↦ 044 }
i { }
j { 101 ↦ *UsingZ*, 103 ↦ *ZInPractice*, 107 ↦ *ZWays* }
k {101 ↦ *UsingZ*, 103 ↦ *ZInPractice*, 107 ↦ *ZWays*, 105 ↦ *ZNotes* }

Ex 10.3

1a *branch* : *SORT-CODE* ⇸ *BANK-BRANCH*

---

branch = { 112233 ↦ *floydsOadby*, 445566 ↦ *barcWestBath*, 778899 ↦ *bshKeld* }

b *hasMother* : *PERSON* ⇸ *PERSON*

---

hasMother = { *bart* ↦ *marge*, *lisa* ↦ *marge*, *maggie* ↦ *marge* }

c *copy* : *ACCESSION-NUMBER* ⇸ *COPY*

---

copy : { *088079* ↦ *mottAndRendelDatabaseProjects* }

Ex 10.4

1a *format*
b undefined

2a { *steve* ↦ 23, *chris* ↦ 22, *sonu* ↦ 24 }
b *age(sam)* is undefined

3a 5
b { 1 ↦ *occupied*, 2 ↦ *vacant*, 3 ↦ *occupied*, 4 ↦ *empty*, 5 ↦ *vacant*, 6 ↦ *occupied* }
c { 1 ↦ *occupied*, 2 ↦ *occupied*, 3 ↦ *occupied*, 4 ↦ *empty*, 5 ↦ *vacant*, 6 ↦ *occupied* }
d { 1 ↦ *vacant*, 2 ↦ *vacant*, 3 ↦ *occupied*, 4 ↦ *empty*, 5 ↦ *vacant*, 6 ↦ *occupied* }
e *vacant*
f { 1 ↦ *vacant*, 2 ↦ *vacant*, 3 ↦ *occupied*, 4 ↦ *empty*, 6 ↦ *occupied* }

Ex 10.5

1 [*ROOM*] — a basic type introduced into the specification

STATUS ::= *occupied* | *vacant* — a free type with values *occupied* and *vacant*

Hotel

$room : ROOM \pfun STATUS$

the system state schema named *Hotel*
room is a set of room-status pairs.
each room is uniquely identified so *room* is a function, a schema variable

InitHotel

$Hotel$

$room = \{\}$

room is the empty set

Commission

$\Delta Hotel$
$room? : ROOM$

includes entire contents of *Hotel* schema
the room input is a *ROOM* object

$room? \notin \mathrm{dom}\ room$
$room' = room \cup \{ room? \mapsto vacant \}$

room input is not in the domain of *room*
adds the room-vacant pair to *room*

Occupy

$\Delta Hotel$
$room? : ROOM$

variables defined in *Hotel* may change

$room? \in \mathrm{dom}\ room$
$room\ (room?) = vacant$
$room' = room \oplus \{ room? \mapsto occupied \}$

the room input is in the domain of *room*
function application shows it is *vacant*
function override updates *room*

Vacate

$\Delta Hotel$
$room? : ROOM$

$room? \in \mathrm{dom}\ room$
$room\ (room?) = occupied$
$room' = room \oplus \{ room? \mapsto vacant \}$

dom is the first component of all pairs

room input is updated with *vacant*

QueryOccupiedRooms

$\Xi Hotel$
$occupied! : \mathbb{Z}$

variables in *Hotel* remain unchanged
occupied output is an integer

$occupied! = \#\ room$

and contain the number of pairs in *room*

Decommission

$\Delta Hotel$
$room? : ROOM$

$room? \in \text{dom } room$
$room\ (room?) = vacant$
$room' = room \setminus \{ room? \mapsto room\ (room?) \}$ removes the input room from *room* set

REPORT ::= *ok* | *alreadyAdded* | *noSuchRoom* | *notEmpty* | *notOccupied*

is a free type with the given values.

Success

$report! : REPORT$ report output is a *REPORT* object

$report! = ok$ and its value is *ok*

DuplicateRoom

$\Xi Hotel$
$room? : ROOM$
$report! : REPORT$

$room? \in \text{dom } room$
$report! = alreadyAdded$

NoSuchRoom

$\Xi Hotel$
$room? : ROOM$
$report! : REPORT$

$room? \notin \text{dom } room$
$report! = noSuchRoom$

RoomNotEmpty

$\Xi Hotel$
$room? : ROOM$
$report! : REPORT$

$room\ (room?) \neq vacant$ the room input is not vacant
$report! = notEmpty$

RoomNotOccupied

$\Xi Hotel$
$room? : ROOM$
$report! : REPORT$

$room\ (room?) \neq occupied$
$report! = notOccupied$

$CommissionTot \mathrel{\widehat=} (Commission \land Success) \lor DuplicateRoom$

$\mathrel{\widehat=}$ means schema definition. $\land$ means AND. $\lor$ means OR.

2 The Library Loans System

We introduce BORROWER, the collection of all possible library borrowers, and COPY, the collection of all possible copies of items that may be loaned to borrowers.

$[BORROWER, COPY]$

We define lending limit, a limit on the number of items a library member may borrow at any one time. We are not concerned with the actual value of lending limit.

$lendingLimit : \mathbb{Z}$

We specify *Library* as a finite (i.e. countable) set (i.e. collection) of borrowers, a finite set of loanable copies and a set of loans that relate each borrowed copy to its borrower. Each copy is uniquely identified. Each loaned copy must belong to the set of loanable copies. Each person who has borrowed a copy must be a registered library borrower. Nobody can borrow more than the lending limit.

Library

$borrower : \mathbb{F}\ BORROWER$
$copy : \mathbb{F}\ COPY$
$loan : COPY \twoheadrightarrow BORROWER$

$\mathrm{dom}\ loan \subseteq copy$
$\mathrm{ran}\ loan \subseteq borrower$
$loan = \{ c : COPY; b : BORROWER \mid c \mapsto b \in loan \land \#\ (loan \rhd b) \leqslant lendingLimit \}$

Initially, there are no registered borrowers, no loanable copies and no loans.

InitLibrary

Library

$borrower = \varnothing$

$copy = \varnothing$

$loan = \varnothing$

We add a new borrower to the library. The new borrower cannot already be a member of the library. The collections of loanable copies and loans remain unchanged.

AddBorrower

$\Delta Library$

$borrower? : BORROWER$

$borrower? \notin borrower$

$borrower' = borrower \cup \{ borrower? \}$

$copy' = copy$

$loan' = loan$

We add a new loanable copy to the system. The new copy cannot already be in the collection of loanable copies. The collections of borrowers and loans remain unchanged.

AddCopy

$\Delta Library$

$copy? : COPY$

$copy? \notin copy$

$copy' = copy \cup \{ copy? \}$

$borrower' = borrower$

$loan' = loan$

We issue (i.e. loan) a copy to a borrower. The borrower must be a member of the library and must have not have reached their lending limit. The copy must be a loanable copy and not currently out on loan. We record the new loan. Issuing a copy to a borrower has no affect on the collection of borrowers or the collection of loanable copies.

Issue

$\Delta Library$
$borrower? : BORROWER$
$copy? : COPY$

$borrower? \in borrowers$
$\# (loan \rhd \{ borrower? \}) < lendingLimit$
$copy? \in copy$
$copy? \notin \mathrm{dom}\ loan$
$loan' = loan \cup \{ copy? \mapsto borrower? \}$
$borrower' = borrower$
$copy' = copy$

We record the return of a loaned copy. The copy must be currently recorded as out on loan; we remove the loan from our list of current loans. The collections of borrowers and copies remain unchanged.

Return

$\Delta Library$
$copy? : COPY$

$copy? \in \mathrm{dom}\ loan$
$loan' = loan \setminus \{ copy? \mapsto loan\ (copy?) \}$
$borrower' = borrower$
$copy' = copy$

We search through the current loans to determine who, if anybody, has a given copy out on loan.

QueryWhoHasCopy

$\Xi Library$
$copy? : COPY$
$borrower! : \mathbb{F}\ BORROWER$

$borrower! = loan\ (\!|\ \{ copy? \}\ |\!)$

We search through the current loans to determine which copies, if any, that a given borrower has out on loan.

QueryBorrowersLoans

$\Xi Library$
$borrower? : BORROWER$
$copy! : \mathbb{F}\ COPY$

$copy! = loan \sim (\!|\ \{\ borrower?\ \}\ |\!)$

We now turn our attention to the error scenarios, none of which update our lists of borrowers, loanable copies or loans.

First we define the REPORT type with the values success or otherwise to be reported by our various processes.

$$REPORT ::= ok \mid isAMember \mid isALoanCopy \mid notAMember \mid notALoanCopy \mid lendingLimitReached \mid copyOnLoan \mid copyNotOnLoan$$

We define *Success*, which reports *ok*; we shall combine it with all our main success scenarios.

Success

$\Xi Library$
$report! : REPORT$

$report! = ok$

We cannot enrol the same person twice as a member of the library.

IsAMember

$\Xi Library$
$borrower? : BORROWER$
$report! : REPORT$

$borrower? \in borrower$
$report! = isAMember$

We cannot add the same copy to our collection of loanable copies twice.

IsALoanCopy

$\Xi Library$
$copy : COPY$
$report! : REPORT$

$copy? \in copy$
$report! = isALoanCopy$

A person who is not a member of the library cannot take anything out on loan.

IsANotMember

$\Xi Library$
$borrower? : BORROWER$
$report! : REPORT$

$borrower? \notin borrower$
$report! = notAMember$

Any member of the library cannot borrow more items than they are entitled to.

LendingLimitReached

$\Xi Library$
$borrower? : BORROWER$
$report! : REPORT$

$\# (loan \rhd \{ borrower? \}) \geqslant lendingLimit$
$report! = lendingLimitReached$

Borrowers cannot take out on loan any copy that is not loanable.

NotALoanCopy

$\Xi Library$
$copy? : COPY$
$report! : REPORT$

$copy? \notin copy$
$report! = notALoanCopy$

It is an error for a member to borrow a copy that is already out on loan.

CopyOnLoan

$\Xi Library$
$copy? : COPY$
$report! : REPORT$

$copy? \in \mathrm{dom}\ loan$
$report! = copyOnLoan$

You cannot record the return of a copy if it is not out on loan.

CopyNotOnLoan

$\Xi Library$
$copy? : COPY$
$report! : REPORT$

$copy? \notin \text{dom } loan$
$report! = copyNotOnLoan$

We can now define our total and complete processes.

The complete add a borrower process is defined to be add borrower and success, or the borrower is already a member.

$$AddBorrowerTot \mathrel{\widehat=} (AddBorrower \land Success) \lor IsAMember$$

The complete add a loanable copy process is defined to be add copy and success, or the copy has already been added.

$$AddCopyTot \mathrel{\widehat=} (AddCopy \land Success) \lor IsALoanCopy$$

The complete issue a copy to a borrower process is defined to be issue and success; it is an error if the borrower is not a member, or if their lending limit has been reached, or if the copy they are attempting to borrow is not available for loan.

$$IssueTot \mathrel{\widehat=} (Issue \land Success) \lor IsNotAMember \lor LendingLimitReached \lor NotALoanCopy \lor CopyOnLoan$$

Finally, our complete return process is defined to be return and success, or the copy is not out on loan.

$$ReturnTot \mathrel{\widehat=} (Return \land Success) \lor CopyNotOnLoan$$

Chapter 11

Ex 11.1

1a head $< h, e, a, d > = h$
b last $< l, a, s, t > = t$
c front $< f, r, o, n, t > = < f, r, o, n >$
d tail $< t, a, i, l > = < a, i, l >$
e $< m, a, x > \frown < p, o, w, e, r > = < m, a, x, p, o, w, e, r >$
f $< f, i, l, t, e, r > \upharpoonright < i, t, s > = < i, t >$
g $< 2, 4, 6 > \upharpoonright < e, x, t, r, a > = < x, r >$

2 *text* : seq *CHAR* - a sequence of characters, repetitions allowed e.g. $< a, b, b, a >$
text : seq *CHAR* - a sequence of characters, repetitions not allowd e.g. $< a, b >$

Ex 11.2

1 Models the behaviour of aircraft stacked on a queue waiting to join the queue on the final approach.

```
bound : ℕ
```

There is an upper limit to the number of aircraft that can be queued in the stack. If this limit is reached aircraft are directed to another airport.

```
[FLIGHTID]
```

Queue

```
queue : iseq FLIGHTID
```

A queue is a sequence of aircraft.

InitQueue

```
Queue'
─────────────────
queue' = ⟨⟩
```

Initially, the queue is empty.

Join

```
ΔQueue
aircraft? : FLIGHTID
─────────────────
# queue < bound
aircraft? ∉ ran queue
queue' = queue ⌢ ⟨aircraft?⟩
```

Usually, aircraft join the rear of the queue.

InsertBefore

```
ΔQueue
aircraft? : FLIGHTID
insertBefore? : ℕ × FLIGHTID
─────────────────
# queue < bound
queue ≠ ⟨⟩
insertBefore? ∈ queue
aircraft? ∉ ran queue
queue'
 = squash {q:queue | first q < first insertBefore? }⌢⟨aircraft?⟩
   ⌢ squash { q : queue | first q ≥ first insertBefore? }
```

But an arriving aircraft can be inserted before a particular aircraft already in the queue.

```
┌─Leave──────────────────────────────────────────────┐
│ ΔQueue                                             │
├────────────────────                                │
│ queue ≠ ⟨⟩                                         │
│ queue' = tail queue                                │
└────────────────────────────────────────────────────┘
```

Aircract leaves the stack to join the final approach queue.