Exercises 8.1

1. Write schemas for operations to add and remove a member to or from the video shop.
2. Write a schema for the operation whereby a member returns a video to the shop.
3. How could you modify the state schema to allow a maximum of n videos to be rented by any given member?
4. Write a schema for an operation to output the set of all people who have a given title on rental.
5. Write a schema *SimilarTastes* to output the set of all people who have on rental at least one of the titles currently rented out to a given person.
6. Write exception handling schemas to totalise (make robust) the above operations.
7. Given the schema *AddTitle*, which adds a new title to the stock,

AddTitle

$$\Delta VideoShop$$
$$t? : TITLE$$
$$level? : \mathbb{N}_1$$

$$stockLevel' = stockLevel \cup \{t? \mapsto level?\}$$
$$members' = members$$
$$rented' = rented$$

what implicit precondition is present in this schema?

8. How could you modify the state schema *VideoShop* to allow any given member to rent more than one copy of a given title at the same time?

CHAPTER 9

Sequences

Aims

To introduce the concept of the sequence as a specialised sort of function, to introduce some sequence operators and to demonstrate the application of sequences in writing specifications.

Learning objectives

When you have completed this chapter, you should be able to:

- understand the kinds of system which may be modelled using sequences, and the styles of specification commonly used with sequences;
- understand the effect of relation, function and sequence operators when applied to sequences, and how to construct sequence-valued expressions using them;
- understand how the Z language may be extended by adding generic axiomatic definitions to specifications, and appreciate when it is appropriate to do so.

9.1 Introduction

Sequences embody the idea of the members of a set being arranged in a particular *order*. Examples from everyday life are situations such as a supermarket checkout queue (a sequence of people), a phone directory (a sequence of names arranged alphabetically, each paired up with the corresponding phone number) or a queue at traffic lights (a sequence of vehicles).

Sequences allow us to model the common linear abstract data types of computer science, for example lists, stacks and queues. As artefacts in a specification for a computer program, sequences may naturally be implemented in the target language as arrays, arrangements of pointers and records/structures,

or object-oriented container classes. The translation for functional programming languages, where lists are built in and other recursive data types are easily defined, is even more straightforward.

9.2 Sequences in Z

A sequence is a restricted sort of function. The restriction is that the domain of a sequence must be a prefix subset of $\mathbb{N}_1$, the natural numbers excluding zero. In other words, if the sequence contains n maplets, its domain will be the set $1 \,..\, n$. For example, given the type

$[PERSON]$ the set of all people

the function

$$s : \mathbb{N}_1 \rightarrow PERSON \quad \text{where } s = \{1 \mapsto tom, 2 \mapsto dick, 3 \mapsto harry\}$$

is a sequence, whereas the function

$$t : \mathbb{N}_1 \rightarrow PERSON \quad \text{where } t = \{1 \mapsto tom, 2 \mapsto dick, 6 \mapsto harry\}$$

is not a sequence.

We would refer to s as a *sequence of people*. In other words, a sequence defines an ordering of the items in its range. In the sequence s, *tom* is the first element, *dick* the second and *harry* the third. This sequence may be written using the shorthand notation

$$s = \langle tom, dick, harry \rangle$$

The *length* of a sequence is simply its cardinality, that is the number of pairs it contains. Thus the length of s is 3.

The *empty sequence* is represented by $\langle\rangle$. Its length is 0.

We would declare s as a sequence-valued variable as follows:

$$s : \text{seq } PERSON$$

This declaration is a shorthand for

$$s : \mathbb{N} \rightarrow PERSON$$

together with the restriction

$$\text{dom } s = 1 \,..\, \#s$$

1..0 should be the empty set!

This declaration allows s to be empty. To specify that s has at least one element, we can use the declaration

$$s : \text{seq}_1 \; PERSON$$

In general, sequences may have repeated elements, for example

$$\langle tom, dick, dick \rangle$$

Clearly, a sequence with repeating elements is not injective (one-to-one), and vice versa. Therefore, to specify that, for a sequence t, repeating elements are not allowed, we may use the declaration for an *injective sequence*:

$$t : \text{iseq } PERSON$$

This is useful when we wish to model systems such as supermarket queues, where a person cannot be in two different places within a queue at the same time.

Here are some more examples of sequences:

$$s = \langle \langle 1, 2 \rangle, \langle 3, 2, 7 \rangle, \langle\rangle, \langle 1, 6 \rangle \rangle$$

is a sequence of sequences of integers, and would be declared as

$$s : \text{seq}(\text{seq } \mathbb{Z}) \quad \text{or perhaps} \quad s : \text{iseq}(\text{iseq } \mathbb{Z})$$

and

$$t = \langle \{3, 6\}, \{\}, \{6, 8\}, \{6, 3\} \rangle$$

is a sequence of sets of integers, and would be declared as

$$t : \text{seq}(\mathbb{P}\, \mathbb{Z})$$

and

$$u = \langle \{5 \mapsto 6, 8 \mapsto 4\}, \{2 \mapsto 10, 4 \mapsto 9\} \rangle$$

is a sequence of homogeneous functions on the integers, and would be declared as

$$u : \text{seq}(\mathbb{Z} \rightarrow \mathbb{Z})$$

Now, because a sequence is a restricted sort of function, which in turn is a restricted sort of relation, which is a restricted sort of set, we already have a rich collection of operators which can be used with sequences, provided the

Z type rules are not violated. However, the expressions resulting from applying such operators to sequences are not necessarily sequence-valued. For example, the expression

$$\{1,2\} \lhd \{1 \mapsto tom, 2 \mapsto dick, 3 \mapsto harry\}$$

is the sequence

$$\{1 \mapsto tom, 2 \mapsto dick\}$$

whereas the expression

$$\{1,3\} \lhd \{1 \mapsto tom, 2 \mapsto dick, 3 \mapsto harry\}$$

simplifies to

$$\{1 \mapsto tom, 3 \mapsto harry\}$$

which is a function but is not a sequence, because its domain is not a prefix subset of $\mathbb{N}_1$.

It is extremely important when writing specifications to be very clear about the types of the structures you are using, and the validity of expressions which you are associating with variables of those types in your predicates. Each declaration of a variable of a given type introduces implicit invariants into the schema in which it is used, and one of the most common sources of error in specifications is inconsistencies in the types of expressions. Software tools are available to help to identify these errors, but there is no substitute for clarity and depth of thought about the problem, and care in putting together your formal model of it. Here is another example.

The expression

$$\{1 \mapsto tom, 2 \mapsto dick, 3 \mapsto harry\} \cup \{1 \mapsto carol, 2 \mapsto janet\}$$

simplifies to

$$\{1 \mapsto tom, 1 \mapsto carol, 2 \mapsto dick, 2 \mapsto janet, 3 \mapsto harry\}$$

which is a relation that is neither a sequence nor a function. However, the expression

$$(\{1 \mapsto tom, 2 \mapsto dick, 3 \mapsto harry\} \cup \{1 \mapsto carol, 2 \mapsto janet\}) \rhd \{tom, dick\}$$

simplifies to

$$\{1 \mapsto carol, 2 \mapsto janet, 3 \mapsto harry\}$$

which is a sequence.

Exercises 9.1

1. Simplify and comment on the types of the following expressions:

(i) $\langle a, b, c\rangle \cup \langle d, e, f\rangle$
(ii) $\langle a, b, c\rangle \cup \langle a, b\rangle$
(iii) $\langle a, b, c\rangle \cap \langle a, b\rangle$
(iv) $\langle a, b, c\rangle \setminus \langle c\rangle$
(v) $\langle a, b, c\rangle \rhd \{a, b\}$
(vi) $\{1\} \lhd \langle a, b, c\rangle$
(vii) $\langle a, b, c\rangle \oplus \langle d, e, f\rangle$
(viii) $\langle a, b, c\rangle^{-1}; \langle d, e, f\rangle$
(ix) $\langle 1, 2, 3\rangle \rhd (\mathrm{dom}\langle 1, 2, 3\rangle)$

2. Given the sequence $s = \langle tom, dick, harry\rangle$, what is the value of the following?

(i) $s\,1$ (s applied to 1)
(ii) $s(\#s)$

3. Given the types

$[PERSON]$ the set of all people
$[CAR]$ the set of all cars

write down possible values for the following variables and draw the corresponding pictures:

(i) $p : \mathrm{seq}(\mathbb{P}\ PERSON)$
(ii) $q : \mathrm{seq}(PERSON \rightarrow CAR)$
(iii) $r : \mathrm{seq}(\mathrm{seq}\ PERSON)$
(iv) $s : \mathrm{seq}((\mathbb{P}\ PERSON) \leftrightarrow CAR)$
(v) $t : \mathbb{P}(\mathrm{seq}\ CAR)$

9.3 Sequence operators

We will now introduce some additional operators for use with sequences.

The function *head* returns the first element in a non-empty sequence. For example,

$$head\langle tom, dick, harry\rangle = tom$$

Note that *head* returns *tom*, not the maplet $1 \mapsto tom$.

The function *tail* returns the sequence formed by removing the first maplet in a non-empty sequence (and, if necessary, modifying the domain of the result to make it a sequence). For example,

$$tail\langle tom, dick, harry\rangle = \langle dick, harry\rangle$$

The function *last* returns the last element in a non-empty sequence. For example,

$$last\langle tom, dick, harry\rangle = harry$$

Note that *last* returns *harry*, not the maplet $3 \mapsto harry$.

The function *front* returns the sequence formed by removing the last maplet in a non-empty sequence. For example,

$$front\langle tom, dick, harry\rangle = \langle tom, dick\rangle$$

The function *rev* returns the sequence formed by reversing the order of the elements in a given sequence. For example,

$$rev\langle tom, dick, harry\rangle = \langle harry, dick, tom\rangle$$

The *concatenation* operator $\frown$ takes two sequences and returns the sequence formed by 'joining them together'. For example,

$$\langle tom, dick, harry\rangle \frown \langle andy, sandy, randy\rangle = \langle tom, dick, harry, andy, sandy, randy\rangle$$

The *filter* operator $\upharpoonright$ takes a sequence and a set of the same type as the sequence's range set and returns the sequence formed by removing all maplets which do not contain, as their second element, members of the set. For example,

$$\langle tom, dick, harry\rangle \upharpoonright \{dick, harry, sandy\} = \langle dick, harry\rangle$$

The *squash* function takes any function f such that

$$\operatorname{dom} f \subseteq \mathbb{N}_1$$

and returns the sequence formed by modifying the domain of f, maintaining the original order which it defines. For example,

$$squash\{2 \mapsto dick, 3 \mapsto tom, 7 \mapsto harry\} = \{1 \mapsto dick, 2 \mapsto tom, 3 \mapsto harry\}$$

9.4 Generic constants

The above functions have been introduced informally, with English descriptions and examples. However, such functions may be formally defined as *generic constants*. They must be generic, because they must be able to operate on sequences of any given base type; that is, they must be capable of being applied to sequences of integers, sequences of people, sequences of sequences of integers, etc. For example, here is the definition from Spivey (1992) of sequence concatenation:

$$[X]$$
$$\_\frown\_ : \operatorname{seq} X \times \operatorname{seq} X \to \operatorname{seq} X$$
$$\forall s, t : \operatorname{seq} X \bullet s \frown t = s \cup \{n : \operatorname{dom} t \bullet n + \#s \mapsto t(n)\}$$

A generic constant has one or more generic formal parameters. In the above example, the formal parameter is X, which stands for any actual parameter set supplied implicitly when the $\frown$ operator is used. The declaration

$$\_\frown\_ : \operatorname{seq} X \times \operatorname{seq} X \to \operatorname{seq} X$$

states that $\frown$ is an infix function (indicated by the position of the underscores which shows where the parameters should go) which may be applied to any pair of sequences of the same type, to return another sequence of that type. There is a notation for explicitly supplying actual generic parameters when the operator is used, but this may be left implicit. For example, in the expression

$$\langle 1, 2, 3\rangle \frown \langle 4, 5\rangle$$

the formal parameter X has clearly been instantiated as $\mathbb{Z}$. The predicate

$$\forall s, t : \operatorname{seq} X \bullet s \frown t = s \cup \{n : \operatorname{dom} t \bullet n + \#s \mapsto t(n)\}$$

defines the sequence returned by the operator to consist of the pairs from the first operand, together with the pairs from the second one after making an appropriate shift to its domain.

As a further example, here is a generic constant definition for the *head* function:

$$[X]$$
$$head : \operatorname{seq}_1 X \to X$$
$$\forall s : \operatorname{seq}_1 X \bullet head\ s = s\,1$$

The declaration states that *head* is a prefix function (no underscores) which may be applied to any non-empty sequence to return the first element of that sequence. The predicate states that the element returned by *head* is precisely that which would result from applying the sequence to the number 1. The type seq_1 is required because *head* is not defined for the empty sequence. It is also a requirement when defining generic constants that the definition must uniquely determine the value of the constant for all possible values of the formal parameters.

Similar generic constant definitions are used to define many other standard Z operators, and we can use them to define new operators for our own specifications. However, this should not be done to excess. New operators should only be introduced if they improve the clarity of the specification. Examples would be if the operator was required in several places in the specification, or if the expressions required as an alternative to the operator were very complex and opaque. It should be mentioned that it is also possible to define generic *schemas* in Z; see Spivey (1992) for further details.

Exercises 9.2

1. Simplify the following expressions:

   (i) $\langle 1,2,3\rangle \frown \langle\rangle$
   (ii) $\text{dom}\langle a,b,c\rangle$
   (iii) $\text{ran}\langle 1,1,2\rangle$
   (iv) $\{a \mapsto 2, b \mapsto 3, c \mapsto 1\}^{-1}$
   (v) $\text{dom}\langle\langle 1,2\rangle \frown \langle 3,4\rangle\rangle$
   (vi) $\{1\} \lhd \mathit{tail}(\langle a,b,c\rangle)$
   (vii) $\text{dom}((\mathit{front}\langle 1,3,5,7\rangle)^{-1})$
   (viii) $\mathit{head}(\mathit{tail}(\mathit{tail}(\langle 1,7,9,2,2\rangle \frown \langle 2,4,5\rangle)))$
   (ix) $\mathit{last}(\mathit{tail}\,(\langle\langle\rangle, \langle 1\rangle, \langle 1,2\rangle, \langle 1,2,3\rangle, \langle 1,2,3,4\rangle\rangle)) \frown \langle 1,2\rangle$
   (x) $\mathit{squash}(3\,..\,5 \lhd \langle a,b,c,d,e,f\rangle)$
   (xi) $\mathit{rev}(\langle 2,3,4,6,8\rangle \upharpoonright (\text{dom}\langle a,b,c\rangle))$

2. Given the declaration s: $\text{iseq}\,\mathbb{Z}$, write a predicate to specify that the numbers in the range of s are in non-descending order.
3. Give a generic constant definition for the function *tail*.
4. Give a generic constant definition for the function *for* which returns the prefix subsequence of a given sequence from its beginning up to a given position. If the given position is greater than or equal to the length of the sequence, the operation should return the whole sequence. For example,

$$\langle tom, dick, harry\rangle \mathit{\ for\ } 2 = \langle tom, dick\rangle$$
$$\langle tom, dick, harry\rangle \mathit{\ for\ } 7 = \langle tom, dick, harry\rangle$$

9.5 disjoint, partition

We can specify that a sequence of sets

$$\langle A_1, A_2, \ldots, A_n\rangle$$

is *pairwise disjoint*, that is none of the sets intersect with each other, using the expression

$$\text{disjoint}\langle A_1, A_2, \ldots, A_n\rangle$$

For example, the following is true

$$\text{disjoint}\langle\{1,2,3\}, \{7,8\}, \{12,13,6\}\rangle$$

A sequence of sets

$$\langle A_1, A_2, \ldots, A_n\rangle$$

partitions a set S iff the union of all the sets in the sequence is S, and the sets in the sequence are pairwise disjoint. This is captured by the expression

$$\langle A_1, A_2, \ldots, A_n\rangle \text{ partition } S$$

For example, the following is true

$$\langle\{1,2\}, \{3,4\}, \{5\}\rangle \text{ partition } \{1,2,3,4,5\}$$

We will use this concept in the next section.

9.6 The university badminton club revisited

This specification was introduced in Chapters 4 and 5. We will now refine part of it to describe the activity in the hall, which contains one badminton court, where members of the club come to play. People in the hall are either playing a game on the court, or effectively in a queue, waiting to play. We will model this queue as an injective sequence called *waiting*, and represent those playing a game by the set *onCourt*. The specification could be implemented as a computer program to run on a portable machine kept by the club secretary, or more likely, as a system of paper records of club membership, together with a wooden name board to indicate who is queuing and who is playing a game.

Recall that the state in the original example was described by the schema

ClubState

$$badminton : \mathbb{P}\ STUDENT$$
$$hall : \mathbb{P}\ STUDENT$$
$$hall \subseteq badminton$$
$$\#\ hall \leqslant maxPlayers$$

We will reuse and extend this description, by including it in a new state schema, *ClubState2*:

ClubState2

$$ClubState$$
$$onCourt : \mathbb{P}\ STUDENT$$
$$waiting : \text{iseq}\ STUDENT$$
$$\langle onCourt, \text{ran}\ waiting \rangle \text{ partition } hall$$

The predicate

$$\langle onCourt, \text{ran}\ waiting \rangle \text{ partition } hall$$

states that everyone in the hall is either playing a game or waiting to do so, and that nobody is both waiting and playing at the same time!

The initial state, as before, is a club with no members:

InitClubState2

$$ClubState2'$$
$$badminton' = \{\}$$

Note that if *badminton'* is empty, the state invariant implies that all of the sets in the state are empty. We will now specify three operations.

Beginning a new game

For a new game to begin, there must not be a game currently in progress

$$onCourt = \varnothing$$

and there must be at least two people in the queue (one cannot play badminton by oneself!)

$$\#\ waiting \geqslant 2$$

If there are four or more people in the queue, then four people will play in the new game.

$$\#\ waiting \geqslant 4 \Rightarrow \#\ onCourt' = 4$$

If there are less than four people in the hall, then either two people or three people will play in the new game.

$$\#\ waiting < 4 \Rightarrow (\#\ onCourt' = 2) \lor (\#\ onCourt' = 3)$$

The rules of the club state that the people to play the next game will comprise the person at the front of the queue

$$\text{head}\ waiting \in onCourt'$$

together with the appropriate number of people as defined above, selected by him/her from up to the next five positions in the queue

$$onCourt' \subseteq ran(1\,..\,6 \lhd waiting)$$

Note that this predicate is non-deterministic in that it simply states that all the people in the new game must have come from the first six places in the queue. We can specify the rules for choosing the players, but our Z specification cannot capture the vindictiveness and favouritism involved in making the decision!

Finally, the predicate

$$waiting' = waiting \upharpoonright ((\text{ran}\ waiting) \setminus onCourt')$$

states that the new queue is equal to the old queue with those chosen for the game removed. We subtract the players chosen for the new game from the range of *waiting*, and then filter the sequence with this set.

The operation schema is as follows:

NewGame

$$\Delta\ ClubState2$$
$$onCourt = \varnothing$$
$$\#\ waiting \geqslant 2$$
$$\#\ waiting \geqslant 4 \Rightarrow \#\ onCourt' = 4$$
$$\#\ waiting < 4 \Rightarrow (\#\ onCourt' = 2) \lor (\#\ onCourt' = 3)$$
$$\text{head}\ waiting \in onCourt'$$
$$onCourt' \subseteq \text{ran}(1\,..\,6 \lhd waiting)$$
$$waiting' = waiting \upharpoonright ((\text{ran}\ waiting) \setminus onCourt')$$
$$hall' = hall$$
$$badminton' = badminton$$

Ending a game

To end a game, there must be one taking place!

$$onCourt \neq \{\}$$

The players come off the court

$$onCourt' = \{\}$$

and join the back of the queue in an unspecified order.

$$\exists s : \text{iseq } STUDENT \bullet (\text{ran } s = onCourt \land waiting' = waiting \frown s)$$

Again, this predicate is non-deterministic in that it describes the relationship required between the sets without actually stating how the operation is to achieve this. This may seem a little strange to those used to writing programs, where one must state specifically how a task is to be done, but at the specification level we are free to think at a higher level of abstraction: to write an expression which characterises the relationship between a 'before' state and an 'after' state, without necessarily indicating how this is to be achieved. The specification task is not about writing recipes for achieving results, but simply stating what those results should be. (Some clubs have a policy that the winners of the game get to go ahead of the losers in the waiting list, but in a doubles game, there is still uncertainty about which of the winners and which of the losers goes ahead of the other!)

The operation schema is as follows:

FinishGame

Δ *ClubState2*

$onCourt \neq \{\}$
$onCourt' = \{\}$
$\exists s : \text{iseq } STUDENT \bullet$
$(\text{ran } s = onCourt \land waiting' = waiting \frown s)$
$hall' = hall$
$badminton' = badminton$

put those who were playing back into the queue in some non-specified order

A person leaving the hall

We assume that our person $p?$ does not leave in the middle of a game!

$$p? \in \text{ran } waiting$$

We use $\rhd$ to remove our person from the queue, and *squash* to restore the result to be a valid sequence.

$$waiting' = squash(waiting \rhd \{p?\})$$

The person is removed from the hall by the predicate

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

The operation schema is as follows:

LeaveHall

Δ *ClubState2*
$p? : STUDENT$

$p? \in \text{ran } waiting$
$waiting' = squash(waiting \rhd \{p?\})$
$hall' = hall \setminus \{p?\}$
$badminton' = badminton$

Note that we have not explicitly stated that *onCourt* is not changed. This is implicitly specified by the state invariant. However, it is sometimes considered to be good practice, in the interests of clarity, to make such properties explicit in the operation schema.

This is a new version of the *LeaveHall* schema, which was first written for the simplified system state described in Chapters 4 and 5. We could have reused and modified the original version, but this would not have improved the readability of the specification. To complete the specification, we would have to modify the other operation schemas from Chapters 4 and 5 to operate on the new state, and we would have to totalise all operations using the schema calculus. You may wish to try this as an additional exercise.

Exercises 9.3

1. Write a schema to specify an operation for a person to enter the hall and join the back of the *waiting* queue.
2. Write a Z predicate which states that a given sequence of characters s is a substring of a given sequence of characters t.
3. Write a Z expression for the number of occurrences of a natural number n in a sequence of natural numbers s.

4. What characteristics must a sequence possess if its inverse is also a sequence?
5. Write a Z predicate which states that a given sequence of characters s is a palindrome; that is, it spells the same backwards as it does forwards.

CHAPTER 10

A third specification: Project allocation

Aims

To build on the material of the previous chapters by developing a further specification which uses sequences, and to introduce the concept of schema composition. can be applied to web service composition !

Learning objectives

When you have completed this chapter, you should be able to:

- feel more confident in identifying appropriate applications of sequences in your specifications;
- develop specifications which use combinations of all the discrete mathematical structures which you have met in this book;
- use schema composition to create new operation schemas from existing ones.

10.1 Introduction

In the previous chapter, we developed a specification which used a sequence to model a queue of people. In this chapter, we will take this a stage further by using sequences in combination with functions to model a more complex situation, namely the allocation of undergraduate projects on a university degree course.

10.2 Allocation of undergraduate projects: the problem

A university requires a computerised system to manage the allocation of the individual projects undertaken by its final-year degree students. Each student