CHAPTER 12

Sequences

12.1 A sequence is a function

Very often it is necessary to be able to distinguish values of a set by position or to permit duplicate values or to impose some ordering on the values. For this a *sequence* is the appropriate structure.

A sequence of elements of type X is regarded in Z as a function from the natural numbers to elements of X. The domain of the function is defined to be the interval of the natural numbers starting from one and going up to the number of elements in the sequence, with no gaps.

A sequence s of elements of type X is declared:

$$s: \mathrm{seq}\ X$$

and is equivalent to declaring the function:

$$s: \mathbb{N} \twoheadrightarrow X$$

with the constraint that:

$$\mathrm{dom}\ s = 1 \mathrel{..} \#s$$

12.2 Sequence constructors

A sequence constant can be constructed by listing its elements in order, enclosed by special angle brackets and separated by commas:

[CITY] the set of cities of the world
flight: seq CITY

$$flight = \langle\ Geneva, Paris, London, NewYork\ \rangle$$

This is a shorthand for the function:

$$flight = \{1 \mapsto Geneva, 2 \mapsto Paris, 3 \mapsto London, 4 \mapsto NewYork\}$$

The order of cities in *flight* might be the order in which those cities are visited on a journey starting in Geneva and finishing in New York.

12.2.1 The length of a sequence

The length of a sequence s is simply the size of the function

$$\#s$$

so

$\#flight = 4$

12.2.2 Empty sequence

The sequence with no elements is written as empty sequence brackets:

$\langle\rangle$

12.2.3 Non-empty sequences

If a sequence may never be empty, that is, it must always have at least one element, it can be declared:

$s: seq_1\ X$

which is the same as defining

$s: seq\ X$

and adding the constraint

$\#s > 0$

12.3 Sequence operators

12.3.1 Selection

Since a sequence is a function, it is possible to select an element by position simply by function application. For example, to select the third element of *flight*:

flight 3 this is London

As with a function, it only makes sense to attempt to select with a value which is in the domain of the function. In terms of the sequence, that means the position value must be between one and the number of elements in the sequence.

12.3.2 Head

The *head* of a sequence is its first element, so

head flight

is the same as

flight 1 this is Geneva

12.3.3 Tail

The *tail* of a sequence is the sequence with its head removed, so

tail flight

is the sequence

⟨ Paris, London, NewYork ⟩

12.3.4 Last

The *last* of a sequence is its last element, so

last flight

is the same as

flight #flight   this is NewYork

12.3.5 Front

The *front* of a sequence is the sequence with its last element removed, so

front flight

is the sequence

⟨ Geneva, Paris, London ⟩

12.3.6 Concatenation

The *concatenation* operator is written

$\frown$

and pronounced 'concatenated with' or 'catenated with'. It chains together two sequences to form a new sequence. For example:

⟨ Geneva, Paris, London, NewYork ⟩ $\frown$ ⟨ Seattle, Tokyo ⟩

is the sequence

⟨ Geneva, Paris, London, NewYork, Seattle, Tokyo ⟩

12.3.7 Filtering

The operation of *filtering* a sequence produces a new sequence, all of whose elements are members of a specified set. Its effect is similar to that of performing a range restriction, then 'squashing up' the elements to close up the gaps left by omitted elements. For example:

flight $\upharpoonright$ {London, Geneva, Rome} is pronounced 'flight filtered by the set containing London, Geneva and Rome' and results in the sequence:

$\langle$ London, Geneva $\rangle$

Note that the ordering remains that of the original sequence.
To form the sequence of those cities of *flight* which are in Europe given:

EuropeanCities: $\mathbb{P}$CITY
EuropeanCities = {Paris, London, Geneva, Rome}

we can write:

flight $\upharpoonright$ EuropeanCities

which is

$\langle$ Geneva, Paris, London $\rangle$

12.3.8 Restriction and squash

Since a sequence is a function and a function is a relation, the relational operators can be used. The relational restriction operators are particularly useful; for example, to select parts of a sequence.
Given:

S: seq X

then

$1..n \lhd S$

is the sequence of the first n elements of S.
In general, a restriction of a sequence *does not* yield a *sequence*, since the resulting domain will not be of contiguous natural numbers starting at one. In that case the special operator *squash* can be used to convert the relation into a sequence by 'closing up the gaps'.
So:

squash ($m .. n \lhd S$)

is the *sequence* of the elements from position m to position n of S. Note that *squash* only works for functions where the domain is the natural numbers.
The sequence filtering

flight $\upharpoonright$ EuropeanCities

is equivalent to

squash (flight $\rhd$ EuropeanCities)

12.3.9 Reversing a sequence

The operator

rev

reverses the order of elements in a sequence. For example

rev flight = ⟨ NewYork, London, Paris, Geneva ⟩

12.3.10 Range

Since a sequence is just a special case of a function, it is permissible, and sometimes useful, to refer to the *range* of a sequence; that is, the *set* of values which appear in the sequence. For example:

ran flight = {Geneva, Paris, London, NewYork}

12.4 Example of using sequences – stack

The well known and widely used data structure called a *stack* can be defined by means of sequences.

A stack is a data structure into which elements can be added ('pushed') and removed ('popped'). The next element to be popped is the one most recently pushed. This behaviour is also explained by referring to this as a *last-in-first-out* structure.

12.4.1 Types

The general type *X* is used:

[X]   any type

12.4.2 The state

```
__Stack_______________
| s:        seq X
|_____________________
```

12.4.3 The initialisation operation

```
__Init________________
| Stack'
|---------------------
| s' = ⟨⟩
|_____________________
```

Initially the sequence *s* is empty.

12.4.4 Push

A new element will be added at the front of the sequence:

```
__Push________________
| ΔStack
| x?:      X
|---------------------
| s' = ⟨x?⟩ ⁀ s
|_____________________
```

Note that the new value could just as well have been added at the back of the sequence, so long as *Pop* then had the appropriate definition.

12.4.5 Pop

An element will be removed from the front of the sequence:

```
__Pop_________________
| ΔStack
| x!:      X
|---------------------
| s ≠ ⟨ ⟩
| x! = head s
| s' = tail s
|_____________________
```

The precondition of *Pop* is that the sequence *s* should not be empty.

An alternative definition of *Pop*, which shows its symmetry with *Push*, is:

```
__Pop_________________
| ΔStack
| x!:      X
|---------------------
| s ≠ ⟨ ⟩
| s = ⟨x!⟩ ⁀ s'
|_____________________
```

12.4.6 Length

The *length* of the stack is the length of the sequence.

```
__Length______________
| ΞStack
| len!:    ℕ
|---------------------
| len! = # s
|_____________________
```

12.5 Example of using sequences – an air route

A route to be taken by a passenger on a journey by air can be described by the sequence of airports that the passenger will pass through. For the proposed journey to be viable, adjacent airports on the route must be connected by air services.

[AIRPORT] the set of airports in the world

```
AirServices
  _connected_: AIRPORT ↔ AIRPORT
```

An operation to propose a viable route from the originating to the destination airport might be:

```
ProposeRoute
  ΞAirServices
  from?, to?:   AIRPORT
  route!:       seq AIRPORT
  ----------------------------
  head route! = from?
  last route! = to?
  (∀ changePos: ℕ |
  changePos ∈ 1..#route! – 1 •
  route changePos connected
  route changePos + 1)
```

Of course, there may in fact not be a viable route between any two airports.

Note that this operation does not rule out providing a route which goes through the same airport more than once, and even allows flights which land back where they started. Since it seems unlikely that passengers would wish to fly more legs on their journeys than necessary, a better version would eliminate such excessively long routes:

```
NoDuplicatesRoute
  ProposeRoute
  ----------------------------
  (∀ i, j: ℕ |
  {i, j} ⊆ 1..#route! •
  i ≠ j ⇒ route i ≠ route j)
```

This says that for any *i* and *j* within the domain of the function (legal positions), if *i* and *j* are different, then so are the values in the sequence at positions *i* and *j*.

An alternative way of stating that no duplicates are permitted is to require that the inverse of the sequence be a function:

```
__NoDuplicatesRoute______
| ProposeRoute
|-----------------------
| route~ ∈ AIRPORT ⇸ ℕ
|________________________
```

or simply to state that:

$\#route = \#\mathrm{ran}\ route$

12.6 Sequences with no duplicates permitted

To specify that there are to be no duplicates in a sequence is a common requirement and so, for convenience, a special declaration of an *injective* sequence can be used.

iseq X

is the set of sequences of *X*'s where no value of *X* appears more than once in the sequence.

```
__NoDuplicatesRoute____________
| ΞAirServices
| from?, to?:      AIRPORT
| route!:          iseq AIRPORT
|------------------------------
| head route! = from?
| last route! = to?
| #route = #ran route
| (∀ changePos: ℕ |
|  changePos ∈ 1..#route! – 1 •
|  route changePos connected
|  route changePos + 1)
|______________________________
```

12.7 Example of using sequences – files in Pascal

In the programming language Pascal a file is a sequential structure of some type of elements. A file can be either in *inspection* mode or in *generation* mode. The file is put into inspection mode by a *Reset* operation and when in inspection mode can be read from by a *Read* operation. The file is put into *generation* mode by a *Rewrite* operation which makes the file empty. When in *generation* mode the file can have new elements appended to it by a *Write* operation.

This specification ignores buffering of data.

[X] any type of data (some restrictions in Pascal)
FILEMODE ::= inspection | generation

12.7.1 The file state

PascalFile

file: seq X
stillToRead: seq X
mode: FILEMODE

$\exists$ alreadyRead: seq X •
alreadyRead $\frown$ stillToRead = file

The part of the file still to be read is always a *suffix* of the whole file.

12.7.2 The Reset operation

Reset

ΔPascalFile

---

mode' = inspection
stillToRead' = file
file' = file

The mode is switched to inspection and the whole of the file is still to be read. The content of the file is not changed by this operation.

12.7.3 The Read operation

Read

ΔPascalFile
x!: X

---

mode = inspection
stillToRead $\neq \langle\, \rangle$
$\langle$ x! $\rangle \frown$ stillToRead' = stillToRead
file' = file
mode' = mode

The mode must be inspection; the part of the file still to be read must not be empty. The value returned is taken from the front of the part of the file still to be read. The file and its mode are unchanged.

12.7.4 The Rewrite operation

```
__Rewrite_______________
| ΔPascalFile
|_______________________
| mode' = generation
| file' = ⟨ ⟩
|________________________
```

The mode is switched to generation and the file becomes empty.

12.7.5 The Write operation

```
__Write_________________
| ΔPascalFile
| x?:      X
|_______________________
| mode = generation
| file' = file ⁀ ⟨ x? ⟩
| mode' = mode
|________________________
```

The mode must be generation. The value to be written is appended to the file. The mode is unchanged.

12.7.6 End of file

End of file is true when the part of the file still to be read is an empty sequence

stillToRead = $\langle\ \rangle$

12.8 Summary of notation

seq X	the set of sequences whose elements are drawn from X $== \{S: \mathbb{N} \nrightarrow X \mid \mathrm{dom}\ S = 1\ ..\ \#S\}$
seq_1 X	set of non-empty sequences
iseq X	set of injective sequences (no duplicates)
#S	the length of the sequence S
$\langle\ \rangle$	the empty sequence { }
$\langle x_1, \ldots x_n \rangle$	$== \{ 1 \mapsto x_1, \ldots, n \mapsto x_n\}$

$\langle x_1, \ldots x_n \rangle \frown \langle y_1, \ldots y_n \rangle$	concatenation: $== \langle x_1, \ldots x_n, y_1, \ldots y_n \rangle$
head S	== S 1
last S	== S #S
tail ($\langle x \rangle \frown$ S)	== S
front (S $\frown \langle x \rangle$)	== S
squash f	the function f (f: $\mathbb{N} \twoheadrightarrow$ X) squashed into a sequence
S $\upharpoonright$ s	the sequence S filtered to elements in s == squash (S $\rhd$ s)
rev S	the sequence S in reverse order

EXERCISES

1. Given the sequences of cities:

   u, v: seq CITY

   and the values

   u = $\langle$ London, Amsterdam, Madrid $\rangle$

   and

   v = $\langle$ Paris, Frankfurt $\rangle$

   write down the values of the sequences:

   u $\frown$ v
   rev (u $\frown$ v)
   rev u
   rev v
   rev v $\frown$ rev u

2. Referring to Question 1, find the value of

   squash (2 .. 4 $\lhd$ rev (u $\frown$ v))

3. Find the value of

   squash (4 .. 2 $\lhd$ rev (u $\frown$ v))

4. Find the value of

   u $\frown$ v $\upharpoonright$ { London, Moscow, Paris, Rome }

5. Find the value of

   tail (u $\frown$ v) $\frown$ front $\langle$ Moscow, Berlin, Warsaw $\rangle$

The following questions use these declarations:

[CHAR] the set of all possible characters

```
 _TEXT_______________
| stream:  seq CHAR
|____________________

 _P__________________________________________________
| ΞTEXT:
| pat?:    seq CHAR
| pos!:    ℕ
|____________________________________________________
| ((∃ before, after: seq CHAR • before ⁀ pat? ⁀ after = stream) ∧
| pos! = #before + 1)
| ∨
| (¬(∃ before, after: seq CHAR • before ⁀ pat? ⁀ after = stream) ∧
| pos! = 0)
|____________________________________________________
```

6. Explain the meaning of each line of and the overall effect of the schema *P*.
7. Write a schema *Delete* which deletes the *first* occurrence of the subsequence *pat?* from the stream and sets *pos!* to the start position of the subsequence (or to zero if the subsequence was not found).