SEQUENCE APPLICATIONS

This page is intentionally blank

SPECIFYING A STACK

This page is intentionally blank

Specifying A Stack

- A *stack* is a data structure into which elements can be added ("pushed") and from which elements can be removed ("popped")

- A stack is such that the next element to be *popped* is always the one most recently *pushed* (it is a *last-in-first-out* structure)

- We shall suppose that the elements stored on the stack are of some generic type X, and model the stack using a sequence:

$$[X] \qquad \text{any type}$$

$$bound : \mathbb{N} \qquad \text{capacity of the stack}$$

Stack[X]

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

$$\#s \leq bound$$

Specifying A Stack

- Initially the sequence (i.e. stack) is empty:

Init[X]

$Stack'[X]$

$s' = \langle\rangle$

  - Recall that the *dashed* notation suggests "initialization" is a dummy operation which has an *after* state but no *before* state (common in Z specifications)

- The *push* operation will be assumed to add a new element x, say, at the *front* of the sequence:

Push [X]

$\Delta Stack[X]$
$x? : X$

$\#s < bound$ -- precondition
$s' = \langle x? \rangle \frown s$ -- x placed at "front"

Specifying A Stack

- The *pop* operation will, similarly, remove an element from the front of the sequence:

Pop [X]

$\Delta Stack[X]$
$x! : X$

$s \neq \langle\,\rangle$ -- s should not be empty
$x! = \textbf{head}\ s$
$s' = \textbf{tail}\ s$

  - Note the precondition for *pop*

- Alternatively, we can adopt a definition for *pop* which is symmetrical with *push*:

Pop [X]

$\Delta Stack[X]$
$x! : X$

$s \neq \langle\,\rangle$ -- s should not be empty
$s = \langle x! \rangle \frown s'$ -- *x* placed at "front"

Specifying A Stack

- To find the *length* of the stack at any time we have:

HowBig

$\Xi Stack[X]$

$size! : \mathbb{N}$

$size! = \# s$

- It should be obvious that, in a correctly specified stack, the *composition* of *push* with *pop* should be a *null* operation:

Push [X] ⨾ Pop [X]

$s : \mathbf{seq}\ X$

$s' : \mathbf{seq}\ X$

$x? : X$

$x! : X$

$x? = x!$

$s' = s$

SPECIFYING AN AIRLINE ROUTE

This page is intentionally blank

Specifying an Airline Route

- The sequence of airports that a passenger passes through when flying to a particular destination may be taken to describe the air route

- Since any route will be defined in terms of the airports along it, we need the given set:

  [AIRPORT] the set of airports in the world

- There is no guarantee that, between any particular pair of airports from the given set AIRPORT, an air service will exist - such a link exists only when the airports are linked by a *connected* relation which describes just those cases where airports are connected by air services:

Air_Services

$$\_ \mathit{connected} \_ : \mathrm{AIRPORT} \leftrightarrow \mathrm{AIRPORT}$$

Specifying an Airline Route

- The operation to propose a viable route from the *originating* to the *destination* airport might be:

Propose_Route

Ξ Air_Services
from?, to? : AIRPORT
route! : **seq** AIRPORT

---

head route! = from?
last route! = to?
($\forall$ position : $\mathbb{N}$ | position $\in$ 1 .. (#route! - 1) •
route!(position) *connected* route!(position+1))
-- uses function application

- The conjuncts specify that
  - the *originating* airport is at the *start* of the sequence
  - the *destination* airport is at the *end* of the sequence
  - the route only lists airports where adjacent pairs are *connected* by an air service

Specifying an Airline Route

- The *Propose_Route* schema has no mechanism for preventing a route going through the same airport more than once (a flight could even terminate where it started)

- A schema eliminating "unnecessary" legs on a journey, might be:

No_Duplicates_Route

Propose_Route -- includes first attempt

$$(\forall i, j : \mathbb{N} \mid i \in 1 \,..\, \#\text{route!} \land j \in 1 \,..\, \#\text{route!} \bullet i \neq j \Rightarrow \text{route!}(i) \neq \text{route!}(j))$$

- Alternatively, to specify that no duplicates are allowed in a sequence we can require the inverse of the sequence to be a function:

No_Duplicates_Route

Propose_Route

$$(\text{route!})^{\sim} \in \text{AIRPORT} \pfun \mathbb{N}$$

Specifying an Airline Route

- The need to specify sequences with no duplicates is a common requirement and so Z permits the special declaration of an *injective* sequence for such cases: **iseq** X

- Using this notation, we could write:

No_Duplicates_Route

$$\Xi \text{ Air\_Services}$$
$$\text{from?}, \text{to?} : \text{AIRPORT}$$
$$\text{route!} : \textbf{iseq}\ \text{AIRPORT}$$

$$\textbf{head}\ \text{route!} = \text{from?}$$
$$\textbf{last}\ \text{route!} = \text{to?}$$
$$(\forall\ \text{position} : \mathbb{N} \mid \text{position} \in 1\ ..\ (\#\text{route!} - 1) \bullet$$
$$\text{route!}(\text{position})\ \textit{connected}\ \text{route!}(\text{position} + 1))$$

SPECIFYING PASCAL'S FILE SYSTEM

This page is intentionally blank

Specifying PASCAL's file system

- In PASCAL, a file is a *sequential* structure of any type of elements

- A file can be in one of two modes:
  - *inspection* mode, or
  - *generation* mode

- *Inspection* mode is
  - initiated by a *Reset* operation, and
  - permits *Read* operations on a file, and
  - has associated BOOLEAN *end_of_file* which is true when no further records are available for reading

- *Generation* mode is
  - initiated by a *Rewrite* operation, and
  - permits *Write* (*append*) operations on a file

Specifying PASCAL's file system

- Our model will ignore the *buffering* of data

- We shall postulate:

  [X] any type of data permitted by PASCAL language
  FILEMODE ::= inspection | generation

- The state of the file may be modelled by:

PASCAL_File[X]

file : **seq** X
still_to_read : **seq** X
mode : FILEMODE

∃ already_read : **seq** X •
already_read ⌢ still_to_read = file

- Note that *end_of_file* will be *true* when *still_to_read* = ⟨ ⟩

Specifying PASCAL's file system

- The *Reset* operation may be specified by:

Reset

ΔPASCAL_File[X] -- *why* Δ?
mode′ = inspection still_to_read′ = file file′ = file

- It should be noted that when the mode switches to *inspection*
  - the whole of the file is still to be read
  - the content of the file is not changed

Specifying PASCAL's file system

- The *Read* operation might be:

Read[X]

ΔPASCAL_File[X]
$x! : \quad X$

---

mode = inspection -- precondition
still_to_read $\neq \langle\ \rangle$ -- precondition
$\langle x! \rangle \frown$ still_to_read′ = still_to_read
mode′ = mode
file′ = file

- In the schema above the preconditions are:
  - the mode must be *inspection*
  - the part of the file still to be read must **not** be *empty*

- Postconditions demand:
  - neither mode nor file are changed, and
  - the *value returned* is taken from the *front of the file section* still to be read

Specifying PASCAL's file system

- The *Rewrite* operation might be:

Rewrite

$$\Delta PASCAL\_File[X]$$

$$mode' = generation$$
$$file' = \langle\ \rangle$$

Note that *Rewrite* makes the file empty

- The *Write* operation might be:

Write[X]

$$\Delta PASCAL\_File[X]$$
$$x? : X$$

$$mode = generation \quad \text{-- precondition}$$
$$file' = file \frown \langle x? \rangle \quad \text{-- postcondition}$$
$$mode' = mode \quad \text{-- postcondition}$$

Specifying PASCAL's file system

- Some might not be happy with the state schema for *PASCAL_file* being, apparently, based upon "reading" records

- An alternative model might be based upon the use of a *file pointer* which indexes the next record to be processed

- So ignoring buffering, as before, the state of the file may be defined by:

Alt_PASCAL_File[X]

file : **seq** X
file_pointer : $\mathbb{N}$
mode : FILEMODE

---

file_pointer $\leq$ # file + 1

Specifying PASCAL's file system

- The *Alt_Reset* operation may be specified by:

Alt_Reset

ΔAlt_PASCAL_File[X] -- why Δ?

mode′ = inspection
file_pointer′ = 1 --indexes reading position
file′ = file

- It should be noted that when the mode switches to *inspection*
  - the whole of the file is still to be read
  - the file pointer indexes the first record to be accessed (if file is empty then file-pointer is off the end of the file)
  - the content of the file is not changed

Specifying PASCAL's file system

- The *Alt_Read* operation might be:

Alt_Read[X]

ΔAlt_PASCAL_File[X]
$x!$: X

mode = inspection	-- precondition
file_pointer ≤ # file	-- precondition
$x!$ = file (file_pointer)	-- postcondition
mode′ = mode	-- postcondition
file′ = file	-- postcondition
file_pointer′ = file_pointer + 1	-- postcondition

- In the schema above the preconditions are:
  - the mode must be *inspection*
  - there must still be records unread

- Postconditions demand:
  - neither the mode nor the file are changed, and
  - the *value returned* is taken from the record indexed by the file-pointer

Specifying PASCAL's file system

- The *Alt_Rewrite* operation might be:

Alt_Rewrite

ΔAlt_PASCAL_File[X]

---

mode′ = generation
file′ = ⟨ ⟩
file_pointer′ = 1

Note that *Alt_Rewrite* makes the file *empty*

- The *Alt_Write* operation might be:

Alt_Write[X]

ΔAlt_PASCAL_File[X]
x? : X

---

mode = generation -- precondition
file_pointer = # file + 1 -- precondition
file′ = file ⌢ ⟨*x*?⟩ -- postcondition
mode′ = mode -- postcondition
file_pointer′ = file_pointer + 1 -- postcondition

This page is intentionally blank

☺

SPECIFYING AIRCRAFT SEATING SYSTEM

This page is intentionally blank

☺

INTRODUCTION

- We shall reconsider the simple system considered earlier which monitors the people who board an aircraft:

> *An aircraft has a fixed capacity and it is required to record the number of people aboard the aircraft at any time. The aircraft seats are not numbered and passengers enter the aircraft and choose seats on a first-come-first-served basis.*

- Our only basic type is [PERSON], the set of all possible persons who may, at some time, board the aircraft

- If the aircraft has a fixed capacity, we may declare:

$$| \; capacity : \mathbb{N}$$

and assign a definite value to *capacity* at an appropriate time

SYSTEM STATE

- Our previous specification was comparatively abstract and used *set* concepts. Now we shall use *sequences* to refine the specification

- The *state* of the system is given by the people on board the aircraft and the number of people on the aircraft must never exceed *capacity* (invariant property of system state)

- If we, also, assume the passenger identifications are held in a sequence (no name/identifier being held more than once) then the state may be given by:

Seq_Aircraft

$$passengers : \textbf{seq}\ \text{PERSON}$$

$$\#\, passengers \leq capacity$$
$$(\forall\, i, j : \mathbb{N}_1 \mid i \in \textbf{dom}\ passengers \land j \in \textbf{dom}\ passengers \bullet i \neq j \Rightarrow passengers\ i \neq passengers\ j)$$

SYSTEM STATE

- Note that we can also specify that the sequence should include no duplicates by alternatively writing:

Seq_Aircraft

$passengers : \mathbf{iseq}\ \text{PERSON}$ -- *injective sequence*

$\# passengers \leq capacity$

- In the earlier specification, the *set* of passengers on the aircraft was named *onboard*
  - Clearly: $onboard = \mathbf{ran}\ passengers$

- The initial state of the system may be taken to be when the aircraft is empty:

Seq_Init

Seq_Aircraft′ -- *included schema*

$passengers' = \langle\ \rangle$ -- *satisfies invariant*

SYSTEM OPERATIONS

- If *Seq_Board* is an initial version of the operation to allow a person to board the aircraft, then:

Seq_Board

ΔSeq_Aircraft -- cater for change of state
$p?$: PERSON -- person to board aircraft

$p? \notin$ **ran** *passengers* -- precondition 1
$\#$ *passengers* $<$ *capacity* -- precondition 2
$passengers' = passengers \frown \langle p? \rangle$
-- new passenger added to *end* of sequence

- We shall defer consideration of what occurs if the preconditions are not satisfied; but consideration of previous *Aircraft Seating* specifications (based on sets) will show what can be done

SYSTEM OPERATIONS

- Similarly if *Seq_Disembark* is a first version of the operation to allow a person to leave the aircraft:

Seq_Disembark

$\Delta Seq\_Aircraft$ -- cater for change of state

$p? : PERSON$ -- person to disembark

$p? \in \textbf{ran}\ passengers$ -- precondition

$(\exists\ before, after : \textbf{iseq}\ PERSON \bullet$
$passengers = before \frown \langle p? \rangle \frown after \wedge$
$passengers' = before \frown after)$
-- postcondition

- The postcondition emphasises that the sequence element modelling the disembarking person must be removed *from the correct place* in the sequence

SYSTEM OPERATIONS

- Alternatively, we might choose to exploit the standard "restriction" operators to recover a more succinct schema for the *disembark* operation:

Alt_Seq_Disembark

ΔSeq_Aircraft -- cater for change of state

$p?$: PERSON -- person to disembark

$p? \in$ **ran** *passengers* -- precondition

$passengers' = \text{squash}(passengers \rhd \{ p? \})$ -- postcondition

- We could also write the second conjunct as:

$passengers' =$ **squash** $(passengers \setminus (passengers \rhd \{p?\})$

or even

$passengers' = passengers \rhd (\textbf{ran}\ passengers \setminus \{p?\})$

ENQUIRY OPERATIONS

- An obvious enquiry is to determine the number of people currently on the aircraft

- If this operation is called *Seq_Number*, we can write:

Seq_Number

$\Xi Seq\_Aircraft$

$num\_on\_board! : \mathbb{N}$

$num\_on\_board! = \#\, passengers$

-- because no duplicates

- Define a free type: YES_OR_NO ::= yes | no then checking for a person on board is:

Seq_On_Board

$\Xi Seq\_Aircraft$ -- there can be no change of state

$p? : PERSON$ -- person to check for

$reply! : YES\_OR\_NO$

$(p? \in \mathbf{ran}\ passengers \land reply! = yes)$

-- found

$\lor\ (p? \notin \mathbf{ran}\ passengers \land reply! = no)$

-- or not found

This page is intentionally blank

☺

EXERCISES

1. Amend the given example on modelling a *stack* using a sequence so that the *top* of the stack is at the *rear* end of the sequence rather than the front.

---

2. Extend the original sequence-based model of a stack to cater for the following:

   (a) *Top* is an operation that returns the element currently at the top of the stack but leaves the stack unaltered;
   (b) the enumerated free type REPORT defines the messages which can report the state of the stack:

   $$REPORT ::= success \mid full \mid empty;$$

   if no error occurs then *success* is reported; *overflow* error occurs if *Push* is applied to a stack which is *full*; *underflow* error occurs if *Pop* is applied to an *empty* stack; after an error arises it may be assumed that further operations may be attempted.

---

3. Use the sequence concept to model a *queue*. A queue is a *first-in-first-out* structure where elements are added to one end and removed from the other.

   Assume that the queue is bounded and subject to error situations similar to those for the stack in 2 above. Define operations and messages as appropriate.

---

4. If a *file* is modelled as a sequence of *bytes* with [BYTE] representing he set of all possible bytes, then the state of the system may be modelled by the schema *Filing* :

   Filing

   file : **seq** BYTE

   (a) Give a schema for a suitable initial state
   (b) Give a schema for an operation to *insert* a *sequence* of bytes *after* a given position in the file
   (c) Give a schema for an operation to *delete* the *sequence* of bytes *within* the file, given suitable *starting* and *ending* positions
   (d) Give a schema for the operation to *copy* a sequence of bytes within the file, given suitable *starting* and *ending* positions, into an output *buffer*.

5. A *Waypoint Database* is part of an aircraft navigation system. It records the navigator's "waypoints" (i.e. intermediate locations) over which the aircraft should pass on a particular route. The *Waypoint Database* is required to hold up to a maximum of 20 waypoints stored in the order in which they should be encountered on the route.

Since the database must be ordered it could be modelled as a *sequence* of consecutive waypoints.

Suppose [WAYPOINT] is taken as the given set of all possible waypoints (which are likely to be latitude/longitude pairs; but this is left unspecified at this stage) then the start of an initial draft of the *Waypoint Database* might be:

[WAYPOINT]

max_points : ℕ
max_points = 20

If INDEX represents the allowable index values for waypoints in the database, then we could write:

INDEX = = 1 .. max_points

and, a possible state schema might have the form:

WayPointDB
db : **seq** WAYPOINT
predicate1 predicate2

where *predicate1* imposes the constraint on the database size and *predicate2* expresses an additional "safety" constraint that adjacent waypoints along a route must be distinct.

At all times, the set of waypoints already entered into the database will occupy *consecutive* sequence locations with their corresponding index values ranging from 1 to *n* where *n* is the number of waypoints stored. If a new waypoint is to be inserted at a position with index value *k*, where $1 \le k \le n$, then each waypoint with an index value within the set denoted by $k+1 \,..\, n$ is shuffled along to the next location (i.e. to the sequence location which has an index 1 greater than that of its original sequence location).

Using the above partial specification as a basis answer the following questions:

(a) Write appropriate symbolic forms for *predicate1* and *predicate2*.
(b) The insertion of a new waypoint to the database may be regarded as being one of four possible operations:

(i) adding the initial waypoint for a route to a previously empty database (*Add_Initial_To_Empty_DB*);
(ii) adding the initial waypoint for a route to a non-empty database (*Add_Initial_To_Nonempty_DB*);
(iii) adding a waypoint at a given position beween two already stored waypoints (*Add_Inter_To_Nonempty_DB*);
(iv) adding the final waypoint of a route to a non-empty database (*Add_Final_To_Nonempty_DB*).

Assuming that, when a new waypoint is inserted, subsequent waypoints are "shifted along" one place (as described previously) and that no existing waypoints can be deleted by any insertion operation, create a separate schema for each of the four insertion operations listed above. All symbolic forms must be adequately annotated in plain English but you do not need to take account of error situations.