Algebraic Specifications

The structure of an algebraic specification

SPECIFICATION NAME <Generic Parameters>

sort <name>
imports <LIST OF SPECIFICATION NAMES>

Informal description of the sort and its operations

Operation signatures setting out the names and the types of the parameters to the operations defined over the sort.

Axioms defining the operations over the sort

List specification

LIST (Elem : [Undefined → Elem])

sort List
imports INTEGER

Create	→ List
Cons (List, Elem)	→ List
Tail (List)	→ List
Head (List)	→ Elem
Length (List)	→ Integer

List specification

```
Head (Create)          = undefined
                              -- error to evaluate an empty list
Head (Cons (L, v))     = if L == Create then v else Head (L)
Length (Create)        = 0
Length (Cons (L, v))   = Length (L) + 1
Tail (Create)          = Create
Tail (Cons (L, v))     = if L == Create then Create
                         else Cons (Tail (L), v))
```

Binary Search Tree

Operation	Description
Create	Creates and empty tree
Add(Binary_tree, Elem)	Adds a node to the binary tree using the usual ordering principles.
Left(Binary_tree)	Returns the left sub-tree of the top of the tree
Data(Binary_tree)	Returns the value of the data element at the top of the tree
Right(Binary_tree)	Returns the right sub-tree of the top of the tree
IsEmpty(Binary_tree)	Returns true of the tree does not contain any elements
Contains(Binary_tree, Elem)	Returns true of the tree contains the given element

BINTREE (Elem: [Undefined → Elem, . == . → Bool, . < . → Bool])
sort Binary_tree
imports BOOLEAN

Defines a binary search tree where the data is of generic type Elem.
Build is an additional primitive constructor operation which is introduced to simplify the specification. It builds a tree given the value of a node and the left and right sub-tree.

Create	→ Binary_tree
Add(Binary_tree, Elem)	→ Binary_tree
Left(Binary_tree)	→ Binary_tree
Data(Binary_tree)	→ Elem
Right(Binary_tree)	→ Binary_tree
IsEmpty(Binary_tree)	→ Boolean
Contains(Binary_tree, Elem)	→ Boolean
Build(Binary_tree, Elem, Binary_tree)	→ Biniary_tree

```
Add (Create, E)                 = Build(Create, E, Create)
Add (B, E)                      = if E < Data (B) then Add (Left(B), E)
                                  else Add (Right (B), E)
Left (Create)                   = Create
Right (Create)                  = Create
Data (Create)                   = Undefined
Left (Build(L, D, R))           = L
Right (Build(L, D, R))          = R
Data (Build(L, D, R))           = D
IsEmpty(Create)                 = true
IsEmpty (Build (L, D, R))       = false
Contains (Create, E)            = false
Contains (Build (L, D, R), E)   = if E = D then true
                                  else if  E < D then Contains (L, E)
                                  else Contains (R, E)
```

Example 1 – Specification of Sort Coord

COORD

sort Coord
imports INTEGER, BOOLEAN

This specification defines a sort called Coord representing a Cartesian coordinate. The operations defined on Coord are **X** and **Y** which evaluate the X and Y attributes of an entity of this sort and **Eq** which compares two entities of sort Coord for equality.

Create (integer, integer)	→ Coord
X(Coord)	→ integer
Y(Coord)	→ integer
Eq (Coord, Coord)	→ Boolean

X (Create(x,y)) = x
Y(Create(x,y) = y

Eq (Create(x1, y1), Create(x2, y2)) = ((x1 == x2) and (y1 == y2))

CURSOR

sort Cursor
imports INTEGER, COORD, BITMAP

A cursor is a representation of a screen position. Defined operations are Create, Position, Translate, Change_Icon and Display.

Create (Coordinate, Bitmap)	→ Cursor
Translate(Cursor, Integer, Integer)	→ Cursor
Position(Cursor)	→ Coord
Change_Icon(Cursor, Bitmap)	→ Cursor
Display(Cursor)	→ Cursor

Translate(Create(C, Icon), xd, yd) =
Create(COORD.Create(X(C)+xd, Y(C)+yd), Icon)

Position(Create(C, Icon)) = C

Position(Translate(Cursor, xd, yd)) = COORD.Create(X(C)+xd, Y(C)+yd)

Change_Icon(Create(C, Icon), Icon2) = Create(C, Icon2)

New_List specification

Operation	Description
Create	Brings a list into existence
Cons(New_list, Elem)	Adds an element to the end of the list
Add(New_list, Elem)	Adds an element to the front of the list
Head(New_List)	Returns the first element in the list
Tail(New_list)	Return the list with the first element removed.
Member(New_list, Elem)	Returns true if element is present in the list
Length(New_list)	Returns the number of elements in the list

New_List specification

NEW_LIST (Elem : [Undefined → Elem])

---

sort New_List **enrich** List
imports INTEGER, BOOLEAN

Add (New_List, Elem) → New_List
Member (New_List, Elem) → Boolean

Head (Add (L, v)) = v
Length (Add (L, v)) = Length (L) + 1
Tail (Add (L, v)) = L

Member (Create, v) = FALSE
Member (Cons (L, v1), v2) = if v1 == v2 then TRUE else Member (L, v2)
Member (Add (L, v1), v2) = if v1 == v2 then TRUE else Member (L, v2)

Add (Create, v) == Cons (Create, v)

Queue operations

Create	Brings a queue into existence
Cons(Queue, Elem)	Adds an element to the end of the queue
Head (Queue)	Returns the element at the front of the queue
Tail (Queue)	Returns the queue minus its front element
Length (Queue)	Returns the number of elements in the queue
Get (Queue)	Returns a tuple composed of the element at the head of the queue and the queue with the front element removed

QUEUE Specification

QUEUE (Elem : [Undefined → Elem])

sort Queue **enrich** List
imports INTEGER

This specification defines a queue which is first-in, first-out data structure. It can therefore be specified as a LIST where the insert operation adds a member to the end of the queue

Get(Queue) → (Elem, Queue)

Get (Create) = (undefined, Create)
Get (Cons (Q, v)) = (Head (Q), Tail (Cons (Q, V)))

Error specification

- **Error may be tackled in three ways**
  - **Use a *special distinguished constant operation* (Undefined) which conforms to the type of the returned value.**
  - **Define operation evaluation to be a *tuple*, where *an element indicates success or failure*.**
  - **Include a *special failure section* in the specification**

List specification with Error

LIST (Elem : [Undefined → Elem])

sort List
imports INTEGER

Create	→ List
Cons (List, Elem)	→ List
Tail (List)	→ List
Head (List)	→ Elem
Length (List)	→ Integer

Head (Create) = undefined *-- error to evaluate an empty list*
Head (Cons (L, v)) = if L == Create then v else Head (L)
Length (Create) = 0
Length (Cons (L, v)) = Length (L) + 1
Tail (Create) = Create
Tail (Cons (L, v)) = if L == Create then Create else Cons (Tail (L), v))

List specification with Exceptions

LIST (Elem)

sort List
imports INTEGER

Create	→ List
Cons (List, Elem)	→ List
Tail (List)	→ List
Head (List)	→ Elem
Length (List)	→ Integer

Head (Cons (L, v)) = if L == Create then v else Head (L)
Length (Create) = 0
Length (Cons (L, v)) = Length (L) + 1
Tail (Create) = Create
Tail (Cons (L, v)) = if L == Create then Create else Cons (Tail (L), v))

exceptions

Length(L) = 0 $\Rightarrow$ **failure** (Head(L))

BOOLEAN

BOOLEAN

sort Bool

constants true: Bool;
false: Bool

not (Bool) → Bool
and (Bool, Bool) → Bool
or (Bool, Bool) → Bool
impl (Bool, Bool) → Bool
equal (Bool, Bool) → Bool

BOOLEAN

BOOLEAN

not(not(a))	= a
or(a,or(b, c))	= or(or(a,b),c)
or(a, b)	= or(b,a)
not(false)	= true
and(a, b)	= not(or(not(a), not(b))
impl(a, b)	= or(not(a), b)
equal(a, b)	= and(impl(a,b), impl(b,a))
or(a, true)	= true
or(a, false)	= a