1. Introduction to Algebras

A *homogeneous* (or *single–sorted*) algebra $\mathcal{A}$ is $\mathcal{A} = [A, \Omega]$ where A is a non–empty set which contains values of the algebra and is known as the *carrier* set. Ω is a set of operations defined over the carrier set which may include *nullary* operations (constants). The square brackets simply signify that an algebra consists of a pair of items, namely a set of values and a collection of operations. These homogeneous algebras describe small "self–contained" data types like the natural numbers or Boolean values.
As an example, an algebra, $\mathcal{A}_{Natl}$, describing the data type Natural with the familiar operations of addition ($+$) and multiplication ($\times$) is $\mathcal{A}_{Natl} = [\mathbf{N}, \{0, +, \times\}]$ where $\mathbf{N} = \{0, 1, 2, 3, \ldots\}$. (Note that we have included the value "0" (zero) as a nullary operation). Addition and multiplication are both binary operations which take two natural numbers and return a natural number as a result.

Heterogeneous or Many–sorted Algebras: The definition above is readily extended to *heterogeneous* or *many–sorted* algebras. In this case, $A = \{A_i\}$ is a family of non-empty carrier sets A_i. Such algebras can be used to represent more interesting abstract data types such as stacks and queues. For example, the structure of the algebra describing stacks of natural numbers consists of the set of stack values, the set of natural numbers, the set of boolean values and the operations which manipulate these values. The operations of a heterogeneous algebra have domain and range values drawn from the sets of carriers A_i.

Examples: Rather than committing to a representation for values, algrebras can be *abstractly* specified using EQUATIONS (rather like functions in functional programming). Here some algebraic specifications of familiar data structures:

```
SPEC  Boolean
SORT  bool
OPS
   true   :  ->  bool
  false   :  ->  bool
     not  :  bool  ->  bool
     and  :  bool  bool  ->  bool
FORALL b  :  bool
AXIOMS:
  (1)   not(true)  =  false
  (2)   not(false)  =  true
  (3)   and(true,b)  =  b
  (4)   and(b,true)  =  b
  (5)   and(false,b)  =  false
  (6)   and(b,false)  =  false
ENDSPEC

SPEC  Natural
SORT  nat
OPS
```

```
    zero  :  ->  nat
    succ  :  nat  ->  nat
    add   :  nat  nat  ->  nat
  FORALL m, n  : nat
  AXIOMS
    (1)  add(zero, n)  =  n
    (2)  add(succ(m), n)  =  succ(add(m, n))
  ENDSPEC

SPEC  Stack
USING  Natural +  Boolean                    % Carriers are Stack, Natural
SORT  stack                                  %               and Boolean
OPS                                          % Type of Interest = Stack
     init        :  ->  stack                % Signature of each Operation
     push        :  stack   nat  ->  stack
     pop         :  stack  ->  stack
     top         :  stack  ->  nat
     is-empty?   :  stack  ->  bool
     stack-error :  ->  stack
     nat-error   :  ->  nat
FORALL  s  :  stack,  n  :  nat              % universally quantified vars
AXIOMS for is-empty?:
(1)  is-empty?(init)  =  true               % this part gives 'meaning'
(2)  is-empty?(push(s,n))  =  false         % to each operation
AXIOMS for pop:
(3)  pop(init)  =  stack-error
(4)  pop(push(s,n)) =  s
AXIOMS for top:
(5)  top(init)  =  nat-error
(6)  top(push(s,n))  =  n
ENDSPEC
```

NB: For the next few weeks you need to read through chapters 8,9 and 10 of the set book, "The Construction of Formal Specifications".

EXERCISES: 1. Are the following algebras?
(i) A = natural numbers, $\Omega = \{+, -\}$
(ii) A = 'algebraic' (!) expressions of one variable x. $\Omega = \{d/dx\}$.
(iii) A = { lists of numbers, numbers, boolean }, $\Omega = \{append, member\}$
(iv) a FSM, A = states, Ω = elements of the input alphabet.

2.Are the following of type 'bool' according to the specification above? if so, why?
(i) and(not(true),false)
(ii) or(and(true,false),false)
Can you evaluate these expressions using the equations?

3.Try to change the specification of the stack into a specification of a queue.

2. What is an Algebraic Specification?

For background reading on this lecture AND next weeks, consult the book "The Construction of Formal Specifications" by Turner and McCluskey, from the Library, sections 8.5, 8.7, 8.8, 9.1 - 9.4, 10.1 - 10.5.

Above we reviewed the idea of an algebra as a set with operations. Now we shall study in detail the *equational* way of specifying algebras. Specifying algebras using equations is sometimes called a *theory presentation*. We will look at 4 diverse examples to illustrate the idea:

1. The "State" Specification below
2. The "Boolean" specification (last weeks notes)
3. The "Natural" number specification below
4. The "Stack" Specification (last weeks notes)

State specification: consider the following algebraic specification:

```
SPEC  A_State_Machine
SORT  state
OPS
   S  :   ->  state
    a  :  state  ->  state
    b  :  state  ->  state
    c  :  state  ->  state
AXIOMS
  (1)  bS  =  S     (4)  ccS  = S
  (2)  acS  =  cS   (5)  aaS  = aS
  (3)  bcS  =  cS   (6)  baS  = S
                    (7)  caS  = aS
ENDSPEC
```

Consider a FSM with input alphabet $\{a, b, c\}$, state symbols $\{S1, S2, S3\}$ and transition table:

input symbol	input state	output state	input symbol	input state	output state
a	S1	S3	c	S2	S1
b	S1	S1	a	S3	S3
c	S1	S2	b	S3	S1
a	S2	S2	c	S3	S3
b	S2	S2			

What is the connection between the algebraic specification and this FSM? In fact, if we view the FSM as a concrete algebra with values S1,S2,S3, and operations a,b,c (note these operations are TOTAL and CLOSED), the FSM is a **model** of the specification.

DEFINITIONS: A **Term Algebra** of an Alg. Spec is an algebra with *set of values* = the set of all ground terms that can be generated using the signature as a generative grammar, and *operations* = operations as in the signature of the spec. Ground Terms are those involving only constants and operators.

A **model** of an algebraic specification is an algebra which conforms to the signature and equations in that specification e.g. the C implementation of boolean type, with true = 1,

false = 0, and = &&, not = !, can be said to be a *model* of the Boolean specification. The meaning of an algebraic spec. is sometimes identified with all the "initial models" it defines. For the definition of "initial models" etc, consult the Turner and McCluskey book page 217 onwards.

EXERCISES:
1. Write down 10 values of the Term Algebra of each of the four specifications given. Use the equations to find out which of the 10 are equal to each other i.e. separate the 10 into EQUIVALENCE CLASSES using the equations.

2. Determine which of the following are syntactically legal expressions with respect to the signature of Stack

1. `push(pop(push(init,3)),5)`
2. `pop(top(push(init,1)))`
3. `top(pop(push(init,1)))`
4. `push(push(init,2),top(push(init,2)))`
5. `is-empty?(push(pop(push(init,3)),5))`

3. Use the axioms of Stack to reduce all the syntactically legal expressions of Exercise 8.5.

4. Evaluate the expressions

1. `pop(push(pop(push(init,5)),4))`
2. `pop(push(push(init,top(push(init,2))),1))`

5. Show that the following members of the Stack's term algebra are in the same equivalence class:

```
push(pop(push(init,n1)),n2)
```

```
pop(push(push(init,n2),n3))
```

3. Algebraic Specifications: Denotational vs Operational Semantics

In this section we will see that an algebra is a *model* of an algebraic specification. Giving an algebraic specification meaning by reference to its set of *initial* models is sometimes called giving it a **denotational semantics** (cf final year module CAS630). We can give an **operational semantics** to algebraic specifications as follows: Basically, we give treat its equations like re-write rules in a functional language. Note, however, that specifications must be in a certain form to satisfy this - they must be *finitely convergent* A set of rewrite rules is said to be finitely convergent if it has the following properties:

Finite termination : every sequence of rewrites from a given term terminates after a finite number of steps i.e. there are no infinite sequences of rewrites from any term. (called the Church - Rosser property)
Unique termination : every terminating sequence of rewrites from a given term stops at a unique minimal form (that is the 'normal form' of its equivalence class).

EXERCISES

1.Are the following members of the term algebra of Boolean, Stack, or Natural? If not, say why not:
(a) not(not(and(b,false))) (b) and(not,false) (c) top(pop(init))
(d) pop(push(pop(init))) (e) pop(is_empty?(push(init,2)))
(f) add(succ(zero),zero) (g) succ(add(succ(zero),add(zero,succ(zero))))

2. (i) Are the following *models* of the algebraic spec Natural given out? Are they initial models?
(a) the numbers 0,1,2,3,4,5,6 etc with normal additon '+'
(b) the binary numbers 0,1,10,11,100,101,110,111,1000,1001 etc with normal addition
(c) the numbers 1,2,3,4,5,6,7 etc with normal addition
(d) the numbers 0,1,2,3 under addition modulus 4.
(ii) What kind of term algebra has MONOID? How many equivalence classes has it? Are any of (a) - (d) models of it? Are any of (a) - (d) **initial models**?

```
SPEC  MONOID
SORT  monoid
OPS
      id    :  ->  monoid
    _  op _ :  monoid  monoid  ->  monoid
FORALL
       x,y,z  :  monoid
AXIOMS:
  (1)   id op x  =  x
  (2)   x op id  =  x
  (3)   x op (y op z)  =  (x op y) op z
ENDSPEC
```

3.For each term in EX 24 that IS a valid member of a term algebra, evaluate it using the operational semantics given by using the equations as re-write rules.

```
Example: Algebraic specification for a binary tree

   SPEC  Binary-tree
   USING  Natural  +  Boolean
   SORT binary-tree
   OPS
     empty  :  ->  binary-tree
      make  :  binary-tree   nat   binary-tree  ->  binary-tree
      left  :  binary-tree  ->  binary-tree
     right  :  binary-tree  ->  binary-tree
      node  :  binary-tree  ->  nat
 is-empty?  :  binary-tree  ->  bool
    is-in?  :  binary-tree    nat  ->  bool
 nat-error  :  -> nat
   FORALL
       l,r  :  binary-tree
       n,n1 :  nat
   AXIOMS:
   (1)  left(empty)  =  empty
   (2)  left(make(l,n,r))  =  l
   (3)  right(empty)  =  empty
   (4)  right(make(l,n,r))  =  r
   (5)  node(empty)  =  nat-error
   (6)  node(make(l,n,r))  =  n
   (7)  is-empty?(empty)  =  true
   (8)  is-empty?(make(l,n,r))  =  false
   (9)  is-in?(empty,n)  =  false
   (10) is-in?(make(l,n,r),n1)  =  IF  n == n1  THEN  true
                                   ELSE  is-in?(l,n1)
                                   or is-in?(r,n1) ENDIF
   ENDSPEC

% A Binary-tree in a Haskell-like Language

data Binary-tree = Empty | Node Binary-tree Int Binary-tree

left :: Binary-tree -> Binary-tree
left Empty = Empty
left (Node l n r) = s
is-in? :: Binary-tree -> Binary-tree
is-in? Empty = false
is-in? (Node l n r) m = true, m == n
                      = is-in?(l,n) or is-in?(r,n), otherwise

>  || implementation of ‘stack’ in ‘Haskell-like language’
> stack1 * ::= Empty1 | Push1 * (stack1 *)
>        is_empty1 :: (stack1 *) ->  bool
>        top1    :: (stack1 *) ->  *
>        pop1  :: (stack1*) -> (stack1 *)
> is_empty1(Empty1)     = True
> is_empty1(Push1 x s)  = False
> top1(Push1 x s)       = x
> pop1(Push1 x s)       = s
> || test:
>fred1 = top1(pop1( Push1 30 (Push1 20 (Push1 10 Empty1)) ))
```