Mathematics and Computer Science
2021; 6(2): 30-37
http://www.sciencepublishinggroup.com/j/mcs
doi: 10.11648/j.mcs.20210602.11
ISSN: 2575-6036 (Print); ISSN: 2575-6028 (Online)

Algebraic Specification for Input-Output in Abstract Data Types

Patricia Peratto

Normal Institute of Technical Teaching, National Administration of Public Education, Montevideo, Uruguay

Email address:
psperatto@vera.com.uy

To cite this article:
Patricia Peratto. Algebraic Specification for Input-Output in Abstract Data Types. *Mathematics and Computer Science*.
Vol. 6, No. 2, 2021, pp. 30-37. doi: 10.11648/j.mcs.20210602.11

Received: March 23, 2021; **Accepted**: April 21, 2021; **Published**: April 29, 2021

Abstract: Abstract Data Types (ADT) are used when creating software systems, in the systems design. Usually we use algebraic specification to specify the operations in a data type. The use of data types is a methodology or style of working which yields improved design when followed. In this paper we study the addition of input-output operations to the algebraic specification of operations over a data type. The motivation is that input-output operations are used in actual implementations. A specification with input-output is more complete than one without it. We need input-output operations in our programs. This justifies the addition of such operations to the specification. We consider the definition of input-output operations in functional programming in particular in Haskell. Our input-output specifications are not exactly equal to Haskell programs although some of them are likely. We specify input-output operations in a form likely to the specification of the other operations. The result is the algebraic specification of the input-output operations for many frequently used data types. The language considered is sufficiently expressive to model all these operations. The technique is illustrated by means of a variety of examples. We started from sequences, continued with sets and finish with dictionaries. The specifications we present in this paper can be used as specifications of methods of ADT definitions in object oriented programing.

Keywords: Abstract Data Types, Algebraic Specification, Input-Output

1. Introduction

Abstract Data Types (ADT) are used when creating software systems, in the systems design. The idea is to specify operations in data which are independent of the implementation.

Usually algebraic specification is used to specify the operations in a data type. This notation is likely to functional programming.

It is said that is more easy to write correct functional programs than imperative programs because the notation is more likely to mathematics. The idea is to use algebraic specification which is likely to functional languages as a specification language.

The use of data types is a methodology or style of working which yields improved design when followed. It is useful to consider a collection of operations at design time and then specify them in increasingly greater levels of detail until achieving an executable implementation [8-10, 15].

A possibility is to write first the algebraic specification (without worrying about the efficiency), continue with an implementation with recursive functions and possibly finish the implementation with iterative functions or procedures to have a more efficient implementation.

In this paper we study the addition of input-output operations to the algebraic specification of operations over a data type.

The motivation is that input-output operations are used in actual implementations. We specify input-output operations in a form likely to the specification of the other operations.

There are two chief concerns in devising a technique for specification: to define a notation which allows a rigorous definition of operations being representation independent and to learn to use such notation [8].

A good data type specification should give enough information to define the type, but without limiting the possible implementations. Algebraic specification is appropiate for data types design, since it meets this criteria.

Algebraic Specification separates the relevant detail of what from the irrelevant detail of how. We use the term

Abstract Data Type to refer to a class of objects defined by a representation independent specification.

We supply the functionality of the operations giving: name of operation, domain and range. But to rely on one's intuition about the meaning of names is not enough. There are isomorphic functionalities, as by example between Stacks and Queues. We need to specify the semantics of the operations of the type to distinguish them.

Then an algebraic specification of an abstract type consists of a syntactic specification providing names, domains and ranges and a semantic specification consisting on a set of equations defining the meaning of the operations by stating their relationship to one another.

Dealing with input and output in TADs as in functional languages (by example Haskell [16-17]) has as a problem that requires side effects. Mathematical functions always have to return the same results for the same arguments. Any IO library should provide operations to read and write basic types like Integers, Strings, etc. We want this operations to be functions but they are not. An operation that reads a String from the keyboard cannot be a function since it will not return the same String every time.

We can not think of things like "read a String from the keyboard" or "print an Integer" as functions in the pure mathematical sense. We give to them other name (used in Haskell): Actions. And they have special types (in our case the following):

input_String: input String
output_integer: Integer output

We put actions together writting them between {, } and ending each in a semicolon. Instead of <- used in Haskell we use let.

Our input-output specifications are not exactly equal to Haskell programs although some of them are likely. They are equations that must be satisfied and are not necessarily applied to constructors although sometimes they are.

In [18] is summarized what abstraction means. Some of the names used for this concept are: Abstraction, Modularity, Encapsulation, Information Hiding, Separation of Concerns. As they say in the early days of computing, a programming language came with built-in types (as integers, booleans, strings, etc.), built in procedures for input-output and users could define their own procedures. A major advance in software development was the idea of abstract types: that one could design a programming language to allow user-defined types too. This idea came out of the work of many researchers, notably Dahl, Dijkstra, Hoare, Parnas, Liskov and Guttag [1-10, 14-15]. The key idea of data abstraction is that a type is characterized by the operations you can perform on it. What made abstract types new and different was the focus on operations, the user of the type would not need to worry about how its values were actually stored, all that matters are the operations. Critically, a good abstract data type should be representation independent. Changes in representation have no effect on code outside the abstract type itself. For example, the operations offered by Stack are independent of whether it is represented as a linked list or as an array.

The rest of the paper is organized as follows: in section 2 we present the constructions we use in the specifications. In section 3 the specification of sequences, in section 4 the specification of sets and in section 5 the specification of dictionaries. Section 6 present the Conclusion and we finish in section 7 suggesting further work..

2. Algebraic Specification and Input-Output

The constructions we allow to use in our algebraic specification are the following:

1. parameters
2. if-then-else
3. let in expressions
4. boolean expressions
5. recursion
6. sequencing of instructions inside {}
7. return

The constructions 1, 2, 4 and 5 were used in [8]. Besides if-then-else we use if-then. We add to them a let operation whose use is like in functional programs and sequencing that is used in some input and in the output specifications. Finally we have a return operation that finishes a function.

We consider the specification of two kinds of input operations:

1) we read a specific number of elements
2) we read until the sentinel EOF is at the input

For the output, we traverse the ADT printing the elements until reaching an empty data-type.

We will specify three kinds of Collections: Sequences, Sets and Dictionaries. Inside Sequences we consider Stacks and Queues. As Dictionaries we consider Binary Search Trees and Closed Hash tables.

Sequencing of operations is used like in Haskell [16-17] where input-output operations are defined by sequences of actions. In Haskell there are sequences of statements introduced by the do notation. We skip the do keyword. Besides input-output operations we make use of other operations proper of the ADT we are considering. We define sequencing putting the operations inside brackets and finishing by;.

We classify the operations we specify in:

1) constructors
2) observers
3) selectors
4) extenders

Constructors are used to define elements in the ADT, observers return a boolean, selectors return parts of the objects we are considering, extenders are the other operations.

In what follows we can see the operations input_item and output_item as polymorphic functions where item is a type variable or as a monomorphic function where item is a concrete type. More about polymorphism and monomorphism in [13].

3. Specification of Sequences

A sequence is an ordered collection of elements. There is a first element, a second, etc. and each of them occupies a definit position in the sequence. Inside this family there are different ADT's. We will study the specification of input-output for the following sequences: Stacks and Queues.

3.1. Specification of Stacks

A stack is a special kind of sequence in which all insertions and deletions take place at one end, called the top. Other name for a stack is LIFO or last-in-first-out.

Let us begin with the algebraic specification of Stacks without input-output. From now on, we use sometimes pattern matching when specifying the semantics of operations.

```
ADT Stack(item)
Syntax:
constructors:
NewStack:      Stack
Push: (Stack      item)      Stack
observers
IsNewStack: Stack      Boolean
selectors:
Pop: Stack      Stack
Top: Stack      item
Semantics:
IsNewStack(NewStack)=true
IsNewStack(Push(s,i))=false
Pop(t) precondition not(IsNewStack(t))
Pop(Push(s,i))=s
Top(t) precondition not(IsNewStack(t))
Top(Push(s,i))=i
```

We indicate preconditions that must hold before applying an operation. We specify the operations only when holds the precondition. An alternative followed in [8] is to have a value undefined that is returned when an operation is applied to a value that does not fit the precondition, by example Top(NewStack).

We consider the following kind of operations: constructors, whose semantics is primitive and is not given explicitly but by the application of other operations to elements constructed by their application. Observers that give boolean values when applied to elements constructed by application of constructors and selectors that return the parts from which are constructed the elements.

3.1.1. Specification of Input in Stacks

Input operations are applied to (). Usually input operations read a specific number of values or until a sentinel is at the input. We assume the input-output operations of primitive types like Integers, Char, Bool, Strings are given.

Consider first the case in which we read a predefined number of elements. We use functions input_int and input_item to read an integer and an item respectively. We specify the input by

```
Read_Stack: input      Stack
Read_Stack() = let i=input_int() in Read_Stack_Value(i)
input_item: input      item
Read_Stack_Value: int      Stack
Read_Stack_Value(0)=NewStack
Read_Stack_Value(x+1)= let s=Read_Stack_Value(x) in
let i=input_item() in Push(s,i)
```

Another possibility is to read values until a sentinel is read. The specification in this case is

```
Read_Stack: input      Stack
Read_Stack() = Read_Stack_Value(NewStack)
Read_Stack_Value: Stack      Stack
Read_Stack_Value(s) = let i=input_item() in
if (i==EOF) then s
else Read_Stack_Value(Push(s,i))
```

3.1.2. Specification of Output in Stacks

The case of the output uses sequencing. We print the elements of the stack from top to bottom.

```
output_item: item      output
Print_Stack: Stack      output
Print_Stack(s) = if (not(IsNewStack(s)))
then {output_item(Top(s));
Print_Stack(Pop(s))}
```

Print_Stack can be specified also by pattern matching in the constructors. In the empty case we use the return operation.

```
Print_Stack: Stack      output
Print_Stack NewStack = return
Print_Stack (push s i) = {output_item(i); Print_Stack(s)}
```

3.2. Specification of Queues

A queue is another special kind of sequence, where items are inserted at one end (the rear) and deleted at the other end (the front). Another name for a queue is FIFO or first-in-first-out.

Let us begin with the algebraic specification of Queues without input-output.

```
ADT Queue(item)
Syntax:
constructors:
NewQueue:      Queue
Add_at_back: (Queue      item)      Queue
observers:
IsNewQueue: Queue      Boolean
selectors:
Front: Queue      item
Delete: Queue      Queue
extender:
Append: (Queue      Queue)      Queue
Semantics:
IsNewQueue(NewQueue)=true
IsNewQueue(Add_at_back(s,i))=false
Front(t) precondition not(IsNewQueue(t))
Front(Add_at_back(s,i)) = if IsNewQueue(s) then i
else Front(s)
Delete(t) precondition not(IsNewQueue(t))
Delete(Add_at_back(s,i)) = if IsNewQueue(s) then
NewQueue
else Add_at_back(Delete(s),i)
```

```
Append(q,NewQueue)=q
Append(q,Add_at_back(s,i))=Add_at_back(Append(q,s),i)
```

We have added a new kind of operation: an extender. In this example we define *Append* which returns a Queue compossed from another two.

3.2.1. Specification of Input in Queues

We consider the same two cases that in the case of Stacks, i.e. to read an specified number of elements and to read until is input a sentinel.

Consider first the case in which we read a predefined number of elements. We use functions input_int and input_item as before. We specify the input by

```
Read_Queue: input      Queue
Read_Queue() = let i=input_int() in Read_Queue_Value(NewQueue,i)
input_item: input      item
Read_Queue_Value: (Queue      int)      Queue
Read_Queue_Value(s,0) = s
Read_Queue_Value(s,x+1) = let i=input_item() in
let z = Add_at_back(s,i) in Read_Queue_Value(z,x);
```

The specification of the case in which we read values until a lookout is:

```
Read_Queue: input      Queue
Read_Queue() = Read_Queue_Value(NewQueue)
Read_Queue_Value: Queue      Queue
Read_Queue_Value(s) = let i=input_item() in
if (i==EOF) then s
else Read_Queue_Value(Add_at_back(s,i))
```

3.2.2. Specification of Output in Queues

In the case of the output of a queue we print the elements from left to right using sequencing.

```
output_item: item      output
Print_Queue: Queue      output
Print_Queue(s) = if (not(IsNewQueue(s)))
let i=Front(s) in {output_item(i);
Print_Queue(Delete(s));}
```

4. Specification of Sets

This collection represent the mathematical notion of Set. There are no repeated elements and there is no an order relation between the elements. The primitive operations of this ADT are:

```
ADT Set(item)
Syntax:
constructors:
NewSet:      Set
Add: (Set      item)      Set
observers:
IsNewSet: Set      Boolean
Belongs: (Set      item)      Boolean
extenders:
Delete: (Set      item)      Set
Union: (Set      Set)      Set
Intersection: (Set      Set)      Set
Difference: (Set      Set)      Set
Semantics:
IsNewSet(NewSet)=true
IsNewSet(Add(s,i))=false
Belongs(NewSet,i)=false
Belongs(Add(s,i),j) = if (i==j) then true
else Belongs(s,j)
Delete(NewSet,i)=NewSet
Delete(Add(s,i),k) = if (i==k) then s
else Add(Delete(s,k),i)
Union(NewSet,s)=s
Union(Add(s,i),t)=Add(Union(s,t),i)
Intersection(NewSet,s)=NewSet
Intersection(Add(s,i),t) = if Belongs(t,i) then Add(Intersection(s,t),i)
else Intersection(s,t)
Difference(NewSet,s)=NewSet
Difference(Add(s,i),t) = if Belongs(t,i) then Difference(s,Delete(t,i))
else Add(Difference(s,t),i)
```

4.1. Specification of Input in Sets

Consider first the case in which we read a predefined number of elements. We use functions input_int and input_item as before. We specify the input by

```
Read_Set: input      Set
Read_Set() = let i=input_int() in Read_Set_Value(NewSet,i)
input_item: input      item
Read_Set_Value: (Set      int)      Set
Read_Set_Value(s,0)=s
Read_Set_Value(s,x+1) = let j=input_item() in
if Belongs(s,j) then Read_Set_Value(s,x+1);
else Read_Set_Value(Add(s,j),x)
```

We consider when is input an element that already belongs to the set. In this case the element is not added again and we don't decrease the number of elements to be read.

The specification of the case in which we read values until a lookout is:

```
Read_Set: input      Set
Read_Set() = Read_Set_Value(NewSet)
Read_Set_Value: Set      Set
Read_Set_Value(s) = let i=input_item() in
if (i==EOF) then s
else if Belongs(s,i) then Read_Set_Value(s)
else Read_Set_Value(Add(s,i))
```

4.2. Specification of Output in Sets

```
output_item: item      output
Print_Set: Set      output
Print_Set(s) = if (not(IsNewSet(s))) then
let s=Add(t,i) in {output_item(i);
Print_Set(t);}
```

5. Specification of Dictionaries

This family defines collections whose elements have an attribute that is a key that identify them. We will study the specification of input-output for the following dictionaries: Binary Search Trees (BST) and Closed Hash tables.

5.1. Specification of Binary Search Trees

A binary search tree is a dictionary whose elements are ordered by some linear order. Is a binary tree in which all the elements in the left subtree of a node are smaller that the element at the node and all the elements in the right subtree of a node are greater that the element at the node. We call this the search property.

Let us begin with the algebraic specification of Binary Search Trees without input-output. We have an extender AddElem that adds an element to a binary search tree if the element does not belong to the tree in which case

the element is not added. The addition of the elements satisfies the search property. BST's are constructed applying NewBST that gives an empty BST and Add that given two BST and an element returns a BST.

```
ADT BST(item)
Syntax:
constructors:
NewBST:     BST
Add: (BST     BST     item)     BST
observers:
IsNewBST: BST     Boolean
Member: (BST     key)     Boolean
selectors:
Key: item     key
Root: BST     item
Left: BST     BST
Right: BST     BST
Min: BST     item
Max: BST     item
Find: (BST     key)     item
extenders:
Modify: (BST     item)     BST
AddElem: (BST     item)     BST
Delete: (BST     key)     BST
Semantics:
IsNewBST(NewBST)=true
IsNewBST(Add(l,r,i))=false
Member(NewBST,i)=false
Member(Add(l,r,i),j)) = if (Key i==j) then true
else if (Key i<j) then Member(r,j)
else Member(l,j)
Root(t) precondition not(IsNewBST(t))
Root(Add(l,r,i))=i
Left(t) precondition not(IsNewBST(t))
Left(Add(l,r,i))=l
Right(t) precondition not(IsNewBST(t))
Right(Add(l,r,i))=r
Min(t) precondition not(IsNewBST(t))
Min(Add(l,r,i)) = if(IsNewBST(l)) then i
else Min(l)
Max(t) precondition not(IsNewBST(t))
Max(Add(l,r,i)) = if(IsNewBST(r)) then i
else Max(r)
Find(t,j) precondition Member(t,j)
Find(Add(l,r,i),j) = if (Key i==j) then i
else if (Key i<j) then Find(r,j)
else Find(l,j)
Modify(t,j) precondition Member(t,Key j)
Modify(Add(l,r,i),j)) = if (Key i==Key j) then Add(l,r,j)
else if (Key i<Key j) then Add(l,Modify(r,j),i)
else Add(Modify(l,j),r,i)
AddElem(NewBST,i)=Add(NewBST,NewBST,i)
AddElem(Add(NewBST,NewBST,i),j) = if (i<j) then
Add(NewBST,Add(NewBST,NewBST,j),i)
else if (i>j) then Add(Add(NewBST,NewBST,j),NewBST,i)
else Add(NewBST,NewBST,i)
AddElem(Add(NewBST,r,i),j) = if (i<j) then
Add(NewBST,AddElem(r,j),i)
else if (i>j) then Add(AddElem(NewBST,j),r,i)
else Add(NewBST,r,i)
AddElem(Add(l,NewBST,i),j) = if (i<j) then
Add(l,AddElem(NewBST,j),i)
else if (i>j) then Add(AddElem(l,j),NewBST,i)
else Add(l,NewBST,i)
AddElem(Add(l,r,i),j) = if (i<j) then Add(l,AddElem(r,j),i)
else if (i>j) then Add(AddElem(l,j),r,i)
else Add(l,r,i)
Delete(t,j) precondition Member(t,j)
Delete(Add(NewBST,NewBST,i),j) = if (i==j) then
NewBST
else Add(NewBST,NewBST,i)
Delete(Add(NewBST,r,i),j) = if (i>j) then Add(NewBST,r,i)
else if (i<j) then Add(NewBST,Delete(r,j),i)
else r
Delete(Add(l,NewBST,i),j) = if (i<j) then Add(l,NewBST,i)
else if (i>j) then Add(Delete(l,j),NewBST,i)
else l
Delete(Add(l,r,i),j) = if (i<j) then Add(l,Delete(r,j),i)
else if (i>j) then Add(Delete(l,j),r,i)
else let d=Min(r) in Add(l,Delete(r,d),d)
```

The Key function depends on the representation and is not defined.

5.1.1. Specification of Input in BST

Consider first the case in which we read a predefined number of elements. We specify the input by

```
Read_BST: input     BST
Read_BST() = let i=input_int() in
Read_BST_Value(NewBST,i,0)
input_item: input     item
Read_BST_Value: (BST     int     int)     BST
Read_BST_Value(s,0,0) = s
Read_BST_Value(s,0,k) = Read_BST_Value(s,k,0)
Read_BST_Value(s,x+1,k) = let i=input_item() in
{if Member(s,Key i)) then Read_BST_Value(s,x+1,k+1);
else Read_BST_Value(AddElem(s,i),x,k);}
```

We sum in parameter k the number of repetitions in the input to read after again this number of elements. We repeat this process until there are not more repeated elements at the input. We use sequencing to read x elements as part of the input of the x+1 elements.

The specification of the case in which we read values until a lookout is:

```
Read_BST: input      BST
Read_BST() = Read_BST_Value(NewBST)
Read_BST_Value: BST      BST
Read_BST_Value(s) = let i=input_item() in
if (Key i==EOF) then s
else if (Member(s,Key i)) then Read_BST_Value(s)
else Read_BST_Value(AddElem(s,i))
```

in this case, if we read an element that is already in the tree it is not added again.

5.1.2. Specification of Output in BST

In the case of the output of a BST we print the elements in inorder.

```
output_item: item      output
Inorder_BST: BST      output
Inorder_BST(s) = if (not(IsNewBST(s))) then
{Inorder_BST(Left(s));
output_item(Root(s));
Inorder_BST(Right(s));}
```

A definition for Inorder_BST by pattern matching is:

```
Inorder_BST: BST      output
Inorder_BST(NewBST) = return
Inorder_BST(Add(l, r, i)) = {Inorder_BST(l);
output_item(i);
Inorder_BST(r);}
```

5.2. Specification of Closed Hash Tables

When an application needs to store information we convert the key in an index that indicates the position at which is stored the information. If we want to add an element and the position is occupied we search sequentially the first position free. In the same way we search an element when we want to know if is in the table, to delete it or to modify it.

Let us begin with the algebraic specification of Closed Hash.

The operations Select, FHash, SetFree and Key depend on the representation and we don't give definitions for they.

```
ADT Hash(key, item)
Syntax:
constructors:
Empty: item
NewHash:      Hash
Insert: (Hash      key      item)      Hash
observers:
IsFull: Hash      Boolean
IsFree: item      Boolean
Member: (Hash      key)      Boolean
selectors:
Key: item      key
FHash: item      key
Select: (Hash      key)      item
Element: (Hash      key)      item
extensors:
Add: (Hash      item)      Hash
SetFree: (Hash      key)      Hash
Delete: (Hash      key)      Hash
Modify: (Hash      key      item)      Hash
semantics:
IsFree(Empty) = true
IsFree(o) = false
Mem(h,i,k,m) = let j=Select(h,i) in if (not(IsFree(j)) and
Key(j)==k)
then true
else if (not(m-1==i)) then Mem(h,(i+1)%N,k,m)
else false
Member(NewHash,k) = false
Member(h,k) = if (k==0) Mem(h,k,k,N)
else Mem(h,k,k,k)
Element(h,k) precondition Member(h,k)
Elem(h,i,k) = let j=Select(h,i) in if (not(IsFree(j)) and
Key(j)==k) then j
else Elem(h,(i+1)%N,k)
Element(h,k) = Elem(h,k,k)
IsF(h,i,k) = let s=Element(h,i) in if (i<k) then not(IsFree(s))
and isF(h,i+1,k)
else true
IsFull(h) = IsF(h,0,N)
Add(h,k) precondition not(Member(h,Key(k)))
Add'(h,i,k) = let j=Select(h,i) in if (IsFree(j)) then
Insert(h,i,k)
else Add'(h,(i+1)%N,k)
Add(h,k) = if (not(IsFull(h))
then let i=FHash(k) in Add'(h,i,k)
Delete(h,k) precondition Member(h,k)
Del(h,i,k) = let j=Select(h,i) in if (not(IsFree(j)) and
Key(j)==k)
then SetFree(h,i)
else Del(h,(i+1)%N,k)
Delete(h,k) = Del(h,k,k)
Modify(h,k,s) precondition Member(h,k)
Mod(h,i,k,s) = let j=Select(h,i) in if (not(IsFree(j)) and
Key(j)==k)
then Add(Delete(h,k),s)
else Mod(h,(i+1)%N,k,s)
Modify(h,k,s) = Mod(h,k,k,s)
```

Add adds an element that does not belong to the hash table while Insert is a constructor. An element is added in case it does not belong to the table and the table is not full. The definitions of the functions are not given depending on the constructors as in the cases before. They are recursive functions given by pattern matching and we know some of they finish (Element, Delete, Modify) by the precondition in terms of Member.

5.2.1. Specification of Input in Closed Hash Tables

Consider first the case in which we read a predefined number of elements. We use functions input_int and input_item as before. We will specify three ways of adding elements: as in the case of Sets, as in the case of BST and until a lookout.

```
Like Sets:
Read_Hash: input    Hash
Read_Set() = let i=input_int() in Read_Hash_Value(NewHash,i)
input_item: input    item
Read_Hash_Value: (Hash x int)    Hash
Read_Hash_Value(s,0) = s
Read_Hash_Value(s,x+1) = let j=input_item() in
if Member(s,j) then Read_Hash_Value(s,x+1)
else Read_Hash_Value(Add(s,j),x)
Like BST:
Read_Hash: input    Hash
Read_Hash() = let i=input_int() in Read_Hash_Value(NewHash,i,0)
input_item: input    item
Read_Hash_Value: (Hash    int    int)    Hash
Read_Hash_Value(h,0,0) = h
Read_Hash_Value(h,0,k) = Read_Hash_Value(h,k,0)
Read_Hash_Value(h,x+1,k) = let i=input_item() in
if Member(h,i) then Read_Hash_Value(h,x+1,k+1)
else Read_Hash_Value(Add(h,i),x,k)
Until a lookout:
Read_Hash: input    Hash
Read_Hash() = Read_Hash_Value(NewHash)
Read_Hash_Value: Hash    Hash
Read_Hash_Value(s) = let i=input_item() in
if (i==EOF) then s
else if (Member(s,Key(i))) then Read_Hash_Value(s)
else Read_Hash_Value(Add(s,i))
```

5.2.2. Specification of Output in Hash Tables

The operation Set_of_keys below constructs from a Hash Table the Set of their keys.

```
Set_of_keys: Hash    Set
Set_of_keys(NewHash) = NewSet
Set_of_keys(Add(s,i)) = Add(Set_of_keys(s),Key(i))
output_item: item    output
Output_Hash: Hash    output
Output_Hash(h) = let s=Set_of_keys(h) in Output_using_key(s,h)
Output_using_key: Set    Hash    output
Output_using_key(s,h) = if (not(IsNewSet(s))) then
let s=Add(s',k) in {output_item(Element(h,k));
Output_using_key(s',h);}
```

Output_using_key print the elements whose key is in the set.

6. Conclusions

Algebraic specification supports input-output operations in a functional framework. We model input-output as well as the other operations usually considered in algebraic specification of ADTs. The language considered is sufficiently expressive to model all these operations. The technique is illustrated by means of a variety of examples. We started from sequences and continued with sets and dictionaries. The specifications we present in this paper can be used as specifications of methods of ADT definitions in object oriented programming.

Further Work

Remains to study if holds the completeness of the Algebraic Specification of input-output [11-12].

References

[1] Dahl, O.-J., *SIMULA an Algol-Based Simulation Language*. Communications of the ACM, Vol 9, Number 9, September 1966.

[2] Dijkstra, E. W.. *Notes on structured programming*. In Structured Programming, Academic Press, New York, 1972.

[3] Hoare, C. A. R., *Proof of correctness of data representations.* Acta Informatica 1 (1972), 271-281.

[4] Parnas D. L., *A Technique for Software Module Specification with Examples*. Communications of the ACM, Vol 15, Number 5, May 1972.

[5] Hoare, C. A. R., and Wirth, N. *An Axiomatic definition of the programming language Pascal*, Acta Informatica 2 (1973) 335-355.

[6] Liskov B., Zilles S., *Programming with Abstract Data Types*, Proceedings of the ACM Sigplan Symopsium on very high level languages, pag 50-59, 1974.

[7] Parnas D. L. A., *The influence of Software Structure on Reliability*, ACM Sigplan Notices, Vol 10, Issue 6, April 1975.

[8] J. V. Guttag, E. Horowitz, D. R. Musser, *The Design of Data Type Specifications*, Proceedings of the 2nd International Conference on Software Engineering, pp 414-420, San Francisco, California, USA, 1976.

[9] J. V. Guttag, E. Horowitz, D. R. Musser, *Abstract Data Types and Software Validation,* Information Sciences Institute, University of Southern California, August 1976, ARPA ORDER No 2223.

[10] J. Guttag, *Abstract Data Types and the Development of Data Structures,* Communications of the ACM, June 1977, Vol. 20, Number 6.

[11] J. A. Goguen, J. W. Thatcher, E. G. Wagner, J. B. Wright, *Initial Algebra Semantics and Continuous Algebras*, ACM, 1977.

[12] J. A. Bergstra and J. V. Tucker, *The Completeness of the Algebraic Specification Methods for Computable Data Types*, Information and Control 54, 186-200 (1982).

[13] Cardelli, L., Wegner, P., *On Understanding Types, Data Abstraction, and Polymorphism*, Computing Surveys, Vol. 17, No 4, December 1985.

[14] Dahl, O.-J., *The Birth of Object Orientation, The Simula Languages*, Software Pioneers, Springer 2002.

[15] J. Guttag, *Abstract Data Types, Then and Now*, M Broy, E. Denert (Eds): Software Pionners, Springer-Verlag Berlin Heidelberg, 2002.

[16] Hal Daume III, *Yet Another Haskell Tutorial*, Copyright (c) Hal Daume III, 2002-2006.

[17] *Haskell/Print* version from Wikibooks, the open-content textbooks collection. June 2018.

[18] Reading 10: *Abstract Data Types*, course 6.031 Software Construction, MIT, Spring 2018, http://web.mit.edu/6.031/www/sp18/classes/10-abstract-data-types/