Technical Report CS78003-R

Algebraic Specification of "Complex" Data Types

Johannes J. Martin

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

June 1978

Algebraic Specification of 'Complex' Data Types

Johannes J. Martin
Virginia Polytechnic Institute and State University

Abstract

This paper discusses algebraic data type specifications that employ product sets as carriers. This leads to simplifications for certain data types. In particular, the method permits the specification of a complex type by the straight forward translation of an intuitive model of the type into a formal definition, provided that the model is conceived in terms of other well defined types. The concept is illustrated by two examples.

Keywords

Abstract data types, data types, complex types, algebraic specification

Computing Reviews Categories: 5.24, 4.22

Introduction

Heterogeneous algebras first described by [Birkhoff and Lipson 70] have been employed for the definition of data types by [Guttag 75]. According to Guttag, a data type is a pair

$$T = (V, F)$$

where V is a set of domains and F is a set of mappings. One of the members of V is distinguished and will be called the carrier of the type T. It will be denoted by U.

The mappings in F are of the form

$$Fj: \quad Wj1 \; X \; Wj2 \; X \; . \; . \; . \; X \; Wjn \quad -> Vk$$

where the Wji are members of the set Wj. Wj is a subset of V, Vk is a member of V. Wj and Vk are chosen such that the carrier U is a member of the union of Wj and {Vk}.

In this paper, U is considered to be a product set of the form

$$U = U1 \; X \; U2 \; X \; . \; . \; . \; X \; Um.$$

This leads to some simplifications of the definitions of certain data structures.

Terminology and notations

If the carrier of a data type is a product set of more than one factor, the type will be called complex. Otherwise it will be called simple. Functions Fi of complex types that yield values in U are, of course, m-tuples of functions

Fi.1, Fi.2, . . . , Fi.m

such that Fi.j(w) = uj where uj is in Uj and w is a point in the domain of Fi.

U and Fi will be called the complex carrier and the complex functions respectively and the Uj and Fi.j will be called their components.

Axioms will have the form described by Guttag except that they may be stated on the component level as well as on the complex level. (Since a knowledge of the precise definition of Guttag's form is not necessary for the understanding of this paper, a desciption is ommitted.) Note, though, that expressions stated on the complex level can always be expanded down to the component level. If all axioms of a data type can be expressed on the complex level then a complex carrier is obviously not necessary.

In order to enhance readability, components of complex quantities will be separated by periods, but a list of components will not be enclosed in parentheses; periods will have precedence over commas.

Two examples

I) The first example presented to illustrate the concept of complex types is a linear list that permits insertions, deletions, and retrievals of items at some implicitly defined 'place of interest'. In the sequel, this structure will simply be called a 'list'. The syntax of the operations and an intuitive description of their semantics is given next. The domains involved are the carrier 'list', the auxiliary set 'item', and the boolean set {true,false}.

TABLE I.1 Syntax and informal description of lists

1) A new list is created by
   CREATE -> list;

2) The empty list is recognized by the test function
   ISMT(list) -> {true,false};

3) An item is inserted immediately preceding the place of interest by
   INSERT(list,item) -> list;

4) The trailing end of a list starting at the place of interest can be extracted by
   TAIL(list) -> list;

5) The item at the place of interest can be retrieved by
   READ(list) -> item;

6) The item at the place of interest can be deleted from the list by
   DLT(list) -> list;

7) The place of interest can be advanced toward the end of the list (one item at the time) by
   NEXT(list) -> list;

8) The place of interest can be reset to the beginning of the list by
   RESET(list) -> list.

With the conventional use of axiomatic definitions, it is difficult to keep track of the 'place of interest' without resorting to hidden operations and the additional type 'integer'. However, the task of defining this structure becomes rather simple if the elements of the carrier have two components both of which are queues. Intuitively, the list is the concatenation of the two queues. The place of interest is the front of the second queue.

The following informal description of the list operations in terms of the constituent queues, q1 and q2, may assist with the understanding of the formal axioms given thereafter.

TABLE I.2 Description of lists in terms of queues

1) CREATE -> list
creates a pair of empty queues;

2) ISMT(list) -> {true,false}
returns 'true' if both q1 and q2 are empty;
otherwise it returns false;

3) INSERT(list,item) -> list
queues an item to the rear of q1;

4) TAIL(list) -> list
returns empty.q2;

5) READ(list) -> item
retrieves the front of q2;

6) DLT(list) -> list
deletes the front of q2;

7) NEXT(list) -> list
reads and deletes the first item of q2
and queues it to q1;

8) RESET(list) -> list
queues all elements of q2 onto q1,
sets q2 = q1 and q1 = empty.

For the specification of the axioms, the following names will be used to denote the components of lists and list functions:

TABLE I.3 Components of lists and list operations

D0. L = X.Y is a member of 'list',
D1. CREATE = CT1.CT2,
D2. INSERT = IN1.IN2,
D3. TAIL = TL1.TL2,
D4. DLT = DT1.DT2,
D5. NEXT = NX1.NX2,
D6. RESET = RS1.RS2.

TABLE I.4 Axioms: for all L = X.Y in list and I in item

```
L1.   CT1 = CT2 = CT;

L2.   ISMT(CREATE)       =  true;
L3.   ISMT(INSERT(L,I))  = false;
L4.   ISMT(RESET(L))     = IF ISMT(L)
                              THEN true ELSE false;

L5.   IN1(X.Y, I)        = IN1(X.X, I) //denoted IN(X,I)//;
L6.   IN2(X.Y, I)        = Y;

L7.   TAIL(X.Y)          = CT.Y;

L8.   READ(X.IN(Y,I))    = IF ISMT(TAIL(X.Y))
                              THEN  I
                              ELSE  READ(X.Y);
L9.   READ(X.CT)         = error;

L10.  DT1(X.Y)           = X;
L11.  DT2(X.IN(Y,I))     = IF ISMT(TAIL(X,Y))
                              THEN  Y
                              ELSE  IN(DT2(X.Y), I);
L12.  DT2(X.CT)          = error;

L13.  NEXT(L)            = DLT(INSERT(L,READ(L)));

L14.  RESET(X.Y)         = IF ISMT(TAIL(X.Y))
                              THEN  Y.X
                              ELSE  RESET(NEXT(X.Y));
```

L15. As a general rule, a function returns 'error' if any of its arguments equals 'error'.

Theorem: The above axiom system is complete.

The proof of this theorem is greatly simplified by first proving the following

Lemma:

Every valid list is of the form A.B where A and B are members of the set Z1 defined as follows:

1) CT //the empty queue// is an element of Z1
2) if H is an element of Z1 then so is IN(H,T) where T is an item. LL0, the set of all lists, in addition contains elements of the form error.error, error.A, A.error with A in Z1.

Proof of the lemma (by induction):

Base case: The lemma is correct for the value of CREATE.

Inductive hypothesis: L = A.B is a valid list contained in LL0. //Because of axiom 15, invalid lists are of no concern//

Inductive step: The functions that lead from L to a new list L' are

(a) DLT(L), (b) INSERT(L,T), (c) NEXT(L), (d) RESET(L), and (e) TAIL(L).

Thus, it must be proved that all these functions map LL0 into itself.

(a) L' = DLT(L) = DT1(L).DT2(L) = A.DT2(A.B);
IF B = CT THEN L' is invalid //axiom 12//
ELSE //i.e. if B = IN(H,T)//
IF H = CT //i.e. IF ISMT(TAIL(A.H))//
THEN L' = A.CT //axiom 11//
ELSE L' = A.IN(DT2(A.H), T);
In all three cases, L' is in LL0.

(b) L' = INSERT(L,T) = INSERT(A.B, T) = IN(A,T).B

(c) L' = NEXT(L) = DLT(INSERT(L,READ(L)))
L' is in LL0 because of (a) and (b).

(d) L' = RESET(L) = RESET(A.B) //axiom 14//
IF B = CT THEN L' = B.A obviously is in LL0.
ELSE L' = RESET(NEXT(A.B)) is in LL0
since repeated applications of NEXT will
lead to the list C.CT with C in Z1;
then RESET(C.CT) = (CT.C).

(e) L' = TAIL(L) = TAIL(A.B) = CT.B
End of proof of lemma.

Proof of the theorem:

It remains to be shown that READ(L) and ISMT(L) are defined for all valid lists that satisfy the lemma.
For READ(L), this follows directly from the axioms 8 and 9.
For ISMT(L), axiom 2 covers L = CT.CT,
axiom 3 covers L = IN(A,T).B, and
axiom 4 covers L = CT.B with B in Z1.
These three cases cover all valid lists.
End of proof of theorem.

II) The second example, the traversable stack [Majster 77], is represented by a complex data type that has two stacks as its components. The syntax of the operations and an informal description of the semantics is given next. Here again the data structure works with an implicitly defined point of interest.

The domains involved are the carrier 'stack', the set 'item', and the boolean set {true,false}.

TABLE II.1 Syntax and description of traversable stacks

1) CREATE -> stack
creates an empty traversable stack.
The following two operations return 'error' unless the point of interest is at the top of the stack. In this case

2) PUSH(stack,item) -> stack
pushes an item onto the stack and

3) POP(stack) -> stack
pops the top element off the stack.

4) READ(stack) -> item
reads the item at the point of interest.

5) DOWN(stack) -> stack
moves the point of interest one position toward the bottom of the stack.

6) RETURN(stack) -> stack
resets the point of interest to the top of the stack.

7) ISMT(stack) -> {true,false}
is true iff the stack is empty.

Intuitively, the first component of the complex stack represents the actual stack, whereas the second component represents the bottom part of the stack up to the place of interest.

For the specification of the axioms, only the components of CREATE and PUSH need be named;

TABLE II.2 The components of CREATE and PUSH

E1. CREATE = CT1.CT2,
E2. PUSH = PS1.PS2;

TABLE II.3 Axioms: For all S = X.Y in 'stack' and I in 'item'

```
S1.   CT1 = CT2 = CT;

S2.   PS1(X.Y, I)  =  PS1(X.X,I);  //denoted  PS(X,I)//
S3.   PS2(X.Y, I)  =  PS(Y,I);

S4.   ISMT(CREATE)        =  true;
S5.   ISMT(PS(X,I).Y)     =  false;

S6.   DOWN(X.PS(Y,I))     =  X.Y;
S7.   DOWN(X.CT)          =  error;

S8.   READ(X.PS(Y,I))     =  I;
S9.   READ(X.CT)          =  error;

S10.  POP(PUSH(T,I))      =  T;
S11.  POP(CREATE)         =  error;
S12.  POP(DOWN(S))        =  error;

S13.  PUSH(DOWN(S))       =  error;

S14.  RETURN(DOWN(S))     =  RETURN(S);
S15.  RETURN(CREATE)      =  CREATE;
S16.  RETURN(PUSH(S,I))   =  PUSH(S,I);
```

S17. As a general rule, a function returns 'error' if any of its arguments equals 'error'.

The proof that these axioms completely describe traversable stacks is similar to the previous proof concerned with lists.

First, a lemma is proved that defines the form of all valid traversable stacks.

Lemmma:

Every valid traversable stack is of the form B.A where A and B are elements of the set Z2 and B >= A;

with Z2:
1. CT is in Z2;
2. if H is in Z2 then so is PS(H,I).

and >= :
1. A >= A;
2. PS(A,I) >= B if A >= B.

SS0, the set of all stacks, also contains the error element.

The proof that the axiom set is complete has to demonstrate that READ(S) and ISMT(S) is defined for all S in SS0.

Composition of type specifications

Complex data types, as defined in this paper, can be specified in a rather systematic way. This will be demonstrated by reconsidering the first example, the specification of lists.

First, the simple data type 'queue' will be axiomatically defined. The following operations constitute this type:

TABLE III.1 Syntax and informal description of queues

```
F1.   CT              ->  queue    //    create empty queue //;
F2.   SMT(queue)      ->  {true,false}  // is queue empty? //;
F3.   Q(queue,item)   ->  queue    // queue item onto queue //;
F4.   FRT(queue)      ->  item     //        front of queue //;
F5.   DF(queue)       ->  queue    // delete front of queue //;
```

TABLE III.2 Axioms: for all A in queue and I in item

```
Q1.   SMT(CT)      =   true;
Q2.   SMT(Q(A,I))  =   false;

Q3.   FRT(Q(A,I))  = IF SMT(A)
                       THEN  I
                       ELSE  FRT(A);
Q4.   FRT(CT)      =   error;

Q5.   DF(Q(A,I))   = IF SMT(A)
                       THEN  CT
                       ELSE  Q(DF(A),I);
Q6.   DF(CT)       =   error;
```

Now the list can be defined by the following set of statements.

TABLE III.3 Definition of lists in terms of queues

```
P1.    CREATE          =  CT.CT;
P2.    ISMT(X.Y)       =  SMT(X) & SMT(Y);
P3.    INSERT(X.Y, I)  =  Q(X,I).Y;
P4.    TAIL(X.Y)       =  CT.Y;
P5.    READ(X.Y)       =  FRT(Y);
P6.    DLT(X.Y)        =  X.DF(Y);
P7.    NEXT(X.Y)       =  Q(X,FRT(Y)).DF(Y);
P8.    RESET(X.Y)      =  IF SMT(Y)
                            THEN  Y.X
                            ELSE  RESET(NEXT(X.Y));
```

Compare TABLE III.3 with the second informal specification of lists (TABLE I.2) and notice the straight forward way in which these statements express the two-queue model of lists in terms of the queue operations.

With these statements, the axioms for lists (L1 - L15) become theorems in the theory of queues. Except for the RESET axiom, the problem of completeness has disappeared provided that the axiom set for queues is complete.

The axioms for NEXT and for READ will serve as examples to illustrate this point.

The axiom for NEXT (L13) can be derived from the statements P3,P5 - P7 by the following sequence of substitutions.

P7. NEXT(X.Y) = Q(X,FRT(Y)).DF(Y);

a) P5 //for all X FRT(Y) = READ(X.Y)//
permits the transformation

NEXT(X.Y) = Q(X,READ(X.Y)).DF(Y);

Note that equating the first parameter of READ with X, i.e. the first parameter of NEXT and of Q, was done by choice not by necessity.

b) P6 //X.DF(Y) = DLT(X.Y)// gives

NEXT(X.Y) = DLT(Q(X,READ(X.Y)).Y);

c) P3 //for all Y Q(X,I).Y = INSERT(X.Y, I)// gives

NEXT(X.Y) = DLT(INSERT(X.Y, READ(X.Y)));

d) and finally, the replacement L = X.Y leads to

L13. NEXT(L) = DLT(INSERT(L,READ(L)));

The second example is the derivation of the READ axiom (L8) from the statements P2 - P5 and from the queue axiom Q3. This example shows how the recursive structure of L8 derives from the axiom Q3.

```
P5.   READ(X.Y) = FRT(Y);
```

Since there are no substitutions that could replace FRT by a list operation, the axiom Q3 for FRT must be used.

```
Q3.   FRT(Q(A,I))   =   IF SMT(A)
                        THEN  I
                        ELSE  FRT(A);
```

a) with the substitution Y = Q(A,I), Q3 and P5 give

```
READ(X.Q(A,I))   =   IF SMT(A)
                     THEN  I
                     ELSE  FRT(A);
```

b) P4 and P2 lead to

```
READ(X.Q(A,I))   =   IF ISMT(TAIL(X.A))
                     THEN  I
                     ELSE  FRT(A);
```

c) Because of P5, FRT(A) may be replaced by READ(X.A), thus

```
READ(X.Q(A,I))   =   IF ISMT(TAIL(X.A))
                     THEN  I
                     ELSE  READ(X.A);
```

d) P3, D2, and the notational convention stated in L5 permit the substitution Q(X,I) = IN(X,I), which gives

```
L8.   READ(X.IN(A,I))   =   IF ISMT(TAIL(X.A))
                             THEN  I
                             ELSE  READ(X.A).
```

Complex types and restricted types

Many problems that can arise with axiomatic definitions may be overcome by the use of so called hidden functions. This method works, in principle, as follows. The set of functions that belong to the data type defined is enriched by additional functions for the sole purpose of facilitating the definition. The additional functions are chosen such that a finite set of axioms can be found. These new functions, however, are not made accessable to the user.

[Linden 78] interprets this method as definition by restriction. The richer set of functions defines a data type A different from the intended data type B. B is then defined by restricting A to the smaller set of operations. Theorems about objects in A are also true about objects in B provided that these theorems involve only the subset of functions that belong to B.

Hidden functions provide, indeed, a very powerful mechanism for extending the properties of types. The complex types described here can, infact, be realized by hidden (packing and unpacking) functions. On the other hand, it does not seem impossible that everything that can be done with simple types and hidden functions could also be accomplished with complex types. Investigating the relationship between these two concepts seems to be an interesting task for future research.

References

Birkhoff and Lipson 70:
Garrett Birkhoff and John D. Lipson, 'Heterogeneous Algebras', Journal of Combinatorial Theory 8 (1970) pp. 115-133.

Guttag 75:
John V. Guttag, 'The specification and Application to programming of abstract data types', Technical report CSRG 59 (September 75) University of Toronto.

Linden 78:
Theodore A. Linden, 'Specifying abstract data types by restriction', private communication.

Majster 77:
Mila E. Majster, 'Limits of the "Algebraic" specification of abstract data types', SIGPLAN Notices 12,10 (Oct. 1977) pp. 37-42.