On the verification of abstract data types

By L. VARGA

This paper describes a method for verifying the correctness of an abstract data type specification according to a concept about the abstract data type. The abstract data types are formally defined in terms of the algebraic specification technique. General rules are given for constructing theorems about a given abstract data type and for proving the theorems. These theorems serve to convince us of that the given specification correctly takes the meaning of our concept. The method is illustrated by an example.

1. Introduction

One of the most important achievements of programming methodology is the data abstraction. During the recent years, a number of different specification techniques for abstract data types has been proposed. Among these is the algebraic specification method which is described in detailed in Guttag's and Horning's common paper [1] and in an excellent tutorial paper [2].

It is known that a specification must be able to provide separate pictures to its user and its implementor. This two requirements can be satisfied by double specification [4]. A double specification has an abstract and a concrete specification part. An abstract specification is to serve the user's view and a concrete specification is to serve the implementor's view.

In the case of a double specification a verification of the correctness of an implementation according to the concept which is in our mind about the given data type can be carried out in two steps: First we show that an abstract specification correctly reflects the concept and next we verify the correctness of a concrete specification according to the abstract one.

In this paper we will be concerned only with the first phase of this verifictaion procedure when an abstract specification is given in terms the algebraic specification technique.

In section 2 a practical example of algebraic specification is given. In section 3 we provide a method for constructing theorems about the abstract data type and general rules for proving the theorems. The verification is illustrated by an example.

2. Algebraic specification, an example

Data abstraction includes a set of objects and a group of functions that operate upon this object set. An access to an object is only possible through one of the functions. A given set of objects with the functions or operations forms an abstract data type.

A composite abstract data type consists of elementary objects too. We must distinguish between an object to be defined and an elementary object, whose properties are assumed to be known. The functions of an abstract data type may include parameters. Hence a simple model of abstract data type can be characterized by three set (a set of objects, a set of elementary objects, a set of parameters) and a group of functions. Let the names of the above sets be *object, elem* and *parameter* respectively. The domain of a function is generally a cartesian product of the above sets, and its range may be a set of objects or the set of elementary objects.

The group of functions can be divided into two blocks. The first block consists of *constructor functions,* that can be used to build every values of the object set. Hence the range of a constructor function should be the set of object. The second block consists of non constructor function, called *selector functions* because these functions can be used to select parts of an object, for example an elementary object from the object structure or the remains. The range of a selector function generally is the set of elementary objects or the object set itself.

We must distinguish a *generator object* from the remains. The generator object is distinguished by the property that each value of an object set can be generated from it by applying constructor functions one after the others.

A specification method must be suitable for specifying both the syntax and the semantics of the operations.

Now let us see a solution of the specification problem given by the algebraic specification method and choose a simple case of the general abstract data model as an example for illustrating both the specification and verification method.

Let the name of the abstract type be "object"

Syntax

null: → object
assign: object×parameter×elem→object
delete: object×parameter→object
read: object×parameter→elem

Constructors: null, assign

Semantics

s: object; *p:* parameter; *e:* elem;
read (assign (*s*, *p*, *e*), *p'*)=
if $p=p'$ *then e else* read (*s*, *p'*)
delete (assign (*s*, *p*, *e*), *p'*)=
if $p=p'$ *then* delete (*s*, *p*)
else assign (delete (*s*, *p'*), *p*, *e*)
read (null, *p*)=readerror
delete (null, *p*)=null

Auxiliary function

length: object → integer
length (null) = 0
length (assign (s, p, e)) =
length (delete $(s, p)+1$)

Abstract invariant

$I_a(s)$: $0 \leqq \text{length}(s) \leqq n$

Equality

$s_1 = s_2 \equiv (s_1 = \text{null} \wedge s_2 = \text{null}) \vee$
$(\forall p)$ (read $(s_1, p) = \text{read}(s_2, p) \wedge \text{delete}(s_1, p) = \text{delete}(s_2, p)$).

The auxiliary function is distinguished from the other functions by the property that it is not used by the programs using the abstract data type. It is only a specification tool.

The abstract invariant is used to define a bounded object set:

$$\{s | I_a(s)\}$$

The equality axiom reduces the equality of two objects to the equality of their appropriate parts.

3. A proof method

Given an algebraic data type specification and a concept about the same data type, we have to show that the given specification correctly takes the meaning of verifying the correctness of a specification according to a concept, but we can convince ourself of the correctness by proving theorems about an abstract data type given by an algebraic specification. General rules for deriving such theorems are the followings:

1. The semantics of an abstract type is defined by the effects of each selector operation on an abstract object when this object is produced by a constructor operation. Other relations between two operations can be formulated as theorems and can be proved.

For example, in the case of our abstract data type all the theorems generated in this way are:

Theorem a,
assign (assign $(s, p_1, e_1), p_2, e_2)$ = assign (assign $(s, p_2, e_2), p_1, e_1)$, if $p_1 \neq p_2$

Theorem b,
delete (delete $(s, p_1), p_2)$ = delete (delete $(s, p_2), p_1)$,

Theorem c,
read (delete $(s, p_1), p_2)$ = read (s, p_2) if $p_1 \neq p_2$

2. Selector functions map an abstract object to its components. Therefore we have to show that an abstract object can be reconstructed from its selected components by constructor operations. In the case of our example we can generate only one theorem in this way:

Theorem d,

assign (delete (s, p), p, read (s, p))$=s$

We have two general rules for proving these theorems. One of them is the induction and another is that of applying the equality formula for both sides of a theorem.

The induction steps are the following:

First we show that the theorem holds for the generator object. Then supposing that the theorem holds for each element of a subset of the object set we have to prove that the theorem holds for each object generated by a constructor operation from the given subset.

We now use these rules for verifying the above theorems.

Proof of Theorem c,

We prove the Theorem by induction on s.

Basis. If $s=$null, then

delete (null, p_1)$=$null

and the result is immediate.

Induction step. Now choose

$s'=$assign (s, q, e)

and suppose that our theorem is true for s. We have to prove the theorem for s' too. It follows from semantics axioms that

read (delete $(s', p_1), p_2)=$

$=$read (delete $(s, q), p_2)$, if $q=p_1 \wedge p_2 \neq p_1$

$=e$, if $q \neq p_1 \wedge q=p_2$

$=$read (delete $(s, p_1), p_2)$, if $q \neq p_1 \wedge q \neq p_2$

We have

read (delete $(s, q), p_2)=$read (s, p_2), if $q \neq p_2$

read (delete $(s, p_1), p_2)=$read (s, p_2) if $p_1 \neq p_2$

by our induction hypothesis. The equation

read (assign $(s, q, e), p_2)=e$, if $p_2=q$

follows from the semantics of the read operation.

Proof of Theorem a,

The equality definition can be simplified by using Theorem c,. Hence the new equality formula is

$s_1=s_2=(s_1=\text{null} \wedge s_2=\text{null}) \vee (\forall p)$ (read $(s_1, p)=$read (s_2, p)).

Using this definition of equality we have to show that the read operation for both sides of Theorem a, gives the same result what ever is p. The problem can be broken up into three cases corresponding to the relations among the three parameters

1. $p=p_1$

For the left hand side:

read (assign (assign $(s, p_1, e_1), p, e_2), p)=e_2$

and for the right side we have

read (assign (assign $(s, p, e_2), p_1, e_1), p)=$

read (assign $(s, p, e_2), p)=e_2$

by using the axiom for the semantics of a read operation.

2. $p \neq p_1$ $p \neq p_2$

For the left side:
read (assign (assign (s, p_1, e_1), p_2, e_2), p)=
read (assign (s, p_1, e_1), p)=read (s, p)
and for the right side:
read (assign (assign (s, p_2, e_2), p_1, e_1), p)=
read (assign (s, p_2, e_2), p)=read (s, p)

Proof of Theorem b,
Induction on s:

Basis. s=null. Then the theorem is trivially true:

Induction step. Suppose s=assign (s_0, p, e),
where the theorem is true for s_0. Then

1. $p=p_1$
   delete (delete (assign (s_0, p, e), p_1), p_2)=
   delete (delete (s_0, p_1), p_2)

and

   delete (delete (assign (s_0, p, e), p_2), p_1)=
   delete (assign (delete (s_0, p_2), p, e), p_1)=
   delete (delete (s_0, p_2), p_1)=
   delete (delete (s_0, p_1), p_2).

2. $p=p_2$. Symmetrical case.
3. $p\neq p_1 \wedge p\neq p_2$.
   delete (delete (assign (s_0, p, e), p_1), p_2)=
   delete (assign (delete (s_0, p_1), p, e), p_2)=
   assign (delete (delete (s_0, p_1), p_2) p, e)

and similarly for the other side we have

   delete (delete (assign (s_0, p, e), p_1), p_2)=
   assign (delete (delete (s_0, p_2), p_1), p, e)=
   assign (delete (delete (s_0, p_1), p_2), p, e).

Proof of Theorem d,
We can prove the theorem by the axiom of equality. For the right hand side:

1. $q=p$
   read (assign (delete (s, p), p, read (s, p)), q)=
   read (s, q)
2. $q\neq p$
   read (assign (delete (s, p), p, read (s, p)), q)=
   read (delete (s, p), q)=read (s, q)

4. Closing comments

The procedural nature of a data type, which is important for the implementor, can be described by the Hoare-like specification [3]. In this case we have an abstract specification in algebraic form and a concrete specification in Hoare-like form. We have no direct method to verify the correctness of a Hoare-like specifi-

cation according to an algebraic specification. However it is easy to transform an algebraic specification into Hoare-like form and then we can use the verification method given by Hoare [3].

EÖTVÖS LORÁND UNIVERSITY,
MÚZEUM KRT. 6—8.
BUDAPEST, HUNGARY
H—1088

References

[1] GUTTAG, J. V. and J, HORNING, The algebraic specification of abstract data types, *Acta Inform.*, v. 10, 1978, pp. 27—52.
[2] GUTTAG, J. V., Notes on type abstraction, *IEEE Trans. Software Engrg.*, SE—6, 1980, pp. 13—23.
[3] HOARE, C. A. R., Proofs of correctness of data representations, *Acta Inform.*, v. 1, 1972, pp. 271—281.
[4] WULF, W. A., R. L. LONDON and M. SHAW, An introduction to the construction and verification of Alphard programs, *IEEE Trans. Software Engrg.*, SE—2, 1976, pp. 253—265.

(Received March 2, 1982)