A UNIFIED APPROACH TO THE SPECIFICATION AND VERIFICATION OF ABSTRACT DATA TYPES

Lawrence Flon

Computer Science Department
University of Southern California
Los Angeles, California 90007

Jaydev Misra†

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Abstract

We present what we feel to be a unification of two major although previously unrelated specification techniques for abstract data types – the algebraic and abstract model approaches. Included is a disciplined specification methodology and its resultant proof obligation for an implementation. Several important issues are discussed, including equality, bounded objects and procedural implementation. We conclude with some philosophy on design methodology and specification methodology.

1. Introduction

The use of data abstraction in programming is well recognized as a valuable methodological tool. The class mechanism of Simula 67 [1] prompted the realization on the part of software designers that the procedure, an abstraction of control structure, is by itself insufficient as a modularization mechanism. The incorporation, during design, of abstractions for data structures leads to intuitive and elegant modular decompositions. Many new programming languages have defined features for implementing data abstractions [9], and the requirements for a common higher order language for the Department of Defense give major emphasis to the idea [10].

The problem of formally specifying and verifying data abstractions has been the subject of much research. In [6], Hoare introduced the idea that specifications for data abstractions should reflect only those properties relevant to programs using the abstraction. It is the responsibility of the implementor to guarantee that his chosen representation and algorithms correctly model the abstract specifications. The form in which abstract specifications should be expressed has been the subject of some debate. Hoare originally proposed the use of input/output assertions that define the effect of an operation in terms of its changes to some abstract representation of the object in question. For example, a stack may be thought of as a sequence of values with access restricted to one end. The push and pop operations on stacks are then expressed in terms of their effect on the sequence. The fact that the implementor may choose linked lists or arrays as the concrete representation simply means that he has taken on the responsibility of demonstrating how these structures can be mapped onto a sequence. Later changes to the concrete representation, as long as the new mapping is consistent, will not affect the programs that use stacks, which only depend upon the properties of mathematical sequences for their correct operation.

An entirely different approach to specifying data abstractions is to exhibit algebraic axioms that describe the effect of an operation in terms of changes to the future behavior of other operations. This approach is called representation independent since it does not rely upon an explicit abstract representation.

It is generally agreed that the algebraic approach yields specifications more appropriate to verifying the correct operation of user programs than those yielded by Hoare's approach, which has been called the abstract model approach. On the other hand, it is also generally agreed that the process of verifying an implementation against operational specifications is conceptually easier than that of verifying it against algebraic specifications. What is desirable then is a methodology for easily verifying an implementation against algebraic-like specifications.

There are two basic problems in verifying a program against algebraic specifications. One is the fact that the algebraic model deals with values rather than objects, and values are immutable. That is, an operation can only transform one value into another – it cannot change the state of a particular value. Thus operations cannot have side effects although in a typical procedural implementation the push and pop operations will not result in new stacks but rather will alter an already exixting one.

Another serious problem is what we call the mutual reference problem. The fact that each operation name may appear in several axioms means that, in the verification methodology described by Guttag [3], each procedure body must be analyzed several times, each time for a different characteristic. In addition to the basic overhead due to multiple analyses, this means that an operation cannot be verified on its own, hence should one of the procedures be changed in the future, all of the axioms referring to the corresponding operation will have to be re-verified. We note that a recent paper by Guttag and Staunstrup [5] addresses this problem.

†Supported in part by National Science Foundation grant MCS77-09812.

CH1401-9/79/0000-0162$00.75

The purpose of this paper is two-fold. In section 2 we provide a disciplined methodology for constructing algebraic specifications. We discuss equality for abstract objects and address the problem of describing objects of bounded size. We introduce a form of specification that allows side effects on abstract objects while retaining the algebraic flavor. We also discuss the need for proving general properties of the specifications themselves.

In section 3 we describe an approach to the verification of implementations for specifications constructed along the lines of section 2. The approach has the advantage of solving the mutual reference problem, thus allowing the independent verification of each operation. Section 4 contains our conclusions and a discussion of some potential methodological implications of our approach.

2. Constructive Specification

Many of the ideas in our specification methodology are based upon the work of Guttag and Horning [4]. We place special emphasis on the notion of constructor functions – that subset of operations that can be used to build all instances of a data type. We suggest a way of defining equality among abstract objects, and we address the problem of defining objects that are bounded in size. We also introduce a specification technique intended to solve the mutability problem. Finally, we suggest that induction and substitution, along with the definition of equality, are the basic inference rules by which theorems may be deduced from the algebraic axioms.

2.1 Formulating Specifications

The constructor functions for a data type are any subset of operations sufficient for building all values of that type. For instance, constructor functions for a deque (double-ended queue) would be init, which creates an instance of an empty deque, addh, which adds an element to the head of a deque, and addt, which adds an element to the tail. Note that all instances of a deque can be constructed out of only init and addh, hence addt need not be taken as a constructor. However, we shall not address the problems of minimality or uniqueness of the set of constructor functions.

Having decided upon the constructors, we can thus define a deque to be:

1. init.
2. addh(q, e), where q is a deque and e is some element.
3. addt(q, e), where q is a deque and e is some element.

Furthermore every deque can be so constructed. Hence the value of a deque can be represented by an applicative sequence of the above functions.

It is important to note that addh and addt need not be further specified. Meanings of other operations will be specified in terms of the constructors. Let remh, remt, headh, and headt respectively denote the operations of removing the head of a deque, removing the tail, returning the value of the head, and returning the value of the tail. Also let empty indicate whether a deque is simply init or not. For each nonconstructor function , we specify its effect on a deque q when q is init, when q is addh($\hat{q}$, e) for some $\hat{q}$, e, and when q is addt($\hat{q}$, e) for some $\hat{q}$, e. This leads to the following set of axioms:

Operations:

init:	→ deque
addh:	deque × elem → deque
addt:	deque × elem → deque
remh:	deque → deque
remt:	deque → deque
headh:	deque → elem
headt:	deque → elem
empty:	deque → boolean

Constructor Functions: init, addh, addt

Axioms:

1. remh(init) = error
2. remh(addh(q, e)) = q
3. remh(addt(q, e)) = if empty(q) then init else addt(remh(q), e)
4. remt(init) = error
5. remt(addh(q, e)) = if empty(q) then init else addh(remt(q), e)
6. remt(addt(q, e)) = q
7. headh(init) = error
8. headh(addh(q, e)) = e
9. headh(addt(q, e)) = if empty(q) then e else headh(q)
10. headt(init) = error
11. headt(addh(q, e)) = if empty(q) then e else headt(q)
12. headt(addt(q, e)) = e
13. empty(init) = true
14. empty(addh(q, e)) = false
15. empty(addt(q, e)) = false

2.2 Equality

The above technique yields an "intuitively complete" specification. We have found that deciding which functions are constructors is a simple matter in practice. Note that two deques may be constructed by different sequences of constructor functions. In particular,

$$\text{addh}(\text{init}, e) = \text{addt}(\text{init}, e)$$

and this cannot be proved from the axioms as stated. Hence an equality operator for deques needs to be defined. Clearly two deques q_1, q_2 are equal if every sequence of functions yields the same result when applied to q_1 as q_2. Such a definition of equality should be defined inductively using only nonconstructor functions. For deques q_1, q_2, equality can be defined as follows:

$$q_1 = q_2 \equiv (\text{empty}(q_1) \wedge \text{empty}(q_2)) \vee (\text{headh}(q_1) = \text{headh}(q_2) \wedge \text{remh}(q_1) = \text{remh}(q_2))$$

We conjecture that such a definition need include only nonconstructor functions, since the equality of deques should be independent of any data not currently contained in them.

2.3 Bounded Objects

We next consider the problem of defining data

objects that are bounded in size. It is often more difficult to define bounded objects than their unbounded counterparts. Consider, for instance, the definition of a bounded deque, whose length is allowed to be at most n. Addition of items to a full deque should lead to an error condition.

We handle this problem by introducing the notion of attributes of an abstract object. Operations on objects are permitted only when attribute values for the object satisfy certain constraints. For a deque q, we introduce the attribute function length(q), which returns the current size of q.

It is an important observation that one needs to specify the changes in attribute values only for constructor functions. Due to the form of algebraic specifications it is possible to deduce the effect on attribute values of nonconstructor functions.

Operation	Effect on Attributes
init	length is set to 0
addh(q, e)	Permitted only if length(q)<n length(addh(q, e)) = length(q)+1
addt(q, e)	Permitted only if length (q) < n length(addt(q, e)) = length(q)+1

2.4 Procedural Semantics

A basic departure of algebraic specifications from typical implementation languages is not reflecting the procedural rather than applicative nature of the language semantics. We propose a simple solution to this problem. Note first that we can recast an axiom of the form

$$f(g(x)) = h(x)$$

in terms of pre- and post-conditions thusly:

$$y = g(x)\ \{z := f(y)\}\ z = h(x)$$

The latter axiom captures precisely the same intent as the algebraic one, but is more suited to procedural implementations. Note further that if we like, we can define a slightly altered f, say f', which does an in-place update:

$$y = g(x)\ \{f'(y)\}\ y = h(x)$$

In proposing what some will call a "bastardized" form of algebriac specification we are in fact asserting that the algebraic specification technique can be viewed as a simplification of the abstract model approach which applies whenever it is truly the case that immutable objects are required. In the general case however, we can view an algebraic axiom as a statement of the input/output behavior of a function operating upon an algebraic word representation rather than the set and sequence representations more common to the abstract model approach. Thus we leave it to the specifier to choose his (specification) weapons, so to speak, but as we shall see in section 3, the verification methodology can be a uniform one.

The previously stated algebraic axioms for deques are recast as follows, the only change to the semantics being the restriction of deques of bounded size (n). Note that the error conditions need not be explicitly stated, since they correspond to states which do not satisfy the appropriate pre-condition.

Attributes:

length: deque → integer

Constructors:

true {q := init} length(q) = 0

length(q') < n {q := addh(q',e)} length(q) = length(q')+1

length(q') < n {q := addt(q',e)} length(q) = length(q')+1

Nonconstructors:

$q' = addh(\hat{q}, e)\ \{q := remh(q')\}\ q = \hat{q}$

$q' = addt(\hat{q}, e)\ \{q := remh(q')\}$ if empty($\hat{q}$) then empty(q) else q = addt(remh($\hat{q}$), e)

$q' = addh(\hat{q}, e)\ \{q := remt(q')\}$ if empty($\hat{q}$) then empty(q) else q = addh(remt($\hat{q}$), e)

$q' = addt(\hat{q}, e)\ \{q := remt(q')\}\ q = \hat{q}$

$q' = addh(\hat{q}, e)\ \{y := headh(q')\}\ y = e$

$q' = addt(\hat{q}, e)\ \{y := head(q')\}$ if empty($\hat{q}$) then y = e else y = headh($\hat{q}$)

$q' = addh(q, e)\ \{y := headt(q')\}$ if empty($\hat{q}$) then y = e else y = headt($\hat{q}$)

$q' = addt(\hat{q}, e)\ \{y := headt(q')\}\ y = e$

q' = init {y := empty(q')} y = true

$q' = addh(\hat{q}, e)\ \{y := empty(q')\}$ y = false

$q' = addt(\hat{q}, e)\ \{y := empty(q')\}$ y = false

The predicate if B then P else Q used above is simply a notational simplification of

$$(B \Rightarrow P) \wedge (\neg B \Rightarrow Q).$$

2.5 Abstract Invariants

A predicate I(A) on an abstract object A is an invariant if and only if every instance of the abstract object satisfies it. According to our methodology, every instance of the abstract object can be created through application of constructor functions only. Hence to show that I(A) is an abstract invariant, it is sufficient to show that I(A) holds following creation of a new object, and, if it holds prior to application of a constructor function along with the pre-condition for that function, then it holds after application. Note that this follows the notion of generator induction [7], although the proof obligation has been lessened since only constructor functions need be considered. Once I(A) has been proven to be an invariant, it can be used in program proofs as a theorem. Consider the proof of the invariant

$$I(q) \equiv 0 \leq length(q) \leq n$$

1. true {q := init} length(q) = 0
   I(q) follows trivially from the post-condition.
2. Given
   length(q') < n {q := addh(q',e)} length(q) = length(q') + 1
   Show
   $0 \leq length(q') \leq n \wedge length(q') < n\ \{q := addh(q', e)\}\ 0 \leq length(q) \leq n$
   This follows trivially also.
3. The proof for addt is similar to (2).

2.6 Specification Analysis

Although one can never be certain of the appro-

priateness of a given specification, it is often useful to prove properties of that specification before attempting an implementation [2]. In proving theorems about algebraic axioms, the simplest rule of inference is that of substitution of the right hand side for the left hand side of an axiom in a proof and vice versa. Another, more important rule is that of induction. This is particularly emphasized by the explicit existence of constructor functions. In order to prove that a predicate P(x) holds for every object x, we need to show that P(init) holds, and assuming P(x) holds, that P(f(x)) holds for every constructor function f. In fact the proof of the abstract invariant is an application of this rule. Following we illustrate the use of induction to prove a theorem about the bounded deque example.

Theorem:

$$\neg[\text{empty}(q) \vee \text{empty}(\text{remt}(q))] \Rightarrow \text{remh}(\text{remt}(q)) = \text{remt}(\text{remh}(q))$$

Proof:

Induction on q:

1. Suppose empty(q). Then the theorem is trivially true.
2. Suppose $q=\text{addh}(\hat{q},e)$. If $\hat{q}=\text{init}$ then empty(remt(q)) reduces to true.
   If $\hat{q}\neq\text{init}$ then
   $\text{remh}(\text{remt}(q))=$
   $\text{remh}(\text{remt}(\text{addh}(\hat{q},e)))=$
   $\text{remh}(\underline{\text{if}}\ \text{empty}(\hat{q})\ \text{then init else addh}(\text{remt}(\hat{q}),e))=$
   $\text{remh}(\text{addh}(\text{remt}(\hat{q}),e))=$
   $\text{remt}(\hat{q})$.
   Also $\text{remt}(\text{remh}(q))=$
   $\text{remt}(\text{remh}(\text{addh}(\hat{q},e)))=$
   $\text{remt}(\hat{q})$.
3. A similar proof applies when $q=\text{addt}(\hat{q},e)$.

3. Verification of an Implementation

Re-casting algebraic specifications in terms of pre- and post-conditions allows us to unify the verification methodology with that of the abstract model approach. The ensuing presentation follows closely the Alphard methodology [8].

The re-cast specifications are to be taken as abstract specifications. We assume we are also given concrete specifications that describe the chosen implementation in detail. Thus given

$$\text{pre}_a\ \{\text{op}_a\}\ \text{post}_a$$

and

$$\text{pre}_c\ \{\text{op}_c\}\ \text{post}_c$$

it is sufficient to show pre_c follows from pre_a and post_a follows from post_c. That is, any concrete representation of an abstract object satisfying pre_a is mapped by the concrete operation op_c to some concrete representation of some abstract object that satisfies post_a. However in order to establish the correspondence between concrete and abstract objects we must introduce a mapping function from the former to the latter. The idea is due to Hoare [6]. We use the notation of [3]. The function ϕ maps a concrete representation c to an abstract value a. Since a may have many different representations, ϕ will likely not have an inverse.

We shall start by presenting a possible implementation for the bounded deque, and then describe the necessary steps in carrying out its verification.

Representation

var m:array [0..n] of elem;
h, t: 0..n;
invariant $0 \leq h, t \leq n$

Concrete Specifications

true {q := init} $h = 0 \wedge t = 0$
$t' \ominus h' < n$ {q := addh(q',e)} $m=\langle m', h, e\rangle \wedge h=h' \ominus 1 \wedge t=t'$
$t' \ominus h' < n$ {q := addt(q',e)} $m=\langle m', t', e\rangle \wedge t=t' \oplus 1 \wedge h=h'$
$t' \ominus h' > 0$ {q := remh(q')} $h=h' \oplus 1 \wedge t=t' \wedge m=m'$
$t' \ominus h' > 0$ {q := remt(q')} $t=t' \ominus 1 \wedge h=h' \wedge m=m'$
$t' \ominus h' > 0$ {y := headh(q')} $y=m'[h']$
$t' \ominus h' > 0$ {y := headt(q')} $y=m'[t' \ominus 1]$
$t' = h'$ {y := empty(q')} y = true
$t' \neq h'$ {y := empty(q')} y = false

Note that we have not actually given algorithms for the concrete operations. We shall assume, as in the Alphard methodology, that the concrete pre- and post-conditions have been separately verified, and that the only remaining task is to verify the abstract properties. In the concrete specifications we have used the notation $\oplus$ to mean addition modulo n+1, and $\ominus$ to mean subtraction modulo n+1. The expression $\langle m, i, x\rangle$ means the array m with the ith element replaced by x. The intent of the counter h is to mark the position of the head element (if there is one). The counter t marks the position of the next free slot after the tail element. The size of the deque at any time is given by $t \ominus h$.

We next define a mapping function ϕ. In fact, we will define two mapping functions ϕ_h and ϕ_t. This reflects the fact that a concrete representation can be created by continuously applying either addh or addt. It should be the case that mapping functions use only constructors. Enough mapping functions must be defined so that each possible concrete representation is in the domain of some mapping function. Having defined several mappings it is then necessary to show that all mapping functions are consistent – i.e., all of them map a given concrete representation to the identical abstract value. For a deque, we define two mapping functions ϕ_h, ϕ_t as follows:

$\phi_h(m,h,t)=\underline{\text{if}}\ h=t\ \underline{\text{then}}\ \text{init}\ \underline{\text{else}}\ \text{addh}(\phi_h(m,h\oplus 1,t),m[h])$

$\phi_t(m,h,t)=\underline{\text{if}}\ h=t\ \underline{\text{then}}\ \text{init}\ \underline{\text{else}}\ \text{addt}(\phi_t(m,h,t\ominus 1),m[t\ominus 1])$

We next define attributes over concrete representations. If f_a is an attribute of abstract objects, then f_c is the corresponding attribute of concrete representations provided

$$f_c(c) = f_a(\phi(c))$$

where c denotes any legal concrete representation. Note that the mapping ϕ may be either mapping function since presumably $\phi_h(c) = \phi_t(c)$ for all c (see (1) below).

We now give precise rules for the verification of an implementation. Let ϕ denote some mapping function, c denote any representation and a any abstract object.

1. Show that alternate mapping functions are equivalent. This is usually proven by using the defini-

tion of equality of abstract objects.

2. Show that the attribute mapping is consistent. That is,

$$f_c(c) = f_a(\phi(c))$$

3. Prove that the abstract invariant is maintained by the implementation

$$I_c(c) \Rightarrow I_a(\phi(c))$$

I_c denotes the concrete invariant, I_a the abstract invariant.

4. Proof of initialization. Given the concrete specification

$$\text{true } \{c := \text{init}_c\}\ \text{pinit}_c(c)$$

and the abstract specification

$$\text{true } \{a := \text{init}_a\}\ \text{pinit}_a(a).$$

Show that

$$\text{pinit}_c(c) \wedge I_c(c) \wedge a = \phi(c) \Rightarrow a = \text{init}_a \wedge \text{pinit}_a(a).$$

5. Proof of nonconstructor functions. Given the concrete specification,

$$\text{pre}_c(c')\ \{c := \text{op}_c(c')\}\ \text{post}_c(c)$$

and the abstract specification

$$\text{pre}_a(a')\ \{a := \text{op}_a(a')\}\ \text{post}_a(a)$$

prove the following:

(i) $I_c(c) \wedge \text{pre}_a(\phi(c)) \Rightarrow \text{pre}_c(c)$.

That is, the abstract pre-condition must imply the concrete pre-condition whenever the concrete object is in a valid state (concrete invariant holds).

(ii) $I_c(c') \wedge \text{pre}_a(\phi(c')) \wedge I_c(c) \wedge \text{post}_c(c) \Rightarrow \text{post}_a(\phi(c))$.

Show that the concrete post-condition implies the abstract post-condition provided the abstract pre-condition holds initially.

6. Proof of constructor functions. For constructor functions the pre- and post-conditions make statements about attribute values. We need to show that the effect of a constructor function g in concrete space is to convert c′ to c such that $\phi(c) = g(\phi(c'))$. Thus we need to prove:

(i) $I_c(c) \wedge a = \phi(c) \wedge \text{pre}_a(a) \Rightarrow \text{pre}_c(c)$

(ii) $I_c(c') \wedge a = \phi(c') \wedge \text{pre}_a(a') \wedge I_c(c) \wedge \text{post}_c(c) \wedge a = \phi(c) \Rightarrow \text{post}_a(a) \wedge a = g(a')$.

We work out the details of the proof for the bounded deque whose specifications have previously been given. We will use the mapping functions ϕ_h and ϕ_t defined previously.

3.1 Equivalence of Alternate Mappings

We would like to prove that

$$\phi_h(m, h, t) = \phi_t(m, h, t).$$

Apply induction on $t \ominus h$.

(i) $t \ominus h = 0$. Then h=t and $\phi_h(m, h, t) = \text{init} = \phi_t(m, h, t)$.

(ii) $t \ominus h > 0$. We need to show that $\phi_h(m, h, t) = \phi_t(m, h, t)$ or $\text{addh}(\phi_h(m, h \oplus 1, t), m[h]) = \text{addt}(\phi_t(m, h, t \ominus 1), m[t \ominus 1])$.

By induction, $\phi_h(m, h, t \ominus 1) = \phi_t(m, h, t \ominus 1)$. Hence we need to show that

$$\text{addh}(\phi_h(m, h \oplus 1, t), m[h]) = \text{addt}(\phi_h(m, h, t \ominus 1), m[t \ominus 1])$$

Using the definition of equality, we need to show that the headh and remh of both sides are equal. We give the proof only for headh.

headh(addh(ϕ_h(m, h⊕1, t), m[h])) = m[h] from the axiom.

headh(addt(ϕ_h(m, h, t⊖1), m[t⊖1])) =
if empty(ϕ_h(m, h, t⊖1)) then m[t⊖1]
else headh(ϕ_h(m, h, t⊖1)) from the axiom
= if h = t⊖1 then m[h] else headh(ϕ_h(m, h, t⊖1))
from the def. of ϕ_h
= if h = t⊖1 then m[h] else headh(if h = t⊖1
then init else addh(ϕ_h(m, h⊕1, t⊖1), m[h])
= if h = t⊖1 then m[h]
else headh(addh(ϕ_h(m, h⊕1, t⊖1, m[h]))
= m[h].

A similar proof can be given to show that

$$\text{remh}(\text{addh}(\phi_h(m, h \oplus 1, t), m[h])) = \text{remh}(\text{addt}(\phi_h(m, h, t \ominus 1), m[t \ominus 1])).$$

3.2 Attribute Mappings are Consistent

We define an attribute size on concrete representations corresponding to the attribute length on abstract objects. Let

$$\text{size}(m, h, t) = t \ominus h.$$

Then we must xhow

$$\text{size}(m, h, t) = \text{length}(\phi(m, h, t)).$$

We prove this by induction on $t \ominus h$.

(i) $t \ominus h = 0$. Then size(m, h, t) = 0 and length(ϕ(m, h, t)) = length(init) = 0.

(ii) $t \ominus h > 0$. Then size(m, h, t) = $t \ominus h$.

$$\begin{aligned}\text{length}(\phi(m, h, t)) &= \text{length}(\text{addh}(\phi(m, h \oplus 1, t), m[h])) \\ &= 1 + \text{length}(\phi(m, h \oplus 1, t)) \\ &= 1 + \text{size}(m, h \oplus 1, t) \\ &= 1 + t \ominus (h \oplus 1) = t \ominus h.\end{aligned}$$

3.3 Proof of Abstract Invariant

I_c: $0 \leq h, t \leq n$

I_a: $0 \leq \text{length}(q) \leq n$.

We have to show

$$I_c(m, h, t) \Rightarrow I_a(\phi(m, h, t))$$

or,

$$0 \leq h, t \leq n \Rightarrow 0 \leq \text{length}(\phi(m, h, t)) \leq n$$

or,

$$0 \leq h, t \leq n \Rightarrow 0 \leq \text{size}(m, h, t) \leq n$$

or,

$$0 \leq h, t \leq n \Rightarrow 0 \leq t \ominus h \leq n.$$

This follows trivially from the definition of ⊖.

3.4 Proof of Initialization

Given the concrete specification

$$\text{true } \{q := \text{init}\}\ h = 0 \wedge t = 0$$

and the abstract specification

$$\text{true } \{q := \text{init}\}\ \text{length}(q) = 0,$$

we have to show that

$h = 0 \wedge t = 0 \wedge 0 \leq h, t \leq n \wedge q = \phi(m, h, t) \Rightarrow$

$q = init \wedge length(q) = 0$.

From

$$q = \phi(m, h, t) \wedge h = 0 \wedge t = 0$$

it follows that q=init. Further, length(q)=size(m,h,t) $= t \ominus h = 0$.

3.5 Proof of Nonconstructor Functions

Proof of remh:

Abstract specification:

$q' = addh(\hat{q}, e)$ {q := remh(q')} $q = \hat{q}$

$q' = addt(\hat{q}, e)$ {q := remh(q')} if empty($\hat{q}$) then empty(q) else q = addt(remh($\hat{q}$))

Concrete specification:

$t' \ominus h' > 0$ {q := remh(q')} $h = h' \oplus 1 \wedge t = t' \wedge m = m'$.

We need to prove the following. (a), (b) are from (5) of section 3 for the first specification above. (c) and (d) are for the second specification above.

(a) $0 \leq h, t \leq n \wedge q = \phi(m,h,t) \wedge q = addh(\hat{q},e) \Rightarrow t \ominus h > 0$

(b) $0 \leq h', t' \leq n \wedge q' = \phi(m', h', t') \wedge q' = addh(\hat{q}, e)$
$\wedge 0 \leq h, t \leq n \wedge h = h' \oplus 1 \wedge t = t' \wedge m = m' \wedge q = \phi(m,h,t) \Rightarrow$
$q = \hat{q}$

(c) $0 \leq h, t \leq n \wedge q = \phi(m,h,t) \wedge q = addt(\hat{q},e) \Rightarrow t \ominus h > 0$

(d) $0 \leq h', t' \leq n \wedge q' = \phi(m',h',t') \wedge q' = addt(\hat{q},e)$
$\wedge 0 \leq h, t \leq n \wedge h = h' \oplus 1 \wedge t = t' \wedge m = m' \wedge q = \phi(m,h,t)$
$\Rightarrow$ if empty($\hat{q}$) then empty($\hat{q}$)
else $q = addt(remh(\hat{q}),e)$.

Proofs of (a), (c) are similar; they follow directly from the abstract invariant

$$0 \leq length(q) \leq n$$

and the fact that

$$length(q') = 1 + length(\hat{q})$$

from the specification of addt. The proof of (b) is a simpler case of the proof of (d). (d) can be broken up into two cases corresponding to the alternatives in the consequent.

(d1) $0 \leq h', t' \leq n \wedge q' = \phi(m',h',t') \wedge q' = addt(\hat{q},e)$
$\wedge 0 \leq h, t \leq n \wedge h = h' \oplus 1 \wedge t = t' \wedge m = m' \wedge q = \phi(m,h,t)$
$\wedge$ empty($\hat{q}$) $\Rightarrow$ empty(q)

(d2) $0 \leq h', t' \leq n \wedge q' = \phi(m',h',t') \wedge q' = addt(\hat{q},e)$
$\wedge 0 \leq h, t \leq n \wedge h = h' \oplus 1 \wedge t = t' \wedge m = m' \wedge q = \phi(m,h,t)$
$\wedge \neg$ empty($\hat{q}$) $\Rightarrow q = addt(remh(\hat{q}),e)$.

Note that (d2) involves remh($\hat{q}$) in the consequent since the original specification of remh was recursive. This creates a problem since we are required to substitute some other definition of remh($\hat{q}$) to carry out the proof. We solve the problem by using induction on length($\hat{q}$). Thus the recursive definition remh is shown to be consistent by proving that

1. the concrete specifications implement the abstract definition when length($\hat{q}$) is 0.

2. assuming that the concrete specifications implement the abstract definition correctly for all $\hat{q}$ such that length($\hat{q}$) < k, then they do so when length($\hat{q}$) = k.

We prove (2) as applied to (d2) in detail. Let $\hat{q} = \phi(\hat{m}, \hat{h}, \hat{t})$. Then the induction hypothesis permits us to replace remh($\hat{q}$) in the consequent by $\phi(\hat{m}, \hat{h} \oplus 1, \hat{t})$. The proposition then simplifies to the following (omitting the invariants):

$\hat{q} = \phi(\hat{m},\hat{h},\hat{t}) \wedge q' = addt(\hat{q},e) = \phi(m',h',t') \wedge$
$q = \phi(m', h' \oplus 1, t') \Rightarrow q = addt(\phi(\hat{m}, \hat{h} \oplus 1, \hat{t}), e)$.

Note that $m' = \langle \hat{m}, \hat{t}, e \rangle$, $h' = \hat{h}$, $t' = \hat{t} \oplus 1$. Hence we have to prove the following proposition:

$q = \phi(\langle \hat{m}, \hat{t}, e \rangle, \hat{h} \oplus 1, \hat{t} \oplus 1) \Rightarrow q = addt(\phi(\hat{m}, \hat{h} \oplus 1, \hat{t}), e)$
or

$\phi(\langle \hat{m}, \hat{t}, e \rangle, \hat{h} \oplus 1, \hat{t} \oplus 1) = addt(\phi(\hat{m}, \hat{h} \oplus 1, \hat{t}), e)$.

This is easily proven using the mapping function ϕ_t for ϕ. Note that our replacement of remh by its concrete specification, using the inductive hypothesis, resulted in a proposition that has only constructor functions and ϕ.

The proof of remt is similar.

Proof of headh:

Abstract specification:

$q = addh(\hat{q})$ {y := headh(q)} $y = e$

$q = addt(\hat{q}, e)$ {y := headh(q)} if empty($\hat{q}$) then $y = e$ else $y = headh(\hat{q})$

Concrete specification:

$t' \ominus h' > 0$ {y := headh(q)} $\wedge\ y = m'[h']$.

It should be noted that the definition of headh is recursive, hence a problem similar to that arising in the verification of remh results. We again use induction on length($\hat{q}$). Application of the inductive hypothesis permits us to replace headh($\hat{q}$) by $m[\hat{h}]$, where $\hat{q} = \phi(\hat{m}, \hat{h}, \hat{t})$. The proof is then similar to that of remh and is omitted. The proof of headt is also similar. The proof of empty is trivial.

3.6 Proof of Constructor Functions

Proof of addt:

Abstract specification:

$length(q') < n$ {q := addt(q', e)} $length(q) = length(q') + 1$

Concrete specification:

$t' \ominus h' < n$ {q := addt(q', e)} $m = \langle m', t', e \rangle \wedge t = t' \oplus 1 \wedge h = h'$

From (6) of section 3 we need to prove

(a) $0 \leq h, t \leq n \wedge q = \phi(m,h,t) \wedge length(q) < n \Rightarrow t \ominus h < n$

(b) $0 \leq h', t' \leq n \wedge q' = \phi(m',h',t') \wedge length(q') < n$
$\wedge 0 \leq h, t \leq n \wedge m = \langle m', t, e \rangle \wedge t = t' \oplus 1 \wedge h = h'$
$\wedge q = \phi(m,h,t) \Rightarrow length(q) = length(q') + 1 \wedge q = addt(q', e)$

Proof of (a):

$t \ominus h = length(q) < n$ is clear.

Proof of (b):

First, we have

$$length(q) = t \ominus h = t' \oplus 1 \ominus h = length(q') \oplus 1$$
$$= length(q') + 1 \text{ since } length(q') < n.$$

To show $q = addt(q', e)$, we expand $q = \phi(m, h, t)$ in the antecedent:

(i) $h = t : t' \oplus 1 = h' \Rightarrow t' \ominus h' = n$ a contradiction.

(ii) $h \neq t$: Then $q = addt(\phi(m, h, t \ominus 1), m[t \ominus 1])$ and $q' = \phi(m', h', t') \wedge m = \langle m', t', e \rangle \wedge h = h' \wedge t = t' \oplus 1$.

Hence $q = addt(q', e)$.

The proof for addh is similar. However, ϕ_h should be used in the antecedent when proving (a) and (b).

4. Conclusions and Methodological Implications

Our approach to the specification and implementation of data types leads to a particular design strategy in which three separate tasks can be identified. These are:

1. abstract data type specification and analysis
2. concrete data type specification and analysis
3. concrete data type implementation and verification.

The design process for a single data abstraction starts with an initial attempt at specification. At this point the constructor, attribute, and nonconstructor functions are identified and axioms are written. Equality of abstract objects is then defined. An abstract invariant is proposed and should be verified against the axioms. Any resulting detected omissions should be corrected. Optimally, other useful theorems should be postulated and proved.

Having achieved a satisfactory specification, an attempt at implementation can now be made. A suitable concrete representation is chosen and a mapping, ϕ, or set of mappings $\phi_1, \phi_2, \ldots, \phi_k$ are defined. All mappings must produce the same abstract object for the same concrete object.

Now it is time for concrete specifications to be formulated. In this paper, we have proposed that these specifications are constructed "creatively," in that an understanding of the abstract specifications is an *à priori* requirement. Once they have been formulated, they must be shown to correctly model the axioms via the mapping function(s) ϕ.

The possibility exists that the concrete specifications are derivable from the abstract axioms and the representation information. For example, in the case of a stack, one axiom would be

$$s = push(\hat{s}, e)\ \{pop(s)\}\ s = \hat{s}.$$

Given the representation (array A, counter p), and the mapping

$\phi(A,p)$=<u>if</u> p=0 <u>then</u> init <u>else</u> push(ϕ(A,p-1),A[p])

we can derive

(1) $s=\phi(A,p) \wedge \phi(A,p)=push(\hat{s},e)\ \{pop(s)\}\ \phi(A,p)=\hat{s}$

which leads to

(2) $s=(A,p) \wedge p \neq 0 \wedge \hat{s}=\phi(A,p-1) \wedge e=A[p]$
$\{pop(s)\}\ A=A' \wedge p=p'-1.$

Although this approach seems promising, it only succeeds in this case because of the fact that ϕ has a functional inverse – i.e., every abstract stack has exactly one concrete representation. Were this not the case, the mapping of (1) to (2) would not be unique, and could only be accomplished by what is probably the same amount of creativity involved in simply guessing at concrete specifications. Whether or not this derivational approach can nevertheless be used practically is an open question.

We have assumed that the concrete specifications are formulated separately from the algebraic axioms, and thus must be shown to correctly model those axioms as previously described. Finally, the data type is implemented from the concrete specifications(which are now known to be "correct"), and verified against them.

The three stages of the design process appear to be suitable for dividing the work between senior analysts who can formulate the algebraic specifications, senior programmers who can formulate the concrete specifications, and junior programmers to do the coding. Although complete verification may still not be practical, this design strategy will hopefully lead to more reliable programs.

As far as the unification of algebraic and abstract model specifications goes, we have shown that a single verification methodology suffices for both. In particular, this means that an automated verification system can have a single underlying theory from which to work, although designers can choose either specification technique as called for by a given situation. A lot of effort has previously been spent on the subject of finding the best general purpose specification technique. It appears however that each proposed technique will likely be ideally suited for some applications and poorly suited for others. Thus, as long as we can unify the underlying theories of the various techniques, we should encourage the use of whatever specification tools are best for a given application.

Acknowledgements

We would like to thank C. A. R. Hoare and H. Mills for useful discussions during the formulation of our methodology. We would also like to thank M. Melliar-Smith for his comments on a previous draft of this paper. Thanks go to Kris Pendleton for a fine job of typing this manuscript several times.

References

1. Dahl, O. J. et al., Simula 67 Common Base Language, Norwegian Computing Center, Oslo (May 1968).
2. Flon, L., "An Approach to Specification Analysis." Unpublished manuscript.
3. Guttag, J. V., "Abstract Data Types and the Development of Data Structures." Comm. ACM 20, 6 (June 1977).
4. Guttag, J. V. and Horning, J., "The Algebraic Specification of Abstract Data Types." To appear in Acta Informatica.
5. Guttag, J. V. and Staunstrup, J., "Algebraic Axioms, Classes, and Program Verification." Unpublished manuscript.
6. Hoare, C. A. R., "Proof of Correctness of Data Representations." Acta Informatica 1 (1972).

7. Wegbreit, B. and Spritzen, J.M., "Proving Properties of Complex Data Structures." J.ACM 23, 2 (April 1976).

8. Wulf, W.A., London, R.L., and Shaw, M., "An Introduction to the Construction and Verification of Alphard Progtams." IEEE Trans. on Software Eng. SE-2, 4 (Dec. 1976).

9. Proceedings of a Conference on Language Design for Reliable Software. Sigplan Notices 12, 3 (March 1977).

10. Department of Defense Requirements for High Order Computer Programming Languages. Sigplan Notices 12, 12 (Dec. 1977).