Allocations of Objects Considered as Nondeterministic Expressions

Towards a More Abstract Axiomatics of Access Types

Sigurd Meldal

Technical Report No. CSL-TR-87-335

Program Analysis and Verification Group Report No. 35

September 1987

The work was supported in part by a Fulbright Fellowship, and by a grant from the Royal Norwegian Council for Scientific and Industrial Research.

Allocations of Objects Considered as Nondeterministic Expressions — Towards a More Abstract Axiomatics of Access Types

Sigurd Meldal*

Computer Systems Lab.

Stanford University

Stanford, CA 94305

Abstract

The concept of access ("reference" or "pointer") values is formalized as parametrized abstract data types, using the axiomatic method of Guttag and Horning as extended by Owe.

Two formalizations are given. The first is a formalization of the approach used in the definition of a partial correctness system for Pascal by Hoare and Wirth. Its lack of abstraction is pointed out. This is caused by the annotation language being too expressive. An approach is taken which results in a more abstract system: The expressiveness of the annotation language is reduced and the allocation operator is viewed as a nondeterministic expression. This reinterpretation of the program language results in an appropriate level of abstraction of the proof system.

An example is given, a verification of a package defining a set type.

Key words and phrases: Axiomatic semantics, access values, program verification, Ada.

* This research was supported in part by a Fulbright Fellowship, and by a grant from the Royal Norwegian Council for Scientific and Industrial Research.

Contents

1. Introduction

The problem of aliasing in programs becomes acute whenever one wishes to reason formally about properties of a program, using e.g. the axiomatic semantics of Hoare [5]. The usual procedure when giving axioms and proof rules for programming languages is to avoid this problem by assuming that no aliasing occurs (in the case of "atomic" objects) or by considering changes to a component of a structure to be changes to the whole structure (in the case of arrays, records, etc.).

The aliasing problem has served as an argument against the use of constructs for dynamically allocating and referencing objects independently of a block-structure, since these are deemed difficult to handle in a formal fashion.

In their axiomatic definition of the programming language Pascal [6], Hoare and Wirth introduced the concept of reference types as ***collections,*** where a reference value serves as an index into a set of objects of the referenced type. Changes to a referenced object, and the allocation of new objects, were considered changes to the collection in a treatment quite analogous to that used when dealing with arrays.

In the programming language Euclid [8], such collections were introduced as a distinct type, and given a formal treatment along the lines of that given in [6].

This methodology has been applied to Ada[1] as well, by McGetrick [10].

In defining the properties of access values and collections, we want to isolate these as much as possible from the properties of the accessed type, whatever that type may be in any specific instance. We also want to restrict the number of provable properties of access values and access types to those observable in terms of program execution (full abstraction). E.g., a mapping of access values onto integers implies an ordering of access values which is not observable in the program execution (there is no ordering of access values in most languages). Obtaining a full abstraction of the formal description of access types allows the implementor of the language maximal flexibility consistent with

[1]Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

the language definition. A higher level of abstraction also has the benefit of larger sets of semantics preserving transformations of program texts. This gives an optimization part of a compiler a larger repertory to choose from, in trying to increase execution speed.

We would like to keep our treatment of access values within the paradigm (and notation) of standard partial correctness. We do not want to be forced to change proof rules of programming language constructs that are unrelated to access values. This precludes for instance the use of the abstract storage structures of [7].

In this paper we axiomatize the concept of collections and access types. We shall give two examples of how this may be done. The first essentially axiomatizes the collection concept used in [6, 8, 10, 9]. This axiomatization is too specific, resulting in distinct semantics of program constructs that would be considered equivalent in most programming languages.

This problem is addressed in our second axiomatization. We reduce the expressiveness of the annotation language by reducing the number of operators on access values. A reinterpretation of the allocation operator in the programming language then gives us the desired proof strength, but with a higher level of abstraction than in the standard formalization. Essentially we shift the proof strength of our verification system from the algebraic specification of the access value type, to the axiomata dealing with assignment and allocation of access values. We give schemata on the standard form for partial correctness proof rules for manipulation of access values.

2. Access types.

Let us briefly run through the functions relevant to our topic.[2]

The following operations are available for access value manipulation: Allocation of new objects, assigning a new value to an object designated by an access value, the NULL constant, and equality of access values. Besides these, for a specific access value type we have all the operations of the accessed type. Our axiomatization isolates the treatment of access values in general from the

[2] We shall use the programming language syntax of Ada.

particular types of the objects designated by the access values. The semantics of any ***particular*** access type is derived from the general semantics of access types presented here, and the semantics of the operations of this accessed type.

We shall not consider explicit deallocation. Errors arising from exhausted storage are properly discussed in connection with a treatment of definedness in general, and shall not be discussed in any detail here.

We presuppose the existence of the accessed type A, and the type BOOLEAN.

Given such a type A, the objects to be accessed, we introduce the access value type T and the "collection" type T_C .

The types T and T_C are mutually defined as two heterogeneous algebras [4]

$$T=[\{BOOLEAN, A, T_C\}, \{NULL, NEXT, STORE, INDEX, ALLOCATED\}]$$

and

$$T_C =[\{BOOLEAN, A,T\}, \{INIT, STORE, NEXT, INDEX, ALLOCATED\}]$$

The functions are intended to have the following interpretations:

INIT: $\rightarrow T_C$	The initial state of the collection.
NULL: $\rightarrow T$	The null pointer, it is not possible to access any object through this access value.
NEXT: $T_C \rightarrow T$	The access value of the next object to be allocated.
STORE: $T_C \times T \times A \rightarrow T_C$	Changing the value of an object changes the state of the collection.
ACCESS: T_C x T $\rightarrow$ A	Accessing the value of an object through a given access value.
ALLOCATED: $T \times T_C \rightarrow$ BOOLEAN	True if the given object is allocated in the given collection.

3. Three valued logic

In our assertions and reasoning about program states we use a many-sorted first order logic with sequences and partial functions. Let A denote the language of this theory. This version of first order logic, called ***weak logic,*** extends the ordinary first order logic with the notion of *definedness*, and the ability to reason about the definedness of constructs. For each n-ary function symbol F we have an n-ary predicate symbol Δ_F which defines the domain where F is defined. Similarly, for each predicate or formula Q with n free variables we have an n-ary predicate symbol Δ_Q identifying the domain where Q is defined.

For all formulae Q and expressions E, Δ_Q and Δ_E are always defined.

Definedness of formulae is defined constructively in terms of the definedness of the components of the formulae. In most cases an undefined component implies that the composite is undefined as well — most operators are strict. The most notable exceptions are the quantifiers and nonstrict conjunction **(and then)** and disjunction (or **else).**

Let Q be $\forall x{:}T.\ Q_1$. Then Δ_Q is valid, and Q is valid iff Q_1 is valid for all values of **x** such that Q_1 is defined.

Let Q be $\exists x{:}T.\ Q_1$. Then Δ_Q is valid, and Q is valid iff there is at least one value which when assigned to **X** makes Q_1 both defined and valid.

The two operators **and then** and or **else** have the expected meaning.

A formula Q is valid in weak logic iff it is valid for all value-assignments to the free variables for which Q is defined.

Instead of the proof-rule

$$\frac{Q,\ \neg Q}{\text{FALSE}}$$

we have the proof rule

$$\frac{Q, \neg Q}{\neg \Delta_Q}$$

i.e. Q is not defined, if we can prove both Q and its negation.

See [12] for a full presentation of this "weak logic".

4. Semantics

The set {INIT, STORE} is a generator basis for T_C (i.e. all variable-free collection expressions may be rewritten as expressions using only INIT and STORE). {NULL, NEXT} is a generator basis for T. INIT is always defined, and store is defined for access values different from NULL[3]

$$\forall C: T_C, X: T, V: A.\ \Delta_{STORE}(C, X, V) \Rightarrow X \neq NULL$$

We use implication rather than equivalence above because all store operations do not necessarily terminate normally. A storage allocation error can cause the termination of the store operation to a new object at any arbitrary point. We do have that storing a value in an object that is already allocated cannot cause an error.

$$\forall C: T_C, X: T. V: A.\ ALLOCATED(X, C) \Rightarrow \Delta_{STORE}(C, X, V)$$

We need to establish the inequality of distinct access values, i.e. access values generated by different instances of the allocator.

$$\forall C: T_C.\ NULL \neq NEXT(C)$$

$$Qc: T_C.\ \neg ALLOCATED(NEXT(C), C)$$

$$\forall C: T_C, X: T, V_1,V_2: A.\ NEXT(STORE(C, X, Vl)) = NEXT(STORE(C, X, V_2)$$

[3]In the following, for a function symbol F, Δ_F is assumed to be universally true, unless stated otherwise.

The first axiom indicates that NULL is never allocated. The second axiom ensures that no two objects allocated at different times have identical access values (we shall define ALLOCATED further shortly). The third axiom indicates that the sequence of access values returned by a sequence of allocations is independent of the values of the objects allocated.

The axioms defining the index and equality functions:

$$\forall X: T.\ \neg\Delta_{ACCESS}(INIT, X)$$

$$\forall C: T_C, X, Y: T, V: A.\ ACCESS(STORE(C, X, V), Y) = \text{if } X = Y \text{ then } V \text{ else } ACCESS(C, Y)$$

$$\forall C: T_C, X, Y: T, V_1, V_2: A.\ STORE(STORE(C, X, V_1), Y, V_2) = \text{if } X = Y \text{ then } STORE(C, X, V_2) \text{ else } STORE(STORE(C, Y, V_2), X, V_1) \quad (1)$$

Trying to access a value in the initial collection is undefined. Accessing an object through an access value returns the latest value assigned that object. (1) also indicates that the order of assignment is irrelevant, as long as the assignments are to distinct objects.

The axioms for ALLOCATED are

$$\forall X: T.\ \neg ALLOCATED(X, INIT)$$

$$\forall C: T_C, X, Y: T.\ ALLOCATED(X, STORE(C, Y, V)) \Leftrightarrow X = Y \vee ALLOCATED(X, C)$$

5. Verifying programs using access types

The proof systems based on partial correctness Hoare triples are usually quite syntactic in nature, in the sense that a precondition of a statement S can be derived by textual transformations of the desired postcondition on the basis of the textual structure of S. However, as the available type constructs of a language become richer, and syntactic sugar becomes available, we get statements that can be likened to shorthand. The common example is the use of arrays, where an assignment

```
A(1) := E;
```

to an array element may be considered shorthand for an assignment to the whole array **A**

    A := STORE(A, I, E);

i.e. A changes value in the I position.

Likewise, functions with side effects have to be considered shorthand for functions with some extra parameters, and where all calls for these other functions always have the globally accessed variables in these extra parameter positions.

We have a situation where the constructs of the programming language are ***interpreted*** in a simpler language where all objects that may be changed in a given statement are made textually visible in the interpretation of that statement.

The construction of an interpretation of any given statement is a simple exercise, but vital for the proper understanding and formulation of proof rules.

Proof rules and axioms for imperatives dealing with access values are trivially derived from the following interpretations:

For each declaration of an access value type T, we implicitly declare the object

    $COLL_T$: T_C:= INIT;

the collection of access values.

In our formalization of the classical approach [6, 8, 10], any invocation of an allocator would be considered an invocation of the NEXT and STORE functions on the current state of the collection. Thus

    new A'(E);

for an access type T is interpreted as

    $COLL_T$:= STORE($COLL_T$, NEXT($COLL_T$), E);

(We shall not delve into the case where an initializing expression is not supplied and the type A does not have a default value.)

Whenever the allocator is used in an expression, and the returned value is used, it is interpreted as the value of NEXT(COLL_T).

Any access through an access value is interpreted as the corresponding indexing operation. E.g. an occurrence of

X.all

is interpreted as

ACCESS(COLL_T, X)

Any assignment through an access value is interpreted as the corresponding store invocation on the current state of the collection. E.g.

X.all := E;

is interpreted as

COLL_T := STORE(COLL_T, X, E);

Applying the above, we get that

x := new T'(E);

is equivalent to

x := NEXT(COLL_T);
COLL_T := STORE(COLL_T, X, E);

On the basis of these transformations, the appropriate partial correctness axiomata may be derived from a standard set (e.g. [2]).

For instance, we get the following axiom for object allocation in the simple case of an allocation being the right hand side of an assignment:

$$\left\{Q^{X,\quad \text{COLL}_T}_{\text{NEXT}(\text{COLL}_T),\ \text{STORE}(\text{COLL}_T,\text{NEXT}(\text{COLL}_T),E)}\right\} \textbf{X := new } \text{T'(E); \{Q\}}$$

Assignment of one access value to another works as in an ordinary value assignment (it does not affect the state of the collection). An assignment to an object designated by an access value, however, does change the collection:

$$\left\{ Q^{\text{COLL}_T}_{\text{STORE}(\text{COLL}_T,X,E)} \right\} \; \textbf{X.all} := E; \; \{Q\}$$

6. A note on garbage collection

A common misunderstanding is that the above formal system only caters to implementations that do not use garbage collection. This is not correct. The axiomatization of the abstract datatypes “access value” and “collection’‘--embodies properties that we demand of all implementations, irrespective of how access values are mapped onto hardware, and regardless of whether this mapping changes during the execution of a program (i.e. in the presence of garbage collection).

7. Increasing abstraction.

There are two problems with the above axiomatization of access values, both of which stem from the expressiveness derivable from the existence of the function NEXT.

The first problem is exemplified by

$$D = \text{STORE}(\text{NEXT}(\text{STORE}(C, \text{NEXT}(C), E_1)), E_2)$$

What is the state of the collection D? From the axioms we can derive the fact that $\neg\text{ALLOCATED}(\text{NEXT}(C), D)$, but also that D is C extended with exactly one new element. This is at best counterintuitive, given our understanding of the function NEXT.

A possibly more serious problem with the semantics we have given is its lack of abstraction. Consider the following two pieces of code [1]:

```
X:= new INTEGER'(l);          and        Y:= new INTEGER'@);          (2)
Y:= new INTEGER'@);                      X:=  new  INTEGER'(l);
```

where X and Y are of type T, where T is access INTEGER. Presumably, the semantics of these two program fragments should be equal, assuming no failed allocation. Given the semantics of access values above, however, they are not. Consider the post-condition

$$X = \mathrm{NEXT}(C_0)$$

with the precondition

$$\mathrm{COLL}_T = C_0$$

This would form a valid (and verifiable) specification of the former program fragment, but an invalid one for the latter.

The problem stems from the introduction of the NEXT function, which essentially counts the number of times new objects have been allocated, identifying uniquely an access value by its place in a sequence of allocations. Thus it is only to be expected that a change in the sequence of allocations is reflected in a change in the identities of the objects allocated and assigned to any given identifier.

No matter how we define NEXT as a function of the collection, we shall encounter this problem. The basis is the fact that we name the next element to be allocated, beyond what is necessary for value carrying purposes. The problem is that our annotation language is ***too expressive.***

A way to avoid this problem is to define "storage structures" (which would include access values and dynamic allocation of objects) in terms of graphs of accessibility from a root object, and the equivalence of all reference paths from the root object to a given node [7].

This approach seems to us to be more suitable for the definition of programming language semantics, and not appropriate for specifying or reasoning about particular programs.

A way out of this unpleasant state of affairs is to reduce the expressiveness of the annotation language. As we shall see, this will keep us within the framework of standard partial correctness with abstract datatypes.

We diagnosed the problem as stemming from the explicit, context-independent naming of the next object to be allocated. The cure is simple, we make do without the explicit NEXT function, and thus the constructive identification of the allocated objects.

We are left with the NULL, INIT, STORE, INDEX and ALLOCATED functions, with INIT and STORE as the generator basis for collections. These, along with equality on access values, turn out to be enough to express all observable characteristics of access values and collections in programs.

Without the NEXT function we no longer have a generator basis for the domain of access values. How, then, do we deal with the allocation of a new object? Since we do not have the NEXT function available any more, we cannot know exactly which object is allocated. All we know is that the allocated object is unequal to all the previously allocated objects. Any assertion about the state of affairs after an allocation cannot assume anything about the access value of the new object, beyond this fact.

Using the notation of [11] we introduce the nondeterministic expression

$$\textbf{some } X{:}\ T.\ B(X)$$

returning an arbitrary value of type T satisfying B. We shall call the predicate B the ***defining predicate*** of the nondeterministic expression.

In order to formally establish a postcondition Q of a statement with occurrences of nondeterministic expressions, we are under the obligation to establish that Q holds for whatever value the expression returns. In terms of weakest liberal preconditions this indicates that we have to universally quantify over all possible values satisfying the defining predicate (B(X) above). A possible axiom for an assignment using nondeterministic expressions is then

$$\left\{\forall X{:}\ T.\ B(X) \Rightarrow Q_X^V\right\}\ V := \text{some } X{:}\ T.\ B(X)\ \{Q\}$$

The explosion of the complexity of the precondition relative to the postcondition is somewhat dismaying, and the universal quantifier is somewhat intimidating. If the variable x has at least on free occurrence in B, both of these may be done away with using Skolem functions in the following fashion.

In order to simplify the precondition we introduce a fresh function symbol for each textual occurrence of a nondeterministic expression in a program. In particular, such a function symbol cannot occur in any postcondition to be established for the statement invoking the particular nondeterministic expression. The defining predicate of the expression is taken as the characterizing axiom of the function symbol, and the function is defined whenever there exists a value of the appropriate type satisfying this defining predicate.

This function symbol may be thought of as representing the value returned by the nondeterministic expression. Since the function symbol must be fresh with respect to the postcondition which is to be established, we have that no information beyond what is derivable from the defining predicate may be carried from one invocation to the next.

For the simple case where the nondeterministic expression occurs as the sole component of the right hand side of an assignment we get the following axiom:

$$\left\{ Q^{V}_{F(A)} \right\}\ V := \text{some } X\text{: } T.\ B(X)\ \{Q\}$$

where F is a new function symbol which does not occur in Q, and A is the list of free variables in B(X), besides X. And we get the axioms

$$\forall A\text{: } T_1.\ \Delta_F(A) \Rightarrow 3x\text{: } T.\ B(x)$$
$$VA\text{: } T_1.\ B(F(A))$$

(where T_1 is the Cartesian product of the types of the free variables in B, besides X) characterizing F. Since every textual occurrence of a nondeterministic expression introduces a distinct function symbol, we have that in order to establish the truth of

$$\text{some } x\text{: } T.\ B(X) = \text{some } X\text{: } T.\ B(X)$$

we have to prove that

$$F_1(V) = F_2(V)$$

where F_1 and F_2 are fresh function symbols, with axioms as indicated above. In order to prove this equality on the basis of these axioms, we would have to prove that

$$\forall A_1, A_2: T_1.\ B(A_1) \wedge B(A_2) \Rightarrow A_1 = A_2$$

which does not hold, in the general case.

The allocation

V **:= new** A'(E);

is understood as equivalent to

V **:= some** X: T. X-f **NULL and not** ALLOCATED($COLL_T$, X);
$COLL_T$:= STORE($COLL_T$, V, E];

The partial correctness axiom is

$$\{ P^{V,\ \ COLL_T}_{F(COLL_T),\ STORE(COLL_T,\ F(COLL_T),\ E)} \}\ V := \textbf{new}\ A'(E)\ \{P\} \tag{3}$$

where F is a fresh function symbol with the characterization

$$\forall C: T_c.\ F(C) \neq NULL \wedge \neg ALLOCATED(C, F(C))$$

and definedness given by

$$\forall C: T_c.\ \Delta_F(C) \Rightarrow \exists X: T.\ X \neq NULL \wedge \neg ALLOCATED(C, F(C))$$

Let us return to the problem that caused this revision, the interchange of two non-interacting allocations of new objects. Let the proof system defined above be named Σ_a.

Definition: ***Two statements*** S_1 ***and*** S_2 ***are*** independent with respect to a proof system Σ *if* $S_1 S_2$ ***and*** $S_2 S_1$ ***are semantically equivalent with respect to*** Σ.

E.g. in the standard partial correctness systems the two assignments X := 3; and Y := V; where X, Y and V are distinct integer variables, are obviously independent.

Definition: ***For a statement or expression S, let*** ACC(S) ***be the set of objects that may be accessed by S, and let*** CHG(S) ***be the set of objects that may be changed by S.***

We have for instance that ACC(X := **new** INTEGER'(V+3)) = {X, $COLL_T$, V} (where T is the type of X, access INTEGER) and CHG(X := **new** INTEGER'(V+3)) = {X, $COLL_T$}.

Theorem: ***Two allocation assignments*** X := **new** T'(E_1); ***and*** Y := **new** T'(E_2); ***are independent with respect to*** Σ_a ***if*** X ***and*** Y ***are distinct identifiers,*** ACC(E_1) intersect CHG(E_2) = $\varnothing$ ***and*** ACC(E_2) intersect CHG(E_1) = $\varnothing$.

In particular the two statements of (2) are independent. Part of the requirement of the theorem is that the initializing expressions do not change the collection. This may be weakened substantially, to require only that the evaluation of one expression does not access an access value changed by the other. We shall not get into that refinement here.

(Note that CHG(X := **new** T'(E_1);) intersect ACC(Y := **new** T'(E_2);) $\neq \varnothing$, they both contain COLLT. Even so they satisfy the requirements of the theorem, and are independent with respect to Σ_a.)

Proof:

Consider two allocation assignments

x := **new** T'(E_1); and **Y** := **new** T'(E_2);

The object is to prove that their order is without semantic significance. Take an arbitrary postcondition. Using axiom (3) twice, we get the specifications

$$\{P^{X,\ \ Y,\ \ COLL_T}_{F_1(COLL_T),\ F_2(C_1),\ STORE(C_1,F_2(C_1),E_2)}\}\ \mathbf{X := new}\ T'(E_1);\ \mathbf{Y} := \mathbf{new}\ T'(E_2);\ \{P\}$$

(where C_1 = STORE($COLL_T$, $F_1(COLL_T)$, El) and

$$\{P^{X,\ \ Y,\ \ COLL_T}_{F_1(C_2),\ F_2(COLL_T),\ STORE(C_2,F_1(C_2),E_1)}\}\ Y := \mathbf{new}\ T'(E_2);\ X := \mathbf{new}\ T'(E_1);\ \{P\}$$

(where C_2 = STORE($COLL_T$, $F_2(COLL_T)$, E_2) with the characterizations

$\forall C$: T_C. $F_1(C) \neq$ NULL A $\neg$ALLOCATED(C, $F_1(C)$)

$\forall C$: T_C. $F_2(C) \neq$ NULL A $\neg$ALLOCATED(C, $F_2(C)$)

We cannot prove the equivalence of

$$(P^{X,\quad Y,\quad COLL_T}_{F_1(COLL_T),\ F_2(C_1),\ STORE(C_1,F_2(C_1),E_2)})\ \text{and}\ (P^{X,\quad Y,\quad COLL_T}_{F_1(C_2),\ F_2(COLL_T),\ STORE(C_2,F_1(C_2),E_1)})$$

since that would entail establishing

$$STORE(STORE(COLL_T, F_1(COLL_T), E_1), F_2(STORE(COLL_T, F_1(COLL_T), E_1)), E_2) =$$
$$STORE(STORE(COLL_T, F_2(COLL_T), E_2), F_1(STORE(COLL_T, F_2(COLL_T), E_2)), E_1)$$

which does not hold for all possible choices of F_1 and F_2..

On the other hand., since F_1 and F_2 are very loosely characterized, we can find models where this equality does hold. Furthermore, because of the requirement that F_1 and F_2 be new identifiers we do have semantic equivalence between the two statement lists. The reason is that in order to establish the preconditions of the two lists, we have to establish that the they hold for ***an arbitrary*** choice of function values satisfying their respective characterizations. Using the noninterference between E_1 and E_2, and the commutativity of STORE operations on distinct access values, we can therefore establish that the two statement lists leave us with equivalent obligations in terms of establishing preconditions for any given postcondition. The two statement lists are therefore semantically equivalent.

..

There is still a level of abstraction that one might wish to attain. Consider the statement

declare
    x: T_1; -- ***where the type*** T_1 ***is declared as*** **access** T_2;
begin
    x := **new** T_2;
end;

In Σ_α, the statement above is provably different from a null statement. Even though neither x nor the object pointed to by X are accessible from outside the block, the effect of allocating a new object is observable as a change in the state of $COLL_T$. One may argue whether this is reasonable or not. We believe that the complexity of a system that would insist that the allocation of unobservable objects be unobservable would be unacceptable. It would have to deal with direct and indirect accessibility of objects, embedding all the information of a garbage collector system.

8. The Anna annotation syntax

In the following example we shall make use of some of the annotation conventions of Anna [9].

A type declaration may be followed by an annotation, e.g.

```
type EVEN is new INTEGER;
wherex: EVEN => X MOD 2 = 0;
```

This constrains all variables of type EVEN to always be divisible by 2, i.e. the annotation defines an ***invariant*** on the type EVEN.

A subprogram may be annotated by expressions using **in** and **out** values of parameters, e.g.

```
procedure SQUARE(X: in out INTEGER)
where out(X = in(X * X));
```

constrains the returned value of the parameter **X** to be the square of the value of **x** upon procedure invocation. The modifier **out** indicates that the following expression is to be evaluated with the parameter values upon termination of the procedure, the modifier **in** indicates that the expression is to be evaluated using the values of the parameters at the time the procedure is called.

A function may be annotated similarly:

```
function SQUARE(X: INTEGER) return INTEGER
where return X * X;
```

The annotation characterizes the value returned by an invocation of the function, and is equivalent to an axiom stating

$$\forall X: \text{INTEGER}.\ \text{SQUARE}(X) = X * X;$$

9. An example

Consider the following specification of a package defining the set type over a given type T.

```
package SMALL-SET is
        limited private type SET;
        procedure INIT(S: out SET);
        function MEMBER(S: SET; X: T) return BOOLEAN;
        procedure ADD(S: in out SET; X: T);
        procedure SUB(S: in out SET; X: T);
end SMALL_SET;
```

We implement the package using a simple linked list! (the expressions in curly brackets are assertions holding at their respective points during program execution.):

```
package body SMALL_SET is
        type SET;
        type LINK is record
                M: T; -- The element
                N: SET; -- Pointer to the next element
        end record;
        type SET is access LINK;
        where   S: SET. ¬MEMBER(S.N, S.M) ∧
                        S1, S2: SET. S1.N = S2.N ⇒ S1 = S2;
        -- Representation invariant:
        -- The sets are represented as nonrepeating lists without cycles,
        -- Distinct set objects have disjoint lists.

        procedure INIT(S: out SET)
        where out (S = NULL)
        is . . . end INIT;
```

[4]The careful reader will notice that we have departed somewhat from Ada syntax. Hopefully the intended semantics of the package will be clear.

```
function MEMBER(S: SET; X: T) return BOOLEAN
where return S ≠ NULL and then X = S.M v MEMBER(S.N, X)
is . . .
end MEMBER;

procedure ADD(S: in out SET; x: T)
where
out (if in(MEMBER(S,X)) then (S,COLL_SET) = in(S,COLL_SET) else
        ∃T: SET. ¬ALLOCATED(T, (in COLL_SET)) ∧ T ≠ NULL ∧
                    (S, COLL_SET) = in(T, STORE(COLL_SET, T, (X,S)))
is
begin {P3}
        if not MEMBER(S, X) then {P2}
                S := new SET'(M=>X; N=>S);
        end if;
end; {P1}

procedure SUB(S: in out SET; X: T)
where out (t/Y: T. MEMBER(S, Y) ⇔ in(MEMBER(S, Y) ∧ Y ≠ X))
is
        I: SET;
begin   {Q5}
        if S ≠ NULL then
        if S.M = X then
                S := S.N;
        else    {Q4}
                I := s;
                while I .N ≠ NULL and then I.N.M ≠ X loop {Q3}
                        I := I.N;
                end loop; {Q2}
                if I.N ≠ NULL then I.N := I.N.N;
                end if;
        end if; end if; {Q1}
end SUB;
```

```
        -- Equality on the set implementation:
        function "="(S, T: SET) return BOOLEAN
        where return ∀x: T. MEMBER(S,X)⇔MEMBER(T,X));
        is . . . end "=";
    end SMALL_SET;
```

It is beyond the scope of this paper to fully prove that the above package implements an abstract SET type. We shall be satisfied with verifying the annotations of the ADD and SUB procedures. In order to do so, we shall make use of the concept of ***effect functions*** [3]. An effect function is an abstraction of the effect of a given procedure or function. Its domain is the Cartesian product of the types of all the accessed variables (be they parameters or globals), and the codomain is the Cartesian product of the types of all the variables that may be changed during the subprogram execution (**out** or **in out** parameters or globals). The effect functions are characterized by axioms derived from the annotations of the respective subprograms. In reasoning about subprogram invocations we have to use effect functions rather than the plain procedure names whenever a procedure accesses global variables. This is because our proof system is syntactical in nature, and we have to have the names of all variables that affect the evaluation of an expression available in the annotations using that expression. For every occurrence of a function in an annotation, we substitute the corresponding effect function when performing a proof. The reader is referred to [3] for a full presentation of effect functions.

In our verification of the ADD and SUB procedures we shall make use of the effect function of the MEMBER function. Besides accessing the parameters, the evaluation of a call to MEMBER accesses the state of $COLL_{SET}$. Therefore, the effect function F_{MEMBER} corresponding to a call MEMBER(S,X) has three parameters – $COLL_{SET}$, S and X. F_{MEMBER} is characterized by the axiom[5]

$$\forall C: T_C, S: SET, X: T.\ F_{MEMBER}(C, S, X) = S \neq NULL$$
$$\textbf{and then } X = ACCESS(C, S).M \lor F_{MEMBER}(C, S, X)$$

Let us first verify that ADD performs as annotated. The proof obligations are

[5]How the axiom is derived from the annotation should be obvious.

$$\{F_{MEMBER}(COLL_{SET}, S, X) \wedge (COLL_{SET}, S) = (C_0, S_0)\}$$
$$ADD_{BODY}$$
$$\{(COLL_{SET}, S) = (C_0, S_0)\}$$

(i.e. that if x already is a member of S, then no changes are made), and

$$\{\neg F_{MEMBER}(COLL_{SET}, S, X) \wedge (COLL_{SET}, S) = (C_0, S_0)\}$$
$$ADD_{BODY} \tag{4}$$
$$\{\exists T: SET.\ \neg ALLOCATED(T, C_0) \wedge T \neq NULL \wedge (COLL_{SET}, S) = (STORE(C_0, T, (X,S_0)), T)\}$$

We may assume that the invariant on the SET type given in the annotation of the type declaration holds[6]. It is trivial to verify the first of these partial correctness theorems. Establishing the second is also straightforward:

Let the precondition of 4 be named P3, and the postcondition P_1 (as indicated in the program). Let P_2 be the precondition of **the then** part of the selection statement. We must have that

$$P_2 \Rightarrow P_1{}^{S,\qquad COLL_{SET}}_{F(COLL_{SET}),\ STORE(COLL_{SET}, F(COLL_{SET}), (X,S))}$$

where F is is a new function symbol, with characterization

$$\forall C: T_C.\ \neg ALLOCATED(F(C), C) \wedge F(C) \neq NULL.$$

When we expand the above, it becomes

$$P_2 \Rightarrow \exists T: SET.\ \neg ALLOCATED(T, C_0) \wedge T \neq NULL \wedge$$
$$(STORE(COLL_{SET}, F(COLL_{SET}), (X,S)), F(COLL_{SET})) = (STORE(C_0, T, (X,S_0)), T)$$

which reduces to TRUE, given the characterization of $F(COLL_{SET})$. Since P3 implies that the test of the selection statement evaluates to TRUE, we have indeed verified the annotation of ADD.

Now we shall verify the consistency of the SUB procedure with its annotation.

[6]To prove that the invariant never is violated demands a separate proof, which we shall not get into here?

The partial correctness theorems that need to be established are

$$\{F_{MEMBER}(COLL_{SET}, S, Y) \land Y \neq X\}$$
$$SUB_{BODY}$$
$$\{F_{MEMBER}(COLL_{SET}, S, Y)\}$$

and

$$\{\neg F_{MEMBER}(COLL_{SET}, S, Y) \lor Y = X\}$$
$$SUB_{BODY}$$
$$\{\neg F_{MEMBER}(COLL_{SET}, S, Y)\}$$

where SUB_{BODY} is the body of the procedure and Y is a fresh name. We shall only prove the first of these:

The postcondition to be established is

$$F_{MEMBER}(COLL_{SET}, S, Y) \qquad Q_1$$

This obliges us to prove that

$$(I = NULL \land F_{MEMBER}(COLL_{SET}, S, Y)) \lor$$
$$(I \neq NULL \land F_{MEMBER}(STORE(COLL_{SET}, I, (I.M, I.N.N)), S, Y)$$

holds after loop termination? This is implied by

$$INV(COLL_{SET}) \land$$
$$F_{MEMBER}(COLL_{SET}, S, Y) \land X \neq Y \land \qquad Q_2$$
$$(F_{MEMBER}(COLL_{SET}, S, X) \Leftrightarrow F_{MEMBER}(COLL_{SET}, I.N, X)) \land$$
$$(I.N = NULL \text{ or eke } I.N.M = X)$$

where $INV(COLL_{SET})$ is the invariant over the SET type.

[7]If we were to expand it fully, I.M would be $ACCESS(COLL_{SET}, I).M$, and likewise I.N.N would be expanded to $ACCESS(COLL_{SET}, ACCESS(COLL_{SET}, I).N).N$.

Let

$$\begin{array}{l} INV(COLL_{SET}) \wedge \\ F_{MEMBER}(COLL_{SET}, S, Y) \wedge X \neq Y \wedge \\ (F_{MEMBER}(COLL_{SET}, S, X) \Leftrightarrow F_{MEMBER}(COLL_{SET}, I.N, X)) \end{array} \qquad Q_3$$

be the loop invariant. It is no great feat to verify that this really is invariant for every iteration, and that

$$\begin{array}{l} INV(COLL_{SET}) \wedge \\ F_{MEMBER}(COLL_{SET}, S, Y) \wedge X \neq Y \wedge \\ I = S \wedge (I \neq NULL \textbf{ and then } X \neq S.M) \end{array} \qquad Q_4$$

implies the invariant. It is equally easy to establish this as the precondition of the loop, given a precondition of

$$INV(COLL_{SET}) \wedge F_{MEMBER}(COLL_{SET}, S, Y) \wedge Y \neq X \qquad Q_5$$

We shall not go through the proof that the other branches of the if-statement all establishes the desired postcondition, given the precondition.

10. Conclusion

We have given two formal semantics to access values. We pointed out a lack of abstraction inherent in the usual approach, and reduced the expressiveness of the language somewhat in order to obtain a fuller abstraction. In order to obtain the necessary strength of the accompanying proof system for partial correctness of programs using access values, we viewed allocations as nondeterministic expressions. The result is a quite abstract semantic system for access values in programming languages, within the framework of standard partial correctness reasoning.

It is interesting that the allocation, an inherently deterministic operation, is best viewed as nondeterministic for verification purposes. We have found that the use of nondeterminism is quite often appropriate in trying to obtain a suitable level of abstraction.

If one finds our syntax cumbersome, we suggest that one adopts for instance that of the Anna annotation language. It introduces a shorthand notation for the most frequently used operators, making expressions more legible at the cost of some unorthodoxy.

11. Bibliography

[11 America, P. ***(1986) Object-oriented programming: A theoretician's introduction.*** Bulletin of the EATCS. 29, June, 69-84.

[21 Apt, K. R. ***(198*** 1) ***Ten years of Hoare's logic: A survey — Part*** I. ACM TOPLAS. 3, 4, 431—483.

[31 Dahl, O.-J. ***(1977) Can program proving be made practical.*** in: Les Fondements de la Programmation. Amirchahy, M., and Neel, D., ed. CCE-CREST, IRIA. 56-115. (Also in: Research Reports in Informatics no. 33, Institute of Informatics, University of Oslo, 1978).

[41 Guttag, J. V., and Horning, J. J. (1978) ***The algebraic specification of abstract data types.*** Acta Informatica. 10, 27—52.

[51 Hoare, *C.* A. R. (1969) ***An axiomatic basis for computer programming.*** Communications of the ACM. 12, 10,

[61 Hoare, C. A. R., and Wirth, ***N.*** (1973) ***An axiomatic definition of the programming language PASCAL,.*** Acta Informatica. 2, 335—355.

[71 Jonkers, H. B. M. (1981) Abstract Storage Structures. Preprint IW 158/81, Mathematisch Centrum, Amsterdam.

[81 London, R. L., Guttag, J. V., Horning, J. J., Lampson, B. W., Mitchell, J. G., and Popek, G. J. (1978) ***Proof rules for the programming language Euclid.*** Acta Informatica. 10, 1, 1—26.

[9] Luckham, D. C., von Henke, F.W., Krieg-Bruckner, B, and Owe, 0. (1984) ANNA — A Language for Annotating Ada Programs. Stanford University, Tech. Report no. CSL-84-261.

[10] McGettrick, A. D. (1982) Program Verification Using Ada. Cambridge Computer Science Texts, Cambridge University Press, Cambridge.

[11] Meldal, S. (1986) ***Partial correctness of exits from concurrent structures.*** BIT. 26, 3, 295—302.

[12] Owe, 0. (1985) An Approach to Program Reasoning Based on a First Order Logic for Partial Functions. Institute of Informatics, University of Oslo, Res. Report no. 89.