TYPES ARE NOT SETS*

James H. Morris, Jr.
Xerox Corporation
Palo Alto Research Center (PARC)
Palo Alto, California 94304

Introduction

The title is not a statement of fact, of course, but an opinion about how language designers should think about types. There has been a natural tendency to look to mathematics for a consistent, precise notion of what types are. The point of view there is extensional: a type is a subset of the universe of values. While this approach may have served its purpose quite adequately in mathematics, defining programming language types in this way ignores some vital ideas. Some interesting developments following the extensional approach are the ALGOL-68 type system [vW], Scott's theory [S], and Reynolds' system [R]. While each of these lend valuable insight to programming languages, I feel they miss an important aspect of types.

Rather than worry about what types are I shall focus on the role of type checking. Type checking seems to serve two distinct purposes: authentication and secrecy. Both are useful when a programmer undertakes to implement a class of abstract objects to be used by many other programmers. He usually proceeds by choosing a representation for the objects in terms of other objects and then writes the required operations to manipulate them.

There are two problems about the interaction of his programs with others that he has great difficulty solving without the aid of a type system to constrain the interactions. First, users of his programs may ask them to operate a value that is not a valid representation of any of the objects he has undertaken to process. This is the authentication problem. He could begin each operation with a well-formedness check, but in many cases the cost would exceed that of the useful processing. Second, users may write programs that depend upon the particular representation he chooses for objects. This precludes his re-designing the representation. This is the secrecy problem.

Type checking is a way to prevent such things from happening. All values used to represent the abstract objects are considered to be of a certain type. The rules are:

(1) Only values of that type can be submitted for processing (authentication).

(2) Only the procedures given can be applied to objects of that type (secrecy).

The remaining question is how to decide whether a given value has a particular type. The foregoing discussion should serve to persuade the reader that the question should not turn on what the value is, but rather where it came from or who created it. In other words type should not be an extensional property.

I shall now outline the design for an extendible type system based on the foregoing considerations.

Modules

The overall structure of the language must permit different parts of the same program to have different capabilities. Any language with textual scope rules has this property to some extent; e.g. a FORTRAN subroutine has exclusive access to its local variables.

*This work was supported by NSF Grant GJ34342X while the author was at the University of California, Berkeley.

The syntactic construct module is introduced to provide such structure. Basically, a module has the form of a complete ALGOL-60 program (delimited by module, end); the main program part (if any) may be used to perform initialization. A module may appear anywhere a declaration can. Any identifiers declared at the outer-most level of the module may be tagged (with +) to signify that they are entries in which case they are known in the enclosing block. If the identifier denotes storage that storage may be read (but not written) from outside the module. If it denotes a procedure, the procedure may be called from outside. A module is very similar to a SIMULA class [D].

Example 1. The following program defines and tests a scheme to remember equivalence classes. The array A and procedure R are unknown outside the module, while count, Equiv, and Equate are known.

```
module
    integer array A [1:100];
    integer +count;
    integer procedure R(i);integer i;
      begin integer t; t:=i;
         while A[t]≠0 do t:=A[t];
         R:=t
      end;
    Boolean procedure +Equiv (x,y);
    integer x,y;
         Equiv := R(x) = R(y)
    procedure +Equate (x,y); integer x,y;
         begin integer u,v;
           u:=R(x); v:=R(y);
           if u≠v then begin count:=count-1;
                             A[u]:=v
         end
         end;
    for count:=1 step 1 until 100 do
         A[count]:=0;
    count :=100
 end
 Equate (10,20);
 Equate (20,5);
 Print (count); Print (Equiv (10,5))
```

The programmer of a module is supposed to assume nothing about the programs that access its entries. He should use private storage to maintain the integrity of his data structures and treat parameters with suspicion. (The previous example violates the latter unless array-bounds checking is assumed.)

A module's initialization part is executed whenever its enclosing block is entered. The question of whether modules can call each other recursively is ducked.

While it is conceptually simpler to regard every program as one piece of text, in practice one must break them up to facilitate editing, compiling, etc. The most natural way to do this seems to map (large) blocks into file directories and modules into files.

Types

First, a thoroughly dynamic type mechanism, similar to that in [M] will be presented. It seems more appropriate for operating systems and is similar also the one designed for Carnegie's HYDRA system [W].

Performing the operation Createseal(false) delivers to the caller two capabilities (in the form of procedures) $Seal_i$ and $Unseal_i$ along with the integer i. These things are unique in the sense that Createseal returns different versions every time it is called. Its role is analogous to a locksmith's.

$$Seal_i(X)$$

produces a value X' with the following limited properties

(i) Testseal(X',i) is true
(ii) $Unseal_i(X')=X$

In addition, unless y is a value produced by $Seal_i$, Testseal(y,i) is false and Unseal (y) causes an error. The seal is opaque in the sense that no properties of X can be discovered; i.e. a primitive applied to X' causes an error.

The operation Createseal(true) produces a transparent sealing operation $Seal_i$, along with i. The value

$$X'=Seal\ (X)$$

retains all the properties of X except that Testseal(X') is true. In addition, Testseal (y,i) is false unless y was produced by a succession of seal operations one of which was $Seal_i$. In other words, transparent seals are like trademarks in that any number may be affixed without obscuring each other or any other properties of the object.

It is suggested that a programmer implementing a new type use these operations in the following way.

(1) During module initialization he calls Createseal, with false if he wishes an opaque type, true otherwise.

(2) Every time an object of the new type is passed out of the module he guarantees that it is sealed.

(3) Every time a value allegedly of the new type is passed into the module he tests its seal and unseals it if it is opaque.

This scheme detects type errors by extra-module code since only the module is

able to put its seal on things. Further, if the seal is opaque the representation of the new type is kept a secret.

A considerable gain in clarity and efficiency can be achieved if the following assumptions are valid.

(a) A fixed number of new types are created by the program, independent of the data.

(b) Each new type is represented by a set of values describable as a compile-time type (e.g. an ALGOL-68 mode).

(c) Sealing, testing, and unsealing are performed only at module boundaries in the manner described above.

(d) Testseal is never used.

In such cases the type checking (exclusive of union testing) can be done at compile-time and the sealing-unsealing activity becomes so implicit as to disappear.

To invent a new type in the static system one makes a declaration of the form

type <name> {>|≥} <type description>

Normally the name will be tagged as an entry, indicating that the type name can be used outside the module. If > is used the type is opaque, if ≥ it is transparent. The type checking rules can be summarized by some simple coercion rules

(1) Inside the module <name> is simply an abbreviation for <type description>; i.e. coercions in either direction are allowed.

(2) Outside the module an opaque type cannot be coerced at all; i.e. something declared a <name> can reside only in contexts declared such.

(3) Outside the module a transparent type may be coerced downward; i.e. something of type <name> may move into contexts with type <type description>, but not back.

It should be emphasized that coercion here never induces any computation. The rules simply serve to control the contexts values may move between.

Example 2. The following program defines and exercises three modules. The first implements pairs of real numbers as arrays (assuming arrays are heap allocated, in the ALGOL-68 sense). The second implements intervals as real pairs; the third, complex numbers as real pairs. Each of the new types may be used in declarations outside its module. Since realpair and interval are transparent types a value of type interval may be treated like an array for purposes of reading. However, an assignment to an interval's component

```
int[1] :=5
```

is not permitted because it is regarded as the assignment

```
int := <5,int[2]>
```

and the right-hand side has implicit type real array which implies an upward coercion. Since complex is an opaque type complex numbers must be treated as atoms outside the module, using the procedures re and im to get at their components.

```
module
     type +realpair ≥ real array [1:2];
     realpair procedure +mkpair (x,y);
                                    real x,y;
       begin realpair t; t[1]:=x; t[2]:=y;
             mkpair:=t
       end
end;

module
     type +interval≥realpair;
     interval procedure mkinterval (x,y);
                                      real x,y;
       mkinterval:=if x≤y then mkpair (x,y)
                          else mkpair (y,x);
     interval procedure sumint (x,y);
                                   interval x,y;
       sumint:=mkpair (x[1]+y[1], x[2]+y[2])
end;

module
     type +complex>realpair;
     complex procedure mkcomplex(x,y);
                                     real x,y;
       mkcomplex := mkpair (sqrt(x*x+y*y),
       if x=0 then pi/2*(y/abs(y))
          else arctan (y/x));
     real procedure re(c); complex c;
       re := c[1] * cos (c[2]);
     real procedure im(c); complex c;
       im := c[1] * sin (c[2])
end

real t; real array a [1:2]; realpair x;
interval y; complex z;
y := mkinterval(5,10); x:=y; a:=y; t:=y[2];
y:=sumint (y,y); z:=mkcomplex (3,6.5);
t:=re(z); t:=im(z)
```

There are several statements that would not be legal in the main program. Among them are

```
x:=a; y:=x; y[2]:=6; sumint(y,x);
x:=z; y:=z; t:=z[1]
```

The first four call for upward coercions which are never permitted while the remaining three attempt to coerce an opaque type.

This type system has been designed to facilitate program verification on a modular basis. The general principle is that a module writer should not have to look outside his module to verify its correctness. Part of this is achieved by defining correctness judiciously and the rest by having the type checker restrict how the outside world communicates with a module.

Associated with a new type T may be a type invariant of the form

$$x \varepsilon T \Rightarrow P(x)$$

e.g.,

$$x \varepsilon \text{interval} \Rightarrow x[1] \leq x[2]$$

To prove that it holds everywhere outside the module the verifier must

(1) prove P(v) holds for every variable v of type T at the end of initialization (if any).

(2) at each entry of the module assume the invariant holds and prove P(x) for all changed variables and returned values of type T at the exits of the module.

For example, to prove the interval invariant above one must prove that the parameters of each call of mkpair in mkinterval and sumint are non-decreasing; in summint one may assume the invariant holds for the formal parameters x and y.

Opaque types do not facilitate proofs per se but their significance can be illustrated by considering proofs. The external specification of a module might be given as a set of statements of the form

$$Q(x, f_i(x))$$

where f_i is function describing the effect of the ith procedure entry. E.g.

```
mkcomplex (x,y) = <x,y>
z=<x,y> => re(z)=x
z=<x,y> => im(z)=y
```

Since values of an opaque type cannot be sampled directly there is no need for these statements to be literally true; indeed they are not for the current example. The actual situation can be described by assuming an implicit representation function R which maps the underlying representation into the externally assumed one. E.g.,

```
R(z) = <z[1]*cos(z[2]), z[1]*sin(z[2])>
```

Then the internal specifications given to the module writer are derived by inserting R at appropriate places. E.g.,

```
R(mkcomplex (x,y))=<x,y>
R(z)=<x,y> => re(z)=x
R(z)=<x,y> => im(z)=y
```

Now the module writer is free to choose any R he likes and prove that the programs obey the internal specifications. For this particular example he would be wiser to choose R to be the identity function and revise the program accordingly. The important point is that he is free to change R any time he likes and the external specifications need not change.

Remarks

The type system presented here is like SIMULA's class construction but improves on it in a few respects. First, it allows the module writer to have absolutely private data while SIMULA allows outside programs to access all a class's storage. Second, it makes implementing binary operations on new types simple, something that is nearly impossible in SIMULA. A student of Hoare's, John Elder, has designed revisions of the SIMULA class mechanism to overcome these problems.

The notion of a transparent type provides a very limited form of polymorphism. If several transparent types are coerceable down to a single base type it is possible to write a single procedure that accepts any of those types as a parameter. It is not entirely clear how to achieve true polymorphism since such a procedure is unable to return values of varying type, dependent on the input type.

An interesting point comes up when one tries to justify the proof rule for type invariants: what constitutes a module entry or exit? If one has a language like ALGOL-60 with recursion and non-local transfers, each procedure call in the module must be regarded as an exit followed by an entry. This can be quite bothersome because it requires that everything be cleaned up (i.e. the invariant's truth assured) before every procedure call. It is therefore appealing to put the following constraints on the control structure.

(1) Any recursive chain of procedures must be entirely within one module and cannot involve entries of the module.

(2) A non-local transfer cannot cross any module boundary.

To mitigate the effect of the latter restriction one can provide an error exit mechanism similar to PL/1 on-conditions. An error signal propagates back along the

call chain allowing each module to clean things up so as to restore its invariants.

References

[vW] van Wijngaarden, A., "Report on the Algorithmic Language ALGOL-68," Mathematische Centrum, 1969.

[S] Scott, D., "Outline of a Mathematical Theory of Computation," PRG-2, Oxford University.

[R] Reynolds, J.C., "A Set-Theoretic Approach to the Concept of Type," Argonne National Laboratories, 1969.

[D] Dahl, O.J., and Hoare, C.A.R., "Hierarchical Program Structure," Structural Programming, Academic Press, 1972.

[M] Morris, J.H., "Protection in Programming Languages," Comm ACM, 16 (1), Jan. 1973.

[W] Wulf, W.A., et. al., "Hydra: the Kernal of a Multiprocessor Operating System," Department of Computer Science, Carnegie Mellon University, June 1973.