SOME IDEAS ON DATA TYPES IN HIGH LEVEL LANGUAGES

David Gries and Narain Gehani

TR 75-244

May 1975

This research was supported by NSF Grant GJ-42512. We are grateful to the members of IFIP working group 2.3 on programming methodology for several discussions held at the December 1974 meeting.

SOME IDEAS ON DATA TYPES IN HIGH LEVEL LANGUAGES

David Gries and Narain Gehani
Department of Computer Science
Cornell University
Ithaca, N.Y.

Abstract:

We explore some new and old ideas concerning data types; what a data type is, overloading operators, when and how implicit conversions between programmer data types should be allowed and so forth. The current notion that a data type is a set of values together with basic operations on that set leads us to conclude that formal parameter types need not be so explicitly stated. Given a formal parameter X with operations $\sigma_1, \ldots \sigma_n$ being performed on X within a procedure, one should be able to supply , as actual parameter in call, a variable of any type which has operations $\sigma_1, \ldots, \sigma_n$ defined on it. We introduce a notation for this, using PASCAL as a base language, illustrate the added flexibility it gives us, and show briefly how to implement the idea efficiently.

I. Introduction

A current notion in programming language research is that a data type is not only a set of values, but consists also of a set of basic, primitive operations on these values. This notion will eventually find its way into general purpose languages. In this paper we explore some ideas which will have to be considered, such as overloading operators and procedures, and describing conversions between types.

One of our new ideas is that the domain of an array A, i.e. the set of legal subscript values of A, should be a data type, and that one should be able to declare variables of that type. This addition makes processing of array elements, or iterating over all array elements, simpler in that the same notation is used whether A has one, two, or more dimensions. Stated another way, nested loops are no longer necessary to iterate over the elements of a two or three dimensional array. The new notation is simple and frees the user from having to deal with unnecessary details.

A second new notion generalizes procedures and calls on them. We want to make the type of a formal parameter of a procedure depend on a particular call of that procedure. Thus, for one call P(1.0) of procedure P with formal parameter X, X has type real, while for another call P(1), X has type integer.

If used correctly, this notion of procedure parameters with "variable" types is a useful abstraction. In general, we are not interested in what type a parameter X has, but instead, we are concerned with the operations preformed on X within the procedure. Thus, when we write a procedure to sort an array A, we really are concerned only with the fact that the operations := and $\leqslant$ are defined for array elements of A and not whether we are sorting integers, reals or strings.

This abstraction from data type to the set of operations performed on the data type within a procedure is so important that a high level language should support it. The main difficulty is in embedding it in a high level language in a clean, natural, systematic manner which does not decrease runtime efficiency (if possible!). The only language we felt we could possibly do this with is PASCAL [7,10] (and languages based on PASCAL). In particular, the subrange types, the scalar types and the simple record types (no variants) are important and necessary in our work. Beyond this, we assume that our PASCAL-like base language is more "dynamic" than PASCAL; we assume ALGOL-like block structure, arrays whose bounds are determined when the blocks in which they are declared are entered, subrange types like 1..N where N is a variable, and so on. We will use different syntax notations at times; we rely on the reader's knowledge of programming languages to understand our meaning.

Throughout, it should be remembered that the effects on all parts of the language have not been fully considered, and might change or restrict our ideas somewhat. We are not proposing a fixed extension to PASCAL to be implemented immediately; we are exploring some useful ideas to be used in future languages. Many ideas are only briefly explained here and are covered in more detail in the second authors' thesis [4].

Section 2 is devoted to several aspects of data types: programmer defined data types, defining operations on them through "overloading", implicit conversions between data types, and the domain of an array as a data type. The idea of "overloading" is not new; the MAD language used the notion [9], and Hoare [6] also discusses it.

In Section 3 we discuss the general problem of iteration and notations for it, while Section 4 introduces our generalized procedure, where the type of a formal parameter is not fixed. Section 5 briefly discusses implementation of our ideas. In most cases no runtime efficiency is lost when using the new features.

Section 6 contains a discussion of some problems, such as misuse of these features and proving programs correct.

"sum of elements" makes sense.

2c. Defining Subtypes. A problem exists in the last procedure SUM (2.3); the assignment statement is invalid because S has type complex while 0 is an integer. The programmer should have some way of saying that it is alright in this case, since 0 is in a sense a complex number.

Three general solutions (at least) exist to this problem of type conversion. First, we could require the programmer to write this as S := complex(0,0). This is too harsh; if 0 is usually regarded as a complex number this shouldn't be necessary. After all, even PASCAL allows A := 1 where A is real, simply because the integers are a subset of the reals. For reasons which will be even more apparent near the end of the paper, this solution is too strict and inflexible, and can make programming more difficult rather than simpler.

The second solution is to allow the programmer to define implicit conversion between any two types he chooses. Suppose the name implicit-conversion is reserved for this purpose. To describe implicit conversion from real to integer values, the programmer would write

```
        function  implicit-conversion(X:real): integer;
                  implicit-conversion := trunc(X);
```

in order to make it legal write I := 5.5 if I has type integer.

However, implicit conversion should only occur when one type is a subset of another -- for example, form 1..10 to integer, from integer to real, and from real to complex. Otherwise roundoff errors and runtime inefficiencies creep in.

the function. Note that we are also extending PASCAL in another direction besides overloading; the value of a function may be of <u>any</u> data type.

One can overload an operator as often as he wishes, provided that no two such definitions have the same set of parameter types. When the human reader (or the compiler) reads an expression X + Y, he must be able to unambiguously determine from the known types of X and Y, what is the meaning of "+".

2b. <u>Procedure and Function Overloading</u>. Just as one should be able to "overload" operators, so one should be able to overload any procedure or function name. For example, suppose we have defined a function

```
(2.2)          function    SUM(A:array of integer; N:integer) : integer;
                    var    I, S:integer;
                  begin    S := 0;
                           for I := 1 to N do S := S + A[I];
                           SUM := S   end
```

then one should be able to define

```
(2.3)          function    SUM(A:array of complex, N:integer) : complex;
                    var    I : integer;
                    var    S : complex;
                  begin    S := 0;
                           for I := 1 to N do S := S + A[I];
                           SUM := S  end
```

The point is that function SUM produces the sum of the elements of an array, and we should be able to define it for <u>any</u> set of parameter types which are arrays of values upon which the operation defining other operators, or operation on record types.

Several researchers are attempting to develop notations to support the idea that the definition of a new data type should include definitions of the basic operations [among others, 8,9]; based on SIMULA's class concept [1,2,3]. Our purpose here is not to discuss the whole notation for defining a new data type, but to indicate one possible notation for the definition of an operator on a new data type - overloading.

2a. Operator overloading. If one defines a new data type, say

```
complex = record rpart, ipart : real end
```

whose meaning is to be taken in the mathematical sense of a "complex", and secondly defines complex variables:

```
var X, Y : complex
```

then one should be able to use the conventional operations +, -, * and / on such values. To do this, we suggest overloading the operator, using a definition like

```
(2.1)         function + (X,Y : complex) : complex;
                 begin  result.rpart := X.rpart + Y.rpart;
                        result.ipart := X.ipart + Y.ipart
                 end
```

This then allows the programmer to add two complex values together using conventional notation, e.g.: X + Y + complex (5,3). This definition is much like any other function definition, except that the name result is used to designate assigning to the value of

2. New Data Types and Overloading Operators

PASCAL and (some other languages) allow the programmer to define new data types. Besides arrays, the most important kinds of new data types are (1) the scalar data type - an ordered set of values, (2) the subrange type - a subrange of any scalar data type, and (3) the record type - a structure consisting of a fixed number of components, possibly of different types. Examples are:

```
(1) scalar type:   suit = (clubs, diamonds, hearts, spades)
(2) subrange type: one_to_9 = 1..9
(3) record type:   complex=record rpart:integer; ipart:integer end
```

Each data type is defined (roughly) as the set of values which have that type. Lately, computer scientists have come to the conclusion that a data type is not only a set of values, but consists also of the set of basic operations that can be applied to those values. The type integer is the set of integers with the operations +,-,*,/ and the relational operators; Boolean is the set {false, true} together with the operations and, or and not.

It is felt that the operations := (assignment) and = (equality) should be automatically defined for any data type, programmer-defined or otherwise.

PASCAL does allow these two operations := and = on all data types (except file). Scalar types (like suit above) are automatically ordered and thus one can apply relation operators to them. Subrange types (like 1..9) automatically have all operators extended to them from the integers. However, there is no easy way of

Thus, it would be better to use another notation to indicate that one type is a subset of another, and with this indication allow the programmer to specify the conversion. We propose

```
subtype  (id: type_1): type_2;
   {statement which indicates
    how to convert from type_1 to type_2}
```

Or, if you wish to use a more standard, function-like notation:

```
(2.4)           function  subtype(id: type_1): type_2;
                   {statement which indicates
                    how to convert from type 1 to type 2}
```

so that subtype is a standard function which can be overloaded. For example:

```
function  subtype(X:real): complex;
   subtype := complex(X,0);
```

This at least gives the impression that one is defining how types interact and not just how to convert from one type to another. It is also easy to check for and warn against circularity, as for example in

```
function  subtype  (X:real): complex;
             subtype := complex(X,0);
function  subtype  (X:complex): real;
             subtype := complex.rpart
```

Unfortunately, such circularity is not always a mistake, although for purposes of clarity and efficiency we might not want to allow it. Consider two types, rectangular coordinates and polar coordinates, defined by

```
rectcoord  = record xvalue, yvalue : real end
polarcoord = record rho, theta : real end
```

These two types are equivalent in that you can transform a value from one type to another and back again, but almost certainly some information (accuracy) will be lost in doing so on any computer.

2d. A New Data Type: The Domain of an Array. Quite often a program must iterate over elements of an array, performing some operation on each. The notation for doing this depends on the array in question - a single loop for a one-dimensional array, two nested loops for a two-dimensional array, etc. However, since the task to be performed - iterating over the elements of an array - are the same in all cases, we should be able to use the same notation.

We introduce a new data type here which will facilitate this. The full usefulness and flexibility to be gained by it will not become apparent until later in the paper.

Consider any array A to be a function, from its domain of possible subscript values to the corresponding array element values. In this case, the domain of A is a set of values, which we will represent as domain(A), and it can be considered to be a type. Thus, given

```
var A : array[1..N] of integer
```

we can declare another variable I as

```
var I : domain(A)
```

in which case I can be used as a subscript: A[I].

Let us consider another case:

```
suit = (clubs, diamonds, hearts, spades);
var B: array[suit,1..13] of integer
```

Here, B is a two-parameter function, accepting 2 arguments. If we declare

```
var J: domain(B)
```

then J is a pair of variables; the first takes on values of type suit, the second of type 1..13. In this case B[J] would be a perfectly legal variable reference, as would B[spades,12].

What type does J really have, assuming we want to add this feature to PASCAL? The revised PASCAL report says that

a record type is a structure consisting of a fixed number of components, possibly of different types.

This is exactly what the domain of an array must be, with one component for each dimension of the array, and each component having the type given to the corresponding subscript of the array. To get a standard notation, we arbitrarily say that for an array A declared by

var A: array[t_1,t_2,...,t_n] of ...

where the t_i are types, declaring variable J as

```
var J: domain(A)
```

is exactly equivalent to

var J: record comp1:t_1; comp2:t_2; ...; compn:t_n end

In order to be useful, one must be able to use records in subscript positions. So let us extend our language to allow this. Suppose we declare

```
var C: array[1..10,1..10,suit] of ... ;
var K: domain(C);
var L: 1..10;
var M: record compl: 1..10; comp2: 1..10  end;
var N: suit
```

Assuming that variables K, L and M have been assigned values, the following are all legal references to indexed variables:

```
C[K]
C[L, L, spades]
C[M, N]
```

Thus, one can mix simple variables and records in subscript positions, as long as the individual types of the components match. While this feature *can* be used, it won't be used often. The primary purpose of this added flexibility is to allow a variable I of type domain(A) to be used as a subscript as A[I], no matter how many dimensions A has.

This introduction of the data type domain(A) brings to light a minor mistake in the PASCAL manual [7] (not the report). The manual says that, for example.

```
var A: array[1..20] of array[1..10] of integer
```

and

```
var B: array[1..20,1..10] of integer
```

are equivalent (except for the name of the array); that the notation used to declare B is just an abbreviation of the notation used to declare A.

This is false; A is a one-dimensional array whose array elements happen to be arrays, while B is a two-dimensional array. Obviously,

```
                    var I: domain(A)
is equivalent to    var I: record comp1: 1..20 end
while               var J: domain(B)
is equivalent to    var J: record comp1: 1..20; comp2: 1..10 end .
```

2e. Discussion. There are problems which should not be overlooked, and the foregoing should not be taken as the end result to be put into a language. The effects on other parts of the language must be completely understood. One problem arises with defining subtypes. Suppose we define

```
function subtype(X: integer): complexi;
         subtype := complexi(X,0);
function subtype(X: real): complexr;
         subtype := complexr(X,0);
function subtype(X: complexi): complexr;
         subtype := complexr(X.rpart,X.ipart)
```

Now suppose we have

```
var A: complexr;    A := 0;
```

Should the integer 0 be converted via

```
integer → complexi → complexr, or via
integer →   real   → complexr
```

Should it matter which route is taken? Probably not, since we are working only with subtypes and hence conversion implies no loss of information. But the matter must be further studied.

One rather big point concerns the overloading of operators and procedures. The programmer must be taught to use the overloading in a way which keeps programs understandable. For example, defining addition + of a new data type polynomial as multiplication "would be wrong".

We have the feeling that the same axioms should be used to define + irrespective of the data type, and that these axioms should be used in the proof of correctness. To illustrate what we mean, consider the procedures SUM defined earlier in (2.2) and (2.3), and suppose we see

```
... X := SUM(A,10); ...
```

in a procedure. The assignment statement axiom used in understanding this statement assumes some idea - an axiom if you will, of what SUM does in terms of A and 10. _This idea should be the same no matter what the type of A is_, so that we can prove the program correct (and thus understand the program) irrespective of the type of A.

We will discuss this more when introducing more general procedures where the type of a parameter can be a parameter.

One reason that we have decided not to include a complete notation for defining operations as part of the definition of a data type is that this is a difficult problem. For example, one idea that has been circulating is to use a syntax like

```
<type identifier> = begin <typical PASCAL-like type declaration>;
                                   <definition of operation 1>
                                    .
                                    .
                                    .
                                   <definition of operation 2>
                    end
```

However, this has its problems. The defined operations are to apply to values of the defined type. If this is so, then where does one put a definition of an operation which includes as operands values of two different types? For example, if exponentiation is not defined, we might want to define it:

```
function exp(X: real; Y: integer): real;
    {definition of X^Y}
```

We feel our notation for overloading operations and procedures is a natural extension of the PASCAL procedure notation. It should be easy to learn, easy to use naturally, and can be implemented efficiently.

3. Iteration

There are essentially two types of iteration. One requires us to iterate an unknown number of times until some condition is met (e.g., the while loop). With the other, we iterate some statement as some "index variable" takes on a set of fixed values, known beforehand. Some examples of the latter are

```
for i := 1 to N do S;
for i := N downto 1 do S
for x in X do S          (where X is a set variable)
```

Since all these different notations imply the same thing - iteration over a known, fixed set - it would be wise to have just _one_ notation to represent them all, something like

for <variable> _in_ <set of values> _do_ S

where the <set of values> and <variable> have the same type. If the <set of values> has an ordering, then this ordering is used in assigning values to the index variable <variable>; otherwise no ordering is implied.

For example, the three loops above would be written as

for i _in_ 1..N _do_ S
for i _in_ _reverse_ 1..N _do_ S
for x _in_ X _do_ S

where _reverse_ 1..N is the set 1..N but with the conventional ordering reversed.

Hoare [6] has already discussed such a general iteration statement. His axiom for _for_ x _in_ X _do_ S where X is an unordered set is as follows (using his notation, [ ] denotes the empty set, and if s is a set, I(s) is an assertion about the set s):

$$\frac{s_1 \subset s,\ x \in (s - s_1),\ \{I(s_1)\}\ S\ \{I(s_1 \cup x)\}}{\{I([\ ])\},\ \underline{\text{for}}\ x\ \underline{\text{in}}\ X\ \underline{\text{do}}\ S \quad \{I(X)\}}$$

If X is an ordered set, we then use the following proof rule. Let $X = \{x_1, x_2, \ldots, x_n\}$ where $x_i < x_{i+1}$, $1 \leq i < n$. Then

$$\frac{1 \leqslant i < n,\ \{I([x_1, \ldots, x_i])\}\ S\ \{I([x_1, \ldots, x_{i+1}])\}}{\{I([\ ])\}\ \underline{\text{for}}\ x\ \underline{\text{in}}\ X\ \underline{\text{do}}\ S\ \{I(X)\}}$$

The neat thing about this is that we have one rather than several iteration notations, and two iteration axioms (one for the ordered, one for the unordered case) rather than several. In effect, we have reduced the problem of defining different kinds of iteration to the simpler problem of defining different sets of values (and orderings on them). Some notations for <sets of values> are

notation	meaning
any expression E	the set consisting of the current value of E.
M..N	the integers M, M+1,..., N, natural ordering.
reverse SV	where SV is any set of values with an ordering; the result is the same set with the ordering reversed. For example, reverse M..N = N, N-1,...,M+1, M.
X	where X is a set variable. This set is unordered.
suit	where suit is a data type defined as suit = (clubs, diamonds, hearts, spades). This set is ordered in PASCAL, in the order given.
any finite data type	the value of that data type, ordered if the data type is ordered. suit, defined above, is an example.
$<SV_1>, <SV_2>, \ldots, <SV_n>$	where the $<SV_i>$ are sets of values, each ordered.[1] The resulting set of values has the obvious ordering: first those in $<SV_1>$, in their order, then those in $<SV_2>$ in their order, etc.

The example

for I in suit do S

illustrates a simple but useful and elegant extension to PASCAL.

Since a data type is, in part, a set of values, we can use any such finite data type as a set of values. Note that, for example, 1..5 is already a (subrange) type, and that PASCAL allows

```
for I in 1..5 do S   (but written for I := 1 to 5 do S)
```

so that this extension is fairly natural.

Note that records are data types. Suppose we declare

```
suit = (clubs,diamonds,hearts,spades);
card = record s: suit; value: 1..13  end;
var I: card
```

Then we can write a loop which iterates over all 52 possible card values of a deck:

```
for I in card do S
```

or perhaps just

```
for I in  record suit; 1..13 end  do S
```

Finally, note that domain(A) is a type, and can thus be used as a set of values. Hence, we can write

```
var A: array[1..N,1..M] of integer;
var I: domain(A);
for I in domain(A) do A[I] :=0
```

and even

```
var B: array[suit,1..13] of integer;
var I: domain(B);
for J in domain(B) do B[J] :=0
```

Thus, the same notation can be used to iterate over the elements of any array, regardless of the number of dimensions and the type of the individual subscripts. Should one wish to specify a definite ordering (remember, domain(A) is an unordered set), one can resort to the old notation:

```
for J.compl in suit do
    for J.comp2 in 1..13 do ... B[J] ...
```

or

```
var L: suit
var K: 1..13;
for L in suit do
    for K in 1..13 do .... B[L,K] ....
```

As a final example of this flexibility, the following can be used to sum the elements of any integer array A, no matter what its dimension or the types of the subscripts.

```
var I: domain(A);
var S: integer;
S := 0;
for I in domain(A) do S := S + A[I]
```

4. Letting Types of Parameters be Parameters.

Current languages with types require that the type of parameters of a procedure be fixed at compile-time -- at least a procedure cannot one time return an integer and the next time a real value. For example, a procedure

```
procedure SORT ( A: array of integer, N: integer);
    {body of procedure which sorts array A[1..N]}
```

cannot be used to sort an array of reals or an array of character variables. Historically, a parameter was associated with an argument of a particular fixed type. This was correct when we viewed our data types as pure sets of values -- a procedure should only work on one kind of value.

When we view a data type as a set of values, *plus basic operations on them*, our view changes somewhat. We abstract away from the idea of a procedure operating on a parameter of a particular data type to the idea of a procedure operating on a *parameter of any data type for which certain basic operations have been defined*. This makes sense. For example, if we write a procedure to sort an array of values, we are not interested in whether the values are integers or reals or what have you, but instead we are interested in - and our proof of correctness of the sort procedure depends on - the fact that the assignment operator := and the ordering operator $\leq$ are defined on the type of array values.

Thus, we should like our programming language to support this abstraction from types to basic operators on values of a type, to support the idea of letting types of parameters be parmeter. One problem, of course, is efficiency. We want to introduce the idea into the language so as *not* to worsen run-time efficiency. We believe the notation and ideas described below satisfy this goal to a large extent, although, of course, the compiler will have more work to do.

First, let us summarize the context in which we'll be working. To keep things simple we will only consider PASCAL variable parameters (call by reference) whose types are programmer-defined as standard scalar types, subrange types, array types and record types without no variants. We will assume that arrays can be more dynamic than in PASCAL, that the bounds of an array parameter need not be given explicitly. For example, we use the notation

<u>procedure</u> X (<u>var</u> A: <u>array</u>^{1} <u>of</u> integer, <u>var</u> B: <u>array</u>^{2} <u>of</u> real);

to denote an array A of one dimension and an array B of two dimensions.

We will assume that for a value parameter of type t, the corresponding argument can be of any subtype (for which implicit conversion is defined). For a variable parameter (defined using <u>var</u>), the argument must have the exact same type (except for array bounds).

To allow the programmer to reference the bounds of an array we use the standard functions

(4.1)	ℓbound(B)	returns a record of type domain(B) consisting of the lower bounds of all subscript positions,
	ubound(B)	returns a record of type domain(B) consisting of the upper bounds of all subscript positions,
	ℓbound(B,i)	returns the lower bound of the i<u>th</u> subscript position,
	ubound(B,i)	returns the upper bound of the i<u>th</u> subscript position,
	rank(B)	returns the number of dimensions of B (o if a scalar).

It may also be advantageous to allow the standard functions

(4.2) min(t) and max(t)

which return the lowest and highest values, respectively, of the finite, ordered type t.

Now let us indicate how types of parameters are made themselves parameters. In the <formal parameter section>, any (reasonable) part of the specification of a parameter type may be replaced by an identifier within braces <> to indicate that the type will depend on the call. This includes the type itself, a subtype, or the number of dimensions of an array. For example,

```
(1) procedure X (var A: <z>);
(2) procedure X (var A: array^2 of <z>);
(3) procedure X (var A: array^<N> of integer);
(4) procedure X (var M: record C1: integer; C2: <z>  end;
                 var N: <z>)
```

The first case states that parameter A may have any type. The second, that A is a 2-dimensional array of any array element type. Procedure heading (3) indicates that parameter A is an array of any number of dimensions, of type integer. Case four indicates that component C2 of parameter M must have the same type as parameter N.

Within the procedure itself, the identifier within braces (e.g., z within <z>) may be used as any type.

To illustrate, we write a procedure which will interchange the values of any two variables of the same type, no matter what the type is.

```
{interchange the values of A and B}
procedure SWAP(var A,B: <type>);
    var T: type;
    begin T := A; A := B; B := T   end
```

Suppose we write a function which computes the greatest common divisor of two integers, but leave the type variable:

```
function GCD(m,n: <type>): <type>;
    begin if n = 0 then GCD := m else GCD := GCD(n,m mod n)
    end
```

This function will produce the GCD of two values of any type on which mod is suitably defined. In particular, this may be done for polynomials. Let us define polynomials of degree 50 or less by

```
polynomial = array[0..50] of integer;
```

where the ith array element is the coefficient a_i of x^i in the polynomial $a_0 + a_1x + a_2x^2 + \ldots + a_{50}x^{50}$. We can indicate that integers are polynomials by the definition

```
function subtype (X: integer): polynomial;
                 var i: integer;
                 begin subtype [0] := X;
                       for i in 1..50 do subtype [i] := 0;
                 end
```

Suppose we also suitably overload the mod function:

```
function mod(X,Y: polymonial): polynomial;
 {body to compute mod of two polynomials}
```

Then the above procedure GCD will work correctly, without change, for polynomials of degree 50 or less as well as integers. Thus, if we declare

```
var A,B,C:  polynomial
```

and suitably initialize them, we can execute

```
                    C := GCD(A,B)
```

Perhaps the most exciting extension is the variability of array parameters. Consider the function

```
(4.3)   function MAX(X: array<N> of <type>): <type>
             var I: domain(X);
             var M: type;
             begin M := min(type);
                   for I in domain(X) do
                       if M < X[I] then M := X[I];
                   MAX := M;
             end
```

Note the use of min(type). One may object to this if type is integer or real. But remember that these types are finite on any machine, and thus min(integer) and min(real) should return the lowest representable integer or real in this implementation. One could also use X(ℓbound(X)) instead of min(type); this would be the first array element value in the conventional sense.

This single function returns the maximum value in the array for any array of any dimension, with any subscript types, and with any ordered array element type. For example, if we have the following variables initialized,

```
             suit = (clubs, diamonds, hearts, spades);
             var A: array[1..10] of suit;
             var B: array[suit, 1..13] of integer
```

then the following are legal function designators

```
             MAX(A)          and              MAX(B)
```

This truly reduces a lot of programming detail, and we believe can be implemented efficiently.

5. Implementation

Questions of implementation boil down mainly to the question of whether or not the compiler can determine the exact type of every reference to a variable at compile time. If it can, then generating the correct code for overloaded operations, determining implicit conversions and inserting code for them, and other such problems, are quite straightforward. If the compiler can *not* determine the type of a variable, then information about the type must be coded in some sort of dope vector and this dope vector must be interpreted at runtime

We will show later on how the compiler can determine type of variables in most cases -- pathological cases involving recursive procedures are the ones which give trouble. But first let us discuss some cases where some sort of interpretation is actually *more* efficient. Consider the function to sum elements of an array of integers:

```
function: SUM(var A: array<N> of integer): integer;
          var J: domain(A);
          var S: integer;
          begin S := 0;
                for J in domain(A) do S := S + A[J];
                SUM := S;  end;
```

Domain(A) is an unordered set, so that the compiler can determine the order in which the various subscript values J should be used. Whether the array has 1,2,3 or more dimensions, clearly the compiler should implement this as a *single* loop, thinking of the array as a one-dimensional array, and J as a single, simple integer variable.

The necessary information to do this (the number of array elements and address of the first one) would appear in the dope vector passed to it for the array argument.

Note that with our new notation, subscript checking is not needed; the compiler knows that J always contains a valid subscript for A.

Of course, the compiler would have to check the procedure body carefully; variable J can be used only as a subscript of A, its components cannot be explicitly referenced, and so forth. Some research is necessary to determine how much the compiler must know to be able to perform such optimizations, and to develop algorithms for extracting such information from a program. But this form of optimization would certainly increase runtime performanc and reduce runtime storage requirements.

Now let us consider how a compiler can process procedures with parameters of "variable" type, and calls on such procedures. Let us, first of all, consider programs with no recursive procedures and no procedure names as parameters. Consider the program given below:

(5.1)

```
        .
        .
        var A,B: integer;
        var C: real;
        procedure P(var X: <type>);
            var Y: type;
            begin . . . end;
        P(A); P(C); P(B);
        .
        .
```

The compiler can textually change this into the following program:

(5.2)

```
.
.
var A,B: integer;
var C: real;
procedure P^1(var X: integer);
    var Y: integer;
    begin . . . end;
procedure P^2(var X: real);
    var Y: real;
    begin . . . end;
P^1(A); P^2(C); P^1(B);
.
.
```

Thus, we look at the original procedure P as a kind of macro. We group the calls on P together so that each group has the same list of argument types, and for each group we generate a procedure P' from P, with the parameter types filled in.

This can be done since there is no recursion, in much the same way that macros are handled in other languages.

At first glance, one might object to this extra processing by the compiler. But remember this; without this generalized procedure facility, the programmer would be writing and running version (5.2) anyway. The use of generalized procedures could lead to less use of auxiliary storage, fewer cards punched, and less computer time, because the programmer works with shorter, less detailed, more general programs.

Now let us consider the use of procedure names as parameters (but no recursion), and where the names are of these general procedures. The simplest way to see that this need not lead to decreased runtime efficiency is to note that the compiler can delete the procedure name parameter as it is making a copy of the procedure. For example,

```
        .
        .
        procedure P(procedure X; Y: integer);
           begin . . . X(...) . . . end;
          .
          .
          P(A,C);
          .
          .
```

becomes

```
          .
          .
        procedure P'(Y: integer);
            begin . . . A(...) . . . end;
          .
          .
          P'(C);
          .
          .
```

Of course, you may not want to do this in a compiler since it can lead to a proliferation of procedures, but it shows in principle that all the work can be done at compile-time.

When we include recursion, the problem is that the macro-type processing described earlier can be a non-terminating process, because it may lead to an infinite number of procedures. Consider the following:

```
procedure P(A: <type>; N: integer);
    var B: array[1..10] of type;
    begin if N ≤ 10 then
              begin . . . P(B,N+1); . . . end
          end
```

Any call of this procedure will eventually terminate at runtime since the second argument is increased by 1 each time and recursion stops when the second argument becomes greater than 10. But for a call P(1,1) the above-described compile-time macro processing leads to an infinite number of procedures with headings

```
procedure P^1(A: integer; N: integer);
procedure P^2(A: array^1 of integer; N: integer);
procedure P^3(A: array^1 of array^1 of integer; N: integer);
                    . . .
```

In general, the problem of determining whether (a program will generate an infinite number of types at runtime is undecidable [4] and hence we have no general way of knowing when to stop generating the above procedures P^1, P^2, P^3, . . . at compile-time. However, the compiler can certainly detect the possibility of having an infinite number of types, and refuse to compile such cases.

This infinite type problem arises in several cases; they all boil down to the idea that a particular parameter of a recursive procedure is possibly called with either

(1) an array with higher dimensions,

(2) arrays of higher and higher nesting (i.e., array of array of ...),

(3) records whose component types are themselves of higher and higher "complexity",

(4) subrange types of higher and higher cardinality (e.g., 1..10, 1..11, 1..12, ...), or

(5) mixtures of the above.

Some of these problems can be handled by considering several similar types to be equivalent - treating all subrange types using a dope vector, and treating arrays of different dimensions but of the same array element type as the same, using a dope vector. The other problems can be resolved by restricting somewhat the specifications of types of parameters, and how such types can be used in declarations, and secondly by having the compiler recognize the situations described above and stop compiling, in effect making them illegal.

With these solutions, the compiler can then determine the type of each variable reference in a program.

6. Summary and Discussion

The main idea we started with in our work was that one should be able to write procedures with parameters of variable types. "Abstraction" can be thought of as a generalizing process in which one concentrates on similarities between things and groups them based on these similarities. This is precisely what we are trying to do by generalizing the procedure concept. And we feel the idea is useful enough that it should be incorporated into higher level languages at some point.

This led us to look at other problems -- a language in which to embed the idea, how to do it clearly, overloading operators and procedures, implicit conversion from sub- to super-types, a more uniform notation for the typical kind of iteration over a fixed set. All these ideas are important; leave one out and you have a less flexible system.

For example, we give below a subtype definition which encompasses almost all the ideas discussed. The definition allows one to use any value i as an array, all of whose elements contain i. For example, one could write A := 0 where x is an integer array.

```
function subtype (X: <type>): array<N> of <type>;
   var I: domain(subtype);
   begin for I in domain(subtype) do subtype[I] = X  end
```

This illustrates quite clearly the flexibility of the extensions, but also indicates that one must be quite careful when extending a language this way. The consequences of an extension are not always seen at first sight.

There are, of course, many problems. Foremost is the fact that a programmer can use these features in an awkward, inconsistent, inefficient manner. For example, if he writes a SUM function to sum elements of an integer array, and overloads it with a SUM function which multiplies two polynomials together, he is being inconsistent. He is not using abstraction correctly; he is not using it to express the similarity of things.

Let us state this in another way. Suppose we write a procedure which uses the plus operator on parameters:

```
procedure P(X,Y: <type>);
          .
          .
          ... X+Y ...
          .
          .
```

In producing a proof of correctness of the body of P -- in terms of its outputs relative to its inputs -- we are making certain assumptions about what + does. The operation "+" satisfies certain axioms which are used in the proof of correctness. Our contention is that there should be only one proof of correctness of the above procedure P, regardless of the <type> of X and Y; the programmer must insure that, if P is called with arguments of type t, that the operator + defined over t satisfies the necessary axioms. The proof of correctness is the thing which is similar for all types <type>, and we can ignore any differences in possible types which don't affect this proof.

One may argue that APL and SNOBOL, two typeless languages, have just such a feature. Anything of any type can be passed as a parameter. These languages are at a disadvantage from this standpoint. First of all, everything must be interpreted at runtime, which is inefficient. Secondly, one cannot tell just by looking at a procedure just what it does; it will depend too heavily on the inputs to that procedure. One cannot even parse a simple expression and understand an operator. For example, in APL the expression B+A gets parsed differently depending on whether B is a variable or a one-argument function.

Types are important when one considers readability, understandability, and proofs of correctness.

There are many unresolved issues in our extension, which depend on the base language in question. One concerns iteration. With our new notation there is, as yet, no good way of expressing

<u>for</u> i := 1 <u>to</u> N <u>by</u> 2 <u>do</u> S

One way is to include sets of values in the language, using set notation such as

{2i+1 | i <u>in</u> 0 .. N/2-1}

and {-i | i <u>in</u> 1 .. N}

Here, the general form would be

{f(i) | i <u>in</u> <set of values>}

where f(i) is an expression in i. If the set of values is ordered, X_1, X_2, .., X_n, then the new ordered set is $f(X_1)$, $f(X_2)$, ..., $f(X_n)$. Care must be given to keeping this flexible, but yet efficient to implement and efficient to execute.

The whole question of data types arises. Is 1..10 a data type, or is it just the data type <u>integer</u> with an extra qualification which the compiler should check (or not check if the user doesn't want it checked, must like subscript checking)? If 1..10 is a data type, why not also 1..10 <u>union</u> 15..20, or {1,3,5,7,9}? The questions to be answered are: what, theoretically, is a data type, and secondly what data types can we implement efficiently?

In section 2 we argued that programmer-defined implicit conversion from a subtype to a supertype was necessary. There are problems with this. For example, the programmer may be unaware of hidden inefficiences due to such conversions. This is a problem for the compiler; it should warn the programmer of them. Also, since many of such implicit conversions deal with constants, they could be performed at compile-time; if possible, perform an in-line macro-like substitutions and normal optimization will take care of it.

A much more important problem is when the subset-superset definitions of types lead to multiple paths of implicit conversions, or even circular paths. Should the types rectangular coordinates and polar coordinates be subsets of each other? Why not? We don't have a satisfactory answer to this. In this case it seems all right, but in general it can lead to inefficient, inconsistent programs.

One idea we were toying with was using a different notation for referencing fields of a record. Suppose we have

```
complex = record rpart, ipart: real  end;
var A: complex
```

A.rpart seems like a good notation. But A(rpart) would be more uniform, yielding the same notation for functions, arrays and records. After all, the above variable A can be thought of as a function with domain {rpart, ipart}. This would allow us to iterate over fields of a record A using domain(A) just as we do with arrays. However, implementation becomes quite inefficient, with much interpretation at runtime, since the field of a record can have different types.

In concluding, we feel the features described are useful and flexible, and together represent a significant step forward in programming language design. But much work needs to be done to incorporate them clearly and efficiently into a language.

References

[1] Dahl, O.J. and C.A.R. Hoare, Hierarchical Program Structures. Structured Programming, Dahl, Dijkstra and Hoare, Academic Press, 1972, 175-220.

[2] Dahl, O.J. and K. Nygaard, "Simula - an Algol-based simulation language", CACM 9 (Sept 1966), 671-678.

[3] Dahl, O.J., B. Myhrhaug, and K. Nygaard, "The Simula 67 Common Base Language", Norwegian Computing Centre, Forskningsveien 1B, Oslo 3.

[4] Gehani, N., Ph.D. Thesis, Cornell University, August 1975.

[5] Hoare, C.A.R., "A Note on the for Statement", BIT 12 (1972), 334-341.

[6] Hoare, C.A.R., "Hints on programming language design. Invited Address, SIGACT/SIGPLAN symposium, Oct. 1973.

[7] Jensen, K. and N. Wirth. PASCAL User Manual and Report. Springer Verlag Lecture Notes in Computer Science 18, 1975 (2nd edition).

[8] Morris, J.H., "Types are not sets". SIGPLAN Symposium on Principles of Programming Languages, (Oct. 1973).

[9] Organick, E.I. A Computer Primer for the MAD Language. Univ. of Mich., Ann Arbor, Mich. 1961, and Computer Data Proc. Center, Univ. of Houston, Houston, Texas 1961.

[10] Wirth, N. The programming language Pascal. Acta Informatica 1, 1971, 35-63.

[11] Wulf, W.A. ALPHARD: Towards a language to support structured programs, Dept. of Comp. Science, Carnegie-Mellon, Pittsburgh, Pa. (April 1974).