GRIES77
Gries, D., Gehani, N.; Some Ideas on Data Types in High-Level Languages; CACM 20, 6 (June 1977) pp. 414-420.

This paper takes the notion that a data type is not only a set of values, but also a set of primitive operations on these values. It then explores this notion by applying it to two concepts: the domains of arrays and generalized procedures.

The domain of an array is the set of legal subscript values of the array. The authors write the domain of an array a as domain(a) and consider it to be a data type. Thus, no matter what dimension an array has or what type its indices are, declaring an index i as:

```
var i:domain(a)
```

makes i the correct type to index a. For example, using PASCAL-like notation, if we declare the following:

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

then j is a pair of variables, the first taking on only legal values of suit, the second taking on any integer in the range 1 to 13.

An interesting extension of this comes in iteration. The authors define two general types of indexed loops:

```
for <variable> in ordered <ordered set of values> do S
for <variable> in unordered <set of values> do S
```

where "unordered" indicates that the order in which values in the <set of values> are assigned to <variable> is immaterial. Thus, for example, to sum over the array b (assuming b has been assigned values), we can write:

```
var s:integer;
s := 0;
for i in unordered domain(b) do
   s := s + b(i);
```

The second concept is a generalized (or generic) procedure which operates on a parameter of any data type for which certain basic operations have been defined. For example, if we have a procedure that sorts an array of values, then it makes no difference whether the values are integers, reals, characters, or any other data types as long as the assingment operator, :=, and the ordering operator, <=, are defined on that type of array value.

This concept brings up the problem of type conversions. For example, suppose that the above loop that sums the elements of an array was a procedure that summed any array values for which the operator + was defined. Then we would declare s to be of the same type as the array values, but then the statement

```
s := 0
```

could cause a type "clash" since 0 is of type integer. To solve this problem, the authors argue in favour of explicit conversion of types by primitive operations that do the conversion. Thus, to initialize s, we could write:

```
s := convertfrominteger(0).
```

Convertfrominteger would be an "overloaded" function in the sense that many functions by that name would exist, one for each type t to which it would be possible to convert from integer. It would be the compiler's job to decide from context which function is actually to be called.

The paper then discusses the implementation of the proposed features and puts forth one method of compilation that seems quite feasible.

The notions proposed in this paper make use of abstraction in the sense that they hide details. Further, they generalize some existing high level language features into more "natural" forms. These notions are relevant to the current evolution and development of programming languages as they are aimed at simplifying and making more consistent the programming tools available to a user while at the same time increasing the power of these tools by generalizing them.