Types

In a set-theoretic language much of the work currently done by types can be handled by the set mechanisms already in the language. For instance, what is generally known as "type checking" could really better be called "domain checking". It is checking to make sure that the values of variables, parameters and outputs of subroutines, and values in data structures fall within certain declared classes of objects. These classes do not have to be types; they can simply be sets, if infinite sets are allowed in the language. This allows much greater latitude and requires fewer special type constructions. An object has only one type, but it may be a member of many sets. This kind of construction will allow us to check that the domain of a particular variable is any of the following:

(1) an integer ≥ 0
(2) an integer I, $1 \leq I \leq N$
(3) an integer or a string
(4) a particular set S in the program.

Now, if sets can be used in place of types for type checking, what role do types play? Each object in the system has one type associated with it. The typing serves to divide the value space of the language into disjoint subsets which can then be used for a number of valuable functions:

(1) They are a starting point for the domain checking which is mentioned above because each type is also a set.
(2) They provide the compiler with information to help in implementation.
(3) They give the programmer the ability to add declarative information concerning
   (a) debugging and error conditions -- see "Conditions on sets" in Newsletter 52 and proposal below.
   (b) what access paths are available on a given type of data structure -- see Section II.D in [1], and

proposal below.

(c) representation of data structures -- see the implementation facility description in [2].

(4) The same operator or programmer defined routine may be used for different purposes on different types.

The following is a proposal for the type mechanism in a language similar to SETL. One declares the type of a tuple by specifying the number of its elements and the domain of each, for example:

COMPLEX IS <REAL, REAL>

In addition, the programmer may want to specify names of functions which access the components of the tuple, for example:

COMPLEX IS <REAL_PART ∈ REAL, IMAG_PART ∈ REAL>

Then he can use these names conveniently to refer the parts of the tuple in an unordered way, see "Tuples" in Newsletter 52. One declares the type of a set by giving its domain, for example:

LOOK_AHEAD IS {TERMINAL ∪ LOOK_AHEAD}

MATRIX IS {<INTEGER, INTEGER, REAL>}

In addition, following the proposal "Conditions on sets" in Newsletter 52, one can attach a predicate to a set type declaration which specifies a condition which the set must satisfy. For instance, we can specify the set of all square matrices using a set-former-like notation as follows:

```
SQ_MATRIX IS {M = {<INTEGER, INTEGER, REAL>}|
  ∃LIMIT ∈ INTEGER | 1 ≤ ∀I ≤ LIMIT | 1 ≤ ∀J ≤ LIMIT |
  #{<I,J,->  ∈ M} = 1}
```

One may also parameterize such declarations as follows:

```
SQ_MATRIX(L) IS {M = {<INTEGER,INTEGER,REAL>}|
  1 ≤ ∀I ≤ L | 1 ≤ ∀J ≤ L | #{<I,J,R1> ∈ M} = 1}
```

We can then declare specific square matrices, such as

```
A ∈ SQ_MATRIX(10)
```

or create them at run time,

```
F(A) = SQ_MATRIX(I)
```

Perhaps the best way to handle the parametrized types is to make them simply functions which return a type as a value. This means, of course, that there should be objects of type "type" and type expressions allowed in the language. I do not present specific details here.

As a more complete example, we present here the type declarations for the algorithm given in Newsletter 56. First we declare a general form of sequence:

```
SEQ(T) IS {S = {<INTEGER,T>} | ∃ LIMIT ∈ INTEGER |
 0 ≤ ∀I ≤ LIMIT | #{<I,-> ∈ S} = 1}
```

The type declarations are then

```
PROD IS SEQ(STRING)
DEF IS {<PROD,STRING>}
STATE IS <PROD,INTEGER,INTEGER>
STATESET IS {STATE}
```

We could have provided more information about STATE's as follows:

```
STATE IS {<P ∈ PROD, NEXT ∈ INTEGER, ORIG ∈ INTEGER>|
          NEXT ≤ #P AND ORIG ≤ N}
```

Notice that this declaration gives us the ability to refer to the elements of a state by name (we didn't use this in the algorithm); i.e., if S is a STATE, we can talk in terms of

```
P(S)     NEXT(S)     ORIG(S)
```

The declarations for variables and parameters are then

```
INPUT ∈ SEQ(STRING)
N ∈ INTEGER
SS ∈ SEQ(STATESET)
ROOT_PROD ∈ PROD
```

Replace the reference to "PROD(G)" in Newsletter 56 to "PROD" and delete the declaration of G in order to make it work with these declarations. That was a mistake which I didn't detect until I wrote the declarations. This illustrates another one of their values.

There is an additional feature which is useful in connection with types. That is the ability to have a particular function perform different ways depending on the domains of its arguments. For example, when we define a function, we can specify domains for its arguments as follows:

```
DEFINEF F(A ∈ INTEGER, B ∈ STRING); body1;;
```

This normally means that if F is called with actual parameters of the wrong kind, it is an error. However, we can provide a second definition of F, i.e.

```
DEFINEF F(A ∈ INTEGER, I ∈ INTEGER); body2;;
```

This then means that if the parameters of F are both INTEGER's we use body2 instead of body1.

This has a number of uses. Let's return to an earlier example. We have a type

```
LOOK_AHEAD IS {TERMINAL ∪ LOOK_AHEAD}
```

We might want to define what it means for two members of such a set to "match" as follows:

```
DEFINEF MATCH(A ∈ TERMINAL, B ∈ TERMINAL); RETURN A = B;;
DEFINEF MATCH(A ∈ TERMINAL, B ∈ LOOK_AHEAD); RETURN A ∈ B;;
DEFINEF MATCH(A ∈ LOOK_AHEAD, B ∈ TERMINAL); RETURN B ∈ A;;
DEFINEF MATCH(A ∈ LOOK_AHEAD, B ∈ LOOK_AHEAD); RETURN A∩B = NL;;
```

This would be quite cumbersome to write using IF statements.

Another important use of this feature is in what I call an "implementation facility". This would essentially allow the programmer to specify how certain types of sets or tuples are to be implemented by rewriting the primitives on these types as programmer defined functions written in terms of lower-level sets and tuples. In order to do this one needs to be able

to add new function definitions for existing primitives in the language which apply only when those primitives are called with certain types as operands. See [2] for examples and details in the context of VERS.

[1] Earley, J. and Caizergues, P.
VERS Manual, Computer Science Dept.,
University of California, Berkeley, 1971

[2] Earley, J. Towards an Understanding of Data Structures. Comm. ACM Oct. 1971.