SETL Newsletter 152

J.T.Schwartz
July 5, 1975

An alternative design for a 'MIDL'-level language.

If, as now appears quite possible, SETL can be optimized to run at speeds within 1/3 or so of lower-level language speeds, it may be undesirable to contaminate 'pure' SETL with any efficiency-oriented extensions. On the other hand, if this level of performance cannot be attained, then MIDL-like extension of SETL will retain appeal. This newsletter will outline such an extension, but one which is more systematic and somewhat more powerful than MIDL. Let us call it MIDL'.

In designing such an extension, one must first describe the class of constructs which one feels can be handled efficiently enough to deserve incorporation in an efficiency-oriented semantic structure. The likely choices here are integers, reals, and strings; arrays and records with selection operators, and pointers. This is essentially the semantic repertoire of ALGOL 68; so we will assume that in addition to the operations of SETL, MIDL' incorates these ALGOL 68 constructs and the corresponding operations. In particular, we assume that structures (and arrays) like those of ALGOL 68 can be defined. However, we will assume that variable types are handled dynamically rather than statically. This accords better with the SETL approach, and should not impose any significant efficiency penalty, since global type analysis will in most cases succeed in determining almost all object types at compile time. (But type-testing operations should be available.)

In any given program some finite collection of structure types will be defined; each such type can be assigned some integer code which can be treated rather like a SETL type code, or, if the type has dimension parameters, like a structured vector consisting of a type code and several integers.

We propose the following rules:

a. SETL objects can have MIDL' pointers, integers, reals, and strings as members and components. MIDL' structures may not contain SETL objects; however, MIDL' pointers can point to SETL objects. To cause a MIDL' pointer to point to a (copy of) the SETL value e, write

p = :e.

For dereferencing a pointer to obtain a SETL value, an ALGOL like 'cast' operator +p (which dereferences and evaluates until a set, tuple, or structure is obtained) can be provided.

b. ALGOL 68-like selection, assignment, coercion, etc., operations can be provided as part of MIDL', with a syntax which allows them to be recognised. The dictions thereby introduced can be intermixed with the existing dictions of SETL in MIDL' programs.

To encourage disciplined use of the dictions (especially the low-level dictions) of MIDL', an auxiliary system of type declarations, which can regulate the pattern in which operations are applied to objects, may be useful. Such a scheme might be developed along approximately the following lines:

a. Allow indefinitely many new 'basic' object type names t to be declared. (These are merely names; the system of declarations in which they appear are distinct from the declarations of MIDL structures.) Build these basic type names up into a type name calculus as follows: if $t, t_1, \ldots, t_n$ are type names, then so are $\{t\}$, $[t]$ (respectively designating sets/sequences of elements of type t), $\langle t_1, \ldots, t_n \rangle$, $(t_1, \ldots, t_n)$ (denoting a programmed procedure with parameters of types $t_1, \ldots, t_n$), and $t(t_1, \ldots, t_n)$ (denoting a programmed function with parameters of types $t_1, \ldots, t_n$, returning a result of type t.) A type name for 'general SETL object' should also be available.

b. Allow the type (in the sense just introduced) of variables $x_1,\ldots,x_n$ to be declared, perhaps in the syntactic form

$$\text{dcl}\ \ x_1(t_1),\ x_2(t_2),\ldots$$

where t_1, t_2,... are type names.

c. Allow the types of the parameters expected by a programmer-defined procedure, and the type of value returned, to be declared. Possible syntactic forms are

define procname $(x_1(t_1),\ x_2(t_2),\ldots)$;
definef fcnname $(x_1(t_1),\ x_2(t_2),\ldots)$t;

where t, t_1, t_2,... are type names.

The following rules can then be applied: if types have been declared for the parameters of a procedure, the procedure can only be invoked with arguments (variables or expressions) of appropriate type. Moreover, variables whose type has been declared can only be used as arguments to procedures having parameters declared to be of the same type, and only expressions of appropriate type can be assigned to such variables.

d. Type declarations attached directly to the parameters of a procedure determine the type of argument for which the procedure can be called. Within the body of the procedure, each of its parameters will generally have a second, different, declaration. For example, one might write

definef sum(m_1(sparsematrix),m_2(sparsematrix))sparsematrix;
dcl m_1({<integer,integer, real>}), m_2({integer,integer,real});
...

The second declaration supplied for a procedure parameter exposes the parameter's internal details for manipulation within the procedure.

Note that the protections furnished by the rules just proposed can easily be evaded, since one can always 'convert' between types simply by writing

```
definef   convert(x(t1)) t2;
return x;
end convert;
```

In spite of this, the scheme just proposed does make non-standard uses of data objects stand out, and thus provides a useful measure of programming discipline. It is also worth observing that this scheme can readily be integrated with a static variant of the user-defined object type mechanisms described in Newsletter 76, allowing the definition of mechanisms which provide not only programming discipline, but notational flexibility and convenience.