SETL Newsletter # 171A

'Basing Semantics' Revisited

J. Schwartz
R. Dewar
December 9, 1976

1. Introduction

This newsletter will outline what seems to be an improved approach to the SETL basing concept. A fundamental idea of the proposed semantic revision is not to allow bases to be used directly as values; that is, all bases will be taken to be 'virtual' in the sense of our earlier terminology. (Of course, this makes the explicit keyword 'virtual'unnecessary). In our new scheme, bases are still declared, e.g. in the form

b: base(∈b');

but a variable declared to be a base cannot appear in any SETL expression and cannot be an assignment target. (Our former use of a base b also as a set s is regarded as confounding two things which are better kept separate; e.g. our former diction s: base set(∈b') is now handled by writing two declarations b: base(∈b') and s: subset(b).)

Values of bases will be represented in much the same way as now contemplated, but our new scheme abolishes the notion of 'ghost element' and with it the firm need for an is_deleted bit. Each declared base corresponds to exactly one resolved name in a total SETL program. For variables declared base (which occur only in a repr setting) the qualifier stacked cannot be stated. (However, bases will sometimes be stacked; but the rules which determine when this happens are implicit and explained below.)

If a variable name b is declared base, the *modes* ∈b and subset(b) become available for use in other declarations.

In addition to variable b declared base, our new system will have 'base values', which can be the values of such variables. Assignment of a new base value v to base variable b will not be called for explicitly, but will take place whenever such an assignment makes conversion of associated non-base variables unnecessary.

Suppose that the variable v has a repr declaration, that its declared repr is r, and that this repr involves at least one base variable b and is not simply the repr $\in$ b. Then the extype field of each SETL object appearing as the value of v will describe its repr fully except for the specific bases appearing in this repr; and a list of these base values, in their left-to-right order of appearance within r, will be held in a part of the run-time representation of v called its base array. For example, if v has the repr
v: smap ($\in$b) set (<$\in$b, subset(b'),smap($\in$b,$\in$b)>,
then the extype field of its value will convey as much information as shown in
smap(*)set(<$\in$ *, subset(*), smap(*,*)>), while the base array associated with this value will contain the values of the base variables b_1 b_1 b_1' b in order.

A variable v declared to have a given repr will conform precisely to this repr. When the value of v is assigned to a general variable g, then its extype and base array will be carried along as part of its value and will show the actual structure of g in all detail. Then, if g is subsequently assigned to a declared variable v', its structure can be checked quickly, and full examination of the details of g can be avoided.

In making an incremental modification such as f(x)= y or s with y of compound objects f, s with declared repr's, we convert x and y (or y alone) to stand in suitable relation to the full declared repr of f (or s). If f lacks a declaration, it may be best simply to convert f to type general if it is incrementally modified. (Though as a matter of fact in some cases, e.g., if f is a based map or smap, we can check to see whether the actual repr of y matches the range extype of f, and if this is so can refrain from modifying the extype of f.)

Note that the proposed system of extypes makes the following approach to assignments d = g possible, where we assume that d has a declared repr (other than $\in$b) and g is general: check the extype field of the value of g for equality with the extype field required for d, and then check the bases in the base array of the value of g for identity with the list of bases specified for d. The routine which performs this latter check can be passed the base array if g and an array of symbol table pointers (defining the bases of d) as its arguments.

2. Base Assignments.

If the declared repr of a variable v contains the base name b, then v is said to be based on b. Suppose that immediately prior to a given point in a program, all the objects based on a set of base variables $b_1,\ldots,b_n$ are dead, and that the set of variables is closed in that it contains b if it contains a b' with the repr b': base($\in$b). Then we say that the bases $b_1,\ldots,b_n$ are substitutable at that program point; we are free to substitute new values for the current values of $b_1,\ldots,b_n$ without spoiling any basing relationship on which we might be relying. In some cases, we will find it advantageous to generate new null base values for $b_1,\ldots,b_n$; in other cases, it can be advantageous to assign existing base values to the base variables $b_1,\ldots,b_n$.

More precisely, consider a simple or multiple assignment

(*) $$<f_1,\ldots,f_k> = g,$$

and suppose that the substitutable base variables at point of occurence of (*) are $b_1,\ldots,b_n$. Let $f_1,\ldots,f_k$ have the declared repr's $r_1,\ldots,r_k$. Let $b_1,\ldots,b_n$ be the largest substitutable subset of $b_1,\ldots,b_n$ with the property that all b_j, $j<m$ appear in at least one r_k. Given a base name b in this list, let f_j be one of the first variables in (*) such that be appears in r_j. (Here we say 'one of the first' rather than 'the first' since if r_j is, e.g., 'smap($\in$b)*mode*', while r_n is '$\in$b', we will wish to use f_j rather than f_k to

determine the new value of b, even if k < j. Let g(j) be the j-th component of the right hand side of (*). If the extype of g(j) matches that implied by r_j, and if the actual base value β occurs in g(j) where b occurs in b_j, then (*) is said to imply the base assignment b = β. If this extype match fails, then (*) is said to imply the base assignment b = nℓ, that is, to imply the creation of a new base value.

We execute (*) by first performing all the base assignments which (*) implies, and then by going on to perform the individual assignments f_j = g(j), during which all necessary conversions are made. (Note again that the base assignments (versus creations of new bases) to be performed are determined dynamically, by examination of the right-hand side of (*), except of course when by global analysis it becomes possible to make this same determination statically.) It is also important to note that the number of conversions which is necessary will be diminished by the base assignments which we perform in connection with (*); indeed, in some cases, assignment of new bases will make all conversion unnecessary. It is precisely for this reason that we choose to associate one or more base assignments with (*).

Note that the compiler may be able to detect sequences of simple assignments which can be treated in the same way as a multiple assignment (*), even though the syntax of a sequence of simple assignments is less explicitly helpful than the syntax of a single multiple assignment. The preceeding rule can then be applied to such sequences of assignments.

Note also that the above rule applies even if the $f_1,\ldots,f_n$ appearing in (*) are fairly general sinister expressions, provided that we agree that the sinister expression h(x) is to be taken as having the declared repr $mode_2$ if h has the declared repr map($mode_1$)$mode_2$, etc.

3. Parameter Passing, Recursion and Base Stacking.

Transmission of arguments on procedure entry can be regarded as a multiple assignment $\langle p_1,\ldots,p_n\rangle = \langle a_1,\ldots,a_n\rangle$. Like any other assignment, this will imply certain associated base assignments, and certain conversions. If necessary, these conversions will be performed in the called procedure. Similarly, in the case of returned parameters the return operation can be regarded as a multiple assignment $\langle a_1,\ldots,a_n\rangle = \langle p_1,\ldots,p_n\rangle$ which again implies certain base assignments and certain conversions. If conversions are necessary after return, they are performed in the calling routine.

Since conversion on call and return can lead to particularly elusive forms of time-wasting, statements which might generate such conversions should always be noted in emphatic compiler warning messages. Of course, we will also want the compiler to note all conversions, even those not associated with procedure calls.) If a particular subprocedure is never used as the value of a procedure variable, then it will be possible to locate all its points of call, and it may be possible to precalculate the repr's of all the variables passed to it and thus to determine all the conversions which take place on call (and perhaps, with some additional difficulty, on return as well.) If available, information of this sort can be used to suppress some call-conversion messages and to increase the precision and severity of others. Warning messages should also be given when the value of a procedure variable is invoked. It is also quite important to provide good histogram of a program's run-time behavior.

If a simple or multiple assignment (*) has a right-hand side g which is either nult or some other constant which is not associated with any particular base, then the general rule stated in the preceeding subsection implies that every substitutable base variable b associated with the assignment (*) is to be given the value n2, i.e., that a new base is to be created and made the value of the variable v. (However, values v

actually based on the former value bv of b do not cause trouble, since they retain pointers to bv, which among other things implies that bv is preserved from ruin by the garbage collector.) By giving b the value nl in such cases, we shorten the length of the vectors needed for storage of objects remotely based on b, and also cut down on the time needed for iteration over subset's of b and map's locally or remotely based on b. (Of course, this way of proceeding can generate indefinitely many base values bv.)

A related case is that in which all the variables $f_1,\ldots,f_n$ of (*) are stacked by a call to a given procedure (and unstacked on return), in which case the assignment (*) simply represents the operation of re-initalising all the stacked variables $f_1,\ldots,f_n$ to Ω. Clearly in this case all the substitutable base variables associated with (*) can be assigned new nl values; but the old value of each of these base variables should be stacked when this happens, and then unstacked on return, so that the current values of $f_1,\ldots,f_n$ always stand in proper relationship to the current values of these base variables.

4. Repr determination for tempories, implied conversions.

The repr of a temporary variable will where possible be determined from the use to which the temporary is put, but where this is not possible from the expression defining the temporary. For example, consider the assignment

(**) s = u + v with x - y.

in which the set s has the declared repr r. Then the temporaries t_1 = u + v and t_2 = u + v with x will both inherit the repr r. Before performing the operation (**) u and v will be converted to the repr r, while x will be converted to the repr r' naturally associated with elements of sets having the repr r. The element y requires no conversion. If there exist one or more base variables b which are substitutable (in the sense defined in section 2) at the point (**), then u will be examined to determine what the new value of b is to be. If the actual repr of u is at least as specific as the declared repr of s, then

new value of b will be a base obtained from an appropriate field of u; otherwise the new value of b will be a new, initially null, base.

As an example of the code defined by the preceeding rules, consider the case in which we have declared

```
repr  s: subset(b);
```

and compile the code fragment

```
s = {x + 1, x ∈ u |C(x)};
```

this expands into the sequence

```
ℓ1:     t = nℓ;
ℓ2:     (∀x∈u)
ℓ3:         if not C(x) then continue;;
ℓ4:         t = t with (x + 1);
ℓ5:     end ∀;
ℓ6:     s = t;
```

Since the (necessarily unique) programmer-defined variable to which the compiler temporary t is assigned is s, t inherits the repr t:subset(b). Thus the quantity x + 1 formed in line ℓ4 will be converted to the element representation ∈b before the with operation is performed. This leads to an efficient treatment of the original code sequence; in particular, unnecessary conversion operations are avoided.

Hopefully, it will not be hard to define efficient basings by exploiting the rules stated above. As an illustration of some of the effects that can be achieved, consider the following repr declarations and associated code:

```
repr    b:base, c:base,
        s:set,
        x:∈b, f: smap(∈b) ∈b, g:subset(b),
        xx:∈c, ff: smap (∈c) ∈c, gg:subset(c);
```

```
ℓ1:     x = ...; f = {...}; g = {...};
ℓ2:     s with <x,f,g>;
        ...
ℓ3:     <xx, ff, gg> = ∋ s;
```

In this example, conversions to the declared basings of x,f, and g take place in line ℓ1, at which point a new base value may be generated; no conversion is implied by ℓ2, since s has been described as a set of general objects; and no conversion is implied by ℓ3, since it will be discovered dynamically that the object ∋s is a tuple, and that after a base assignment the components of this tuple can be assigned to xx, ff, gg respectively without any conversion.

5. A Remark Concerning Local Objects.

The possible kinds of local objects are local subset, local map, and local smap. Local objects can be used somewhat more efficiently than the corresponding remote object types, but a substantial part of this benefit may be dissipated if it is not known statically whether the object is local or remote. Since the likely fate of an object whose shared bit is set is to be garbage collected (this is illustrated by the sequence f = g; f(x) = y; g(u) = v;) we will not want to allow sharing of local objects (even though remote copies of these objects could in principle be created when copies were necessary). On the other hand, we would like to be able to pass local objects to procedures as parameters. Finally, we wish to avoid situations in which a local object would have to be converted to remote form, but where the corresponding remote object could be used without copying.

To meet this complex of requirements, the following approach is suggested.

(a) A variable f will be called potentially local if it has a declared or inferred repr of the form subset, map, or smap, and if it never appears in a context in which its share-bit would be set, or in which its value would be incorporated into a composite object without copying.

In particular, this means that at each simple assignment g = f either the value of f is dead, or the value of f is not dead and will certainly be modified, so that copying of f would be required even if f were represented remotely. Similarly, at an indexed assignment g(x) = f or with operation s with f we require that f be live and certain to undergo modification, so that f would have to be copied even if its representation were remote.

(b) If the variable f might have subset repr and appears as an argument to one of the operations f + g, f -g, f * g we do not consider it potentially local, since for these operations the use of bit-strings in their remote form may have decisive advantages.

(c) If the variable f appears in a simple assignment statement g = f, and g is not part of the collection of all potentially local variables, then f should be dropped from this collectio unless f is live and certain to be modified.

(d) If f is accessible outside a single procedure p and is transmitted as a parameter, or if f is accessible only within p, is not stacked by p, and is transmitted as a parameter, then f should be excluded from the collection of potentially local variables. Moreover, if f is itself a parameter to which a value other than the value of a potentially local variable is passed, or if f is passed as an argument to become the value of a parameter which is not potentially local then f should also be excluded from this collection.

The variables which survive these various exclusions can be designated as definitely local, and the values of these variables can be represented by values of type local subset, local map, or local smap as appropriate.

6. A Remark on Subprocedure and Function repr's.

By providing suitable repr declaration for subroutines and function, we could in principle reduce the number of dynamic checks required in the treatment of procedure variables. On the other hand, the necessity for conversion or procedure call return is more often determined by the consistency with which arguments are based than on the precise bases used. Since we have no way of expressing relationships of this type without substantial extension of our current basing syntax, and since procedure variables to not seem to be of very common use in SETL programs, we shall not use the extype of subr or function to represent anything else than the number of arguments it expects and the pattern of arguments which it modifies.