SETL Newsletter # 151

J. T. Schwartz
June 20, 1975

Additional Considerations Concerning Semi-Automatic Data Structure Choice

This newsletter returns to the theme of NL 39 (*More Detailed Suggestions Concerning 'Data Strategy' Elaborations for SETL*), namely to the idea of a declarative, programmer-assisted approach to the problem of data structure choice. Since such an approach is an alternative to and perhaps also a preparation for fully automatic structure choice, it deserves investigation. This newsletter will simplify and flesh out the suggestions made in NL 39.

The foundamental technical idea with which we will work is that of *basing*. A SETL quantity x is said to be represented in *based form*, or to be *based*, if it is not kept in standard form, but instead is kept in a special form which relates it to some other set s called the *base* for the representation of x. A very wide collection of useful non-standard representations could be devised; however, in the present newsletter we shall concentrate on a modest but typical and probably adequate system of representations.

For each of the basings which we admit, a symbolic notation will be introduced. The family of notations which thereby arises (or perhaps some equivalent family of notations chosen for greater syntactic convenience) constitutes a language of data structures which can be used declaratively as a data structure elaboration language. An example of such a notation is '$\subseteq$ s', which we will use to describe sets which are subsets of s, represented by bits stored locally with the individual elements of s. To indicate that s_1 has this representation we will write $s_1 :: \subseteq$ s, which may be read 's_1 is represented as a subset based on s'.

A full list of the 'based' representations which will constitute our vocabulary of data structures, together with the symbolic notations which we shall use to designate them, is as follows:

s_1:: $\subseteq$ s - the subset s_1 of s is represented by a collection of bits stored locally with the individual elements of s.

s_1::R $\subseteq$ s - The subset s_1 of s is represented by a bitvector. In this case, a generated serial number referencing a particular bit position is stored locally with each individual element of s.

s_1::L $\subseteq$ s - the subset s_1 of s is represented using both a collection of bits stored locally with the individual elements of s and a list of pointers to elements of s. This list serves to expedite iterations over s_1.

s_1::LR $\subseteq$ s - the subset s_1 of s is represented using both a bitvector and a list of pointers to elements of s. In this case, as in the s_1::R$\subseteq$s case, serial numbers are stored locally with the elements of s.

x:: $\in$ s - The object x is represented by a pointer to an element of s.

x:: $<\beta_1,\ldots,\beta_n>$ - the object x is represented as an n-tuple, whose components have the basings $\beta_1,\ldots,\beta_n$ respectively. E.g., one might write x::<$\subseteq$ s, $\in$ s, L$\subseteq$s'>.

x:: {β} - the object x is represented by a hash table with locally stored overflow, whose entries point to objects having the basing β. E.g., one might write x:: {$\subseteq$ s}.

f:: M(s, β) - the set f is a single-valued map defined on (a subset of) s, with map values (or pointers thereto) stored locally with the individual elements of s. The map values are represented using the basing β.

f:: RM(s,β) - f is a single-valued map defined on (a subset of) s, with map values (or pointers thereto) stored (remotely) in an array (i.e., tuple). In this case, as in the s_1::R $\subseteq$ s case, serial numbers are stored locally with the elements of s. The map values are represented using the basing β.

f:: MM(s,β) - f is a multivalued map defined on (a subset of) s the values constituting f{x} being stored as a list referenced by an initial list element pointer stored locally with each element x of s. The map values are represented using the basing β.

f:: RMM(s,β) - f is a multivalued map defined on (a subset of) s, the values constituting f{x} being stored as a list referenced by an initial list pointer stored in an array (i.e., tuple). In this case, as in the s_1::R $\subseteq$ s case, serial numbers are stored locally with the elements of s. The map values are represented using the basing β.

x::Ω - the object x is represented in its standard SETL form.

x:: INT(k) - x is an integer of known size.

x:: STR(k) - x is a string of known maximum size

x:: BIT(k) - x is a bitstring of known maximum size.

The basings listed above can be compounded in obvious ways, and combined as long as certain restrictions are observed. With one exception, we will only allow a set to be used as a basis if it is represented in a manner which implies that its elements will be grouped together in some kind of list or array. For this reason, the basing s_1:: $\subseteq$ s is illegal if s has the basing s::$\subseteq s_2$ or s:: R $\subseteq$ s_2, but legal if s:: L $\subseteq$ s_2. In this latter case, the bits which flag membership/nonmembership in s_1 will be stored with the elements of the 'list of all members of s' that is maintained. Similarly, the basing f:: M(s,β) is illegal if s has the basing s:: $\subseteq$ s_2 or s:: R $\subseteq$ s_2, but legal if s:: L $\subseteq$ s_2, in which case the values of the map f will be stored with the elements of the list of all members of s. The cases in which we allow basings like s_1::$\subseteq$s and f::M(s,β) are: s::L $\subseteq$ s_2, s::LR$\subseteq s_2$, s::{β}, s::Ω, s::M(s_2,β), s::RM(s_2,β), s::MM(s_2,β), s::RMM(s_2,β). That we should allow the basing s_1::$\subseteq$s even when s has the basing s::M(s_2,β) is somewhat exceptional, since the elements of s have no representation of their own, but are merely represented by fields attached to the elements of s_2.

However, this type of basing is useful for storing submaps of maps having a based representation, and is therefore allowed.

Using the fundamental basings which appear in the preceeding list, and compounding them in ways conforming to the rules just stated, we obtain a large family of possible basings. Given a SETL program and a variable x in it, we allow any of these basings to be declared for x. Then, given a typical SETL binary operation i_1 op i_2, and assuming that basings have been declared for its inputs i_1 and i_2, there will exist some standard way of performing the operation, and the operation's result will have some standard basing. As an example of this general remark, consider the operation s_1 with x. Suppose first that s_1 has the representation $s_1 :: \subseteq s$. Then to calculate s_1 with x we first locate x as a member of s; if this location operation is successful, we set the bit attached to the member of s that has been located, and if this changes the bit, we adjust the count and hash maintained with s_1 appropriately. On the other hand, if x cannot be located as a member of s, we consider the basing declared for s_1 to be in error, since in this case the calculation of s_1 with x can make necessary some very extensive reconstruction of s_1, which we do not wish to allow. All of this is shown in the following code, which realises the operation s_1 with x in the case $s_1 :: \subseteq$ s.

```
(4)  if locate(x,s) is ℓx ne Ω then
          if s1_bit  (ℓx) eq 0 then
               s1_bit (ℓx) = 1;  /* set flag bit */
               count(s1) = count(s1) + 1;
                    /* adjust count field of s1 if necessary */
               hash(s1) = hashplus(hash(s1), hash(x));
                      /* adjust hash field of s1 if necessary */
          end if s1 _ bit;
     else
          error ('basing violation in s with x operation, x not
                                                  present in base')
     end if locate;
```

In this code, *locate(x,s)* is a subprocedure that returns a pointer ℓx to the item of s which is equal to x if such an item exists, but Ω otherwise. Also, *count(s_1)* is a field of s_1 in which the current number of elements of s_1 is maintained (if it is necessary to maintain such a count); and *hash(s_1)* is a field of s_1 in which standard hash quantity calculated from its elements is kept, if it is necessary to maintain such a hash (the function *hashplus* adjusts the calculated hash of s_1 in a standard way when the element x is added to s_1). Finally, the 1-bit field s_1_bit(ℓx) contains the bit which flags membership/nonmembership of the item ℓx in the set s_1.

Depending on the way in which x and s are represented, and on the global context in which x and s appear, the code (*) will represent a large or smaller amount of calculation; moreover in certain cases parts of this code can be elided. The most drastic variations in the time needed to execute (*) will come from variations in the time needed to calculate the function locate(x,s). If x has the basing x::∈s we are at one extreme, since locate(x,s) simply reduces to x. If x::Ω and s has a basing like s::L⊆s_1 then we are at another extreme, since locate(x,s) is calculated using what may be a long list search and a sequence of identity tests (which check x for identity with successive elements of this list). In addition to the execution-time variability involved in calculating *locate*, elision of parts of the code (*) will speed (*) up in certain cases. If the set s_1 is never tested for equality with any other set than nℓ and never becomes a member of a collection of sets or part of the domain of a function, then it is not necessary to maintain hash(s_1), and the line in (*) which does so can be elided. If s_1 is used only for membership testing, but is not itself tested for equality with the null set, then no count need be maintained for it, so that the line of (*) which does so can be removed, and (*) elided to

```
if locate(x,s) is ℓx ne Ω then
     s1_bit(ℓx) = 1;
else
     error ('basing violation...etc');
end if;
```

Finally, if global inclusion-membership analysis shows that $x \in s$ must hold, then the test appearing in this last code is unnecessary, and (*) reduces to

```
s1_bit(locate(x,s)) = 1;
```

or even, if x has the basing $x::\in s$, to

```
s1_bit(x) = 1.
```

In the preceeding example, we have assumed that the operation s_1 with x can be performed destructively. If this is not the case, s_1 will have to be copied. If a copy operation must be performed, it may be advantageous to change the representation of s_1 while copying it. This possibility will be explored in more detail below.

In cases like those reviewed above, the manner in which the result of a binary operation i_1 op i_2 is represented will be determined by the representations of i_1 and i_2. For example, unless a conversion is forced, the quantity s with x will have the same representation as s. Conversions will be forced if

a. The left-hand side of a simple assignment

```
x = expn
```

is specified to have a representation different from that which the right hand side of the assignment would ordinarily have; or

b. The map f appearing as the left-hand side of an indexed assignment

```
f(x1,...,xn) = expn
```

is specified to have a representation which makes it necessary to convert either the tuple $\langle x_1,\ldots,x_n, expn\rangle$ or the map-value *expn* to some form different from which it would ordinarily have; or

c. a form different from that which it would ordinarily have is directly specified for the value of an expression (this can be done using syntactic conventions that will be described below.)

Except in these cases, assignments will be performed without conversion of their right hand sides. This rule can be used to allow omission of declarations that would otherwise be required for specification of the form in which the values of a variable x will be represented. If all assignments to x have right hand sides which possess the same representation, then x will automatically be assumed to have this representation. This rule applies also to variables x appearing in set-theoretic iterators $\forall x \in s \ldots$: e.g., if s is declared to have the basing $s::L\subseteq s_1$, then x has the basing $x::\in s_1$, while if $s::S(\beta)$, then x has the basing β. The null set can be represented in any basing and is therefore neutral with regard to basings. A variable whose basing would otherwise be ambiguous because it appears in several assignments which imply different basings must have its basing specifically declared; if this declaration is omitted, a diagnostic should be issued. A basing declaration will force conversion operations to be inserted at appropriate points in a compiled code.

Since expressions in SETL can be quite rich, it will sometimes be desirable to declare the way in which a particular subexpression of a larger expression is represented. In effect, this declares a representation for the compiler-generated temporary variable which corresponds to a subexpression, and may insert some conversion operation into the code which is generated.

Once a representation has been declared for each (programmer and compiler-generated) variable appearing in a binary or n-ary operation, a crude but useful formula for the amount of time required to perform the operation can be generated.

In writing such formulae, we will use the symbol E for one 'elementary operation time', essentially the time required to perform an indexing or test operation; the symbol #S will be used to denote the number of elements in a set S or the number of components of a tuple S. The symbols #∋S will be used to denote the typical number of elements in the members of a set S and the typical number of elements in the components of a tuple S, etc.

We can illustrate the use of these notations by writing a formula for the time required to calculate s_1 with x in cases for which the formula (*) applies. This is (roughly)

3.5 E + TLOC;

where TLOC designates the time required to calculate the auxiliary function location(x,s). The quantity TLOC can itself be calculated given the representations of x and s. Suppose for example that s has the basing s::L⊆s_2. Then locate(x,s) is calculated by searching in the list which represents s For a pointer to an element (of s_2) equal to x. We can therefore offer the estimate

TLOC = ½. # s (2 E + TEQ),

where TEQ is the time required to compare x and an element of s_2 for quality. In a simple case (such as x::∈s_2) TEQ will have a simple estimate (e.g.,E); in more complex cases a formula for TEQ will have to be developed recursively, using the known representations of x and of the members of s_2.

At any rate, formulae estimating the execution time of each primitive operation appearing in a SETL program P will always be derivable once a representation has been specified for each variable of P. These formulae can be printed as program annotations which can help a programmer find an effective overall set of representations. It may also be possible to base a process of automatic data choice on these formulae. A plausible technique is as follows. First, collect frequency information on all the explicit loops and branches of a program.

This will involve assigning a relative execution probability to each of the parts of every if-then-else statement, an expected number of iterations to each while statement, and a relative probability to each boolean condition clause C(x) appearing in an iterator of the type $(\forall x \in s \mid C(x))$. Using this information, an execution frequency can be assigned to each primitive operation of the program P. Information on the sizes of each of the sets s and vectors v appearing in P should also be collected. With all of this information in hand, data representations can be assigned to the objects of P in all possible ways, the resulting execution time calculated, and an optimal representation scheme chosen. For this purpose, a branch-and-bound method which examines the most frequently performed operations of P before processing the other operations of P can be used; such an approach can drastically reduce the number of cases which need to be considered. Branch-and-bound algorithms always work most efficiently if the estimated solution with which they start is not too far from optimal. For this reason, it may be worth providing a heuristic algorithm which attempts to develop an advantageous initial set of representations. Such an algorithm might work by trying to set up a 'highly advantageous' system of representations, i.e., a system of representations which allow many of the high frequency operations of P to be performed in particularly efficient ways. Note for example that s with x can be calculated with particular efficiency if $s::\subseteq s_1$ and $x::\in s_1$, that f(x) can be calculated efficiently if $f::M(s_1)$ and $x::\in s_1$, that s+s' can be calculated efficiently if $s::R\subseteq s_1$ and $s'::R\subseteq s_1$, etc.

When conversions are performed, either in response to some relatively explicit request or in preparing to perform some operation one of whose arguments must be converted, then two or more representations of the same SETL value will come into existence. This value equivalence, if tracked globally, may make it possible to avoid subsequent conversion operations. A technique for global tracking of equality relationships is outlined in section 5 below.

2. Disallowed operations, copying conversions.

Certain basings will be disallowed for variables which enter in particular ways into specific SETL primitives. Sets with the basing $s_1 ::\subseteq s$, $s_1 ::L\subseteq s_1$, $s::M(s_1)$, $s::MM(s_1)$ have what may be described as a 'distributed' rather than a 'grouped' representation, and for this reason are not allowed to become elements of other sets or components of vectors.

As noted above, we do not allow a set which has the basing $s ::\subseteq s_2$ or $s ::R\subseteq s_1$ to serve as a basis for any other object x, since x can just as well be based on s_1 as on s. Moreover, if s has the basing $s ::\in s_1$, we do not allow s to serve as a basis, since this would introduce substantial complications into some of the situations with which we will have to deal.

Operations which cannot be performed destructively will force copies of one or more of their arguments to be made. In some cases, as for example when an argument x has the basing $x::\in s$, the argument's representation will modified while it is being copied. In this particular case, the way in which the copy xc of x is based will depend on the representation of s. The rules which determine the representation of xc from that of s are as follows:

(a) As noted in the preceeding section, neither $s::\subseteq s_2$ nor $s::R\subseteq s_2$ can occur.

(b) If $s::L\subseteq s_2$ or $s::LR\subseteq s_2$, x will be converted to an element x' with basing $x::\in s_2$, and then xc formed as if x had this basing originally.

(c) If $s::M(s_2,\beta)$ then an object x with $x::\in s$ basing is actually represented by a pointer to an element x' of s_2.

We copy x by forming a copy c' of x' and a copy c" of the element x" referenced by the s_field of x' , and then by forming the pair <c',c">. Much the same procedure is used if s::RM(s_2,β), except that in this case the s_field of x' is an index which locates s(x') rather than a pointer to it. The copy of x has the basing <β',β>, where β' is the basing of the copy c' of x'.

(d) If s::MM(s_2,β) or s::RMM(s_2,β), then x is a pair p',p" of pointers, where p' points to an element x' of s_2, and p" points to a hash-table element, which in turn points to an element x" having the basing β. We copy x by forming a copy c' of x' and a copy c" of x" , and then by forming the pair <c',c">. This copy of x has the basing <β',β>, where β' is the basing of the copy c' of x'.

(e) If s::{β}, then x is a pointer to an item y in the hash table which represents s. To copy x we copy y.

(f) If s::Ω, and the value of x is not a tuple, then we proceed as in case (e). But if the value of x is an n-tuple, then x will be represented as a list of n-1 pointers y_j , where $y_1 \cdots y_{n-2}$ will point to hash table items $z_1,\ldots,z_{n-2}$ having the basing Ω, while y_{n-1} will point to a pair whose components z_{n-1},z_n have the basing Ω. We copy x by copying $z_1,\ldots,z_n$ and forming a tuple of the copies. The resulting object has the basing Ω.

We shall now give a full account of the automatic conversions which apply when an object x with a specified basing is copied. It should be noted that time formulae like those presented in section 3 below can be developed for each of these copying operations; however, in the present section we shall,for the sake of brevity, omit these formulae. Note also that the copy operations which we describe copy only the topmost level of a compound data item, not the subobjects of these compound objects. In cases where a full copy is necessary, the procedures described below can be extended recursively from objects to their subobjects.

The basings which must be considered are as follows:

(cop.1) x::$\underline{\subset}$s. To copy x, we iterate over s, skipping the elements which do not belong to x, and building up a hash table of the elements which do. The copy x' of x that is produced has the basing x' ::{$\in$s}.

(cop.2) x::R$\underline{\subset}$s. Here x is represented by a bitvector which has only to be copied. The copy x' of x has the basing x' ::R$\underline{\subset}$s.

(cop.3) x::L$\underline{\subset}$s. This case is almost the same as case (cop.1), except that using the list which forms part of the representation of x we can iterate directly over x rather than over the larger set s. The copy x' of x that is produced has the basing x' ::{$\in$s}.

(cop.4) x::LR$\underline{\subset}$s. Here x is represented by a bitvector and a list which have only to be copied. The copy x' of x has the basing x'::LR$\underline{\subset}$s.

(cop.5) x::$\in$s. This case, which is particularly complex, has been discussed at the beginning of the present section.

(cop.6) x::<β_1,...,β_n>. To copy x,form a new vector x' of equal length with the same components; this will also have basing x'::<β_1,...,β_n>.

(cop.7) x::{β}. To copy x, copy the hashtable which represents it. The copy x' which is formed still has the basing x'::{β}.

(cop.8) x::M(s,β). To copy x we iterate over s, skipping the elements y for which x(y) eq Ω, and building up a hashtable of all the pairs <y, x(y)> for which x(y) ne Ω. The copy x' of x that is produced has the basing x'::{<∈s,β>}.

(cop.9) x::RM(s,β). Here x is represented by a vector of values which has only to be copied. The copy x' of x has the basing x'::RM(s,β).

(cop.10) x::MM(s,β) To copy x, we iterate over s, skipping the elements y for which x{y} eq nl. For each y such that x{y} ne nl, we iterate over all the elements in the subsidiary hashtable which represents x{y}. During this process we build up a hashtable of all the pairs <y,z> for which z∈x{y}. The copy x' of x that is produced has the basing x'::{<∈s,{β}>}.

(cop.11) x::RMM(s,β). To copy x we copy the vector which represents it and all the lists at which this vector points. The copy x' also has the basing x::RMM(s,β).

(cop.12) x::Ω. This uses the SRTL copy routine, which resembles the code used in case (cop.7), but which is substantially more complicated because of the special way in which tuples are handled as members of Ω-based set.

(cop.13) x::INT(k), x::STR(k), x::BIT(k). In these cases the copying operation is elementary, and the copy x' that is formed has the same basing as x.

Forced conversions.

Whenever a conversion is forced, e.g., by explicit assignment of a y having one basing to an x specified to have another, a conversion procedure which produces the representation of x from that of y will have to be executed. We shall now sketch conversion procedures which can be used in the various cases that arise.

Note that time formulae like those presented in section 3 below can be developed for these conversion procedures; however, we shall not give these formulae now.

The various basings that need to be considered are:

(conv.1) x::$\subseteq$s. To convert y, we iterate over its elements ey and for each such element calculate the function ex= locate(ey,s). (Cf. the discussion following (4) of section 1 above.) Then the x_bit of the element of s at which ex points is set.

(conv.2) x::R$\subseteq$s, The procedure used is similar to that applied in case (conv.1), except that we set a bitvector bit whose index is found in the s-element at which ex points.

(conv.3) x::L$\subseteq$s. We proceed as in case (conv.1), also building up a list ℓe of the elements ex=locate(ey,s) as we go along; this list becomes part of the representation of x.

(conv.4) x::LR$\subseteq$s. The procedure is similar to that applied in case (conv.3), except that the bitvector bits which we set are located by indices found in the elements ex of s, and not directly in the elements ex themselves.

(conv.5) x::$\in$s. Here we simply calculate x = locate(y,s).

(conv.6) x::$<\beta_1,\ldots,\beta_n>$. The possible basings for y are y::$<\beta'_1,\ldots,\beta'_n>$, y::$\Omega$, and y::$\in$s. In the first case, we simply convert the components of y individually to the basings β_j, and build up a tuple of the converted components. Much the same remark applies if y::Ω. If y::$\in$s, then the possible basings for s are s::L$\subseteq$s', s::LR$\subseteq$s', s::{β}, s::Ω, s::M(s,β), s::RM(s,β), s::MM(s,β), and s::RMM(s,β). We treat these cases separately as follows:

(conv.6.1) s::L$\subseteq$s', also s::LR$\subseteq$s' Here y points to a list-element, which in turn points to an element of s'. Thus a single indirect reference transforms y to an element y' with the basing y::$\in$s'; applying this transformation repeatedly if necessary, we will eventually reach an element $y^{(n)}$ with the basing $y^{(n)}$::$\in s^{(n)}$, where $s^{(n)}$ has some basing other than $s^{(n)}$::L$\subseteq s^{(n+1)}$.

(conv.6.2) s::{β}. Here y points to a hashtable item, which in turn points to an element with the basing β. Thus a single indirect reference transforms y to an element with the basing β.

(conv.6.3) s::Ω. Here y, if it value is logically an n-tuple, will be represented as a list of n-1 pointers y_j. The quantities $y_1 \ldots y_{n-2}$ point to hash table items $z_1 \ldots z_{n-2}$ having the basing Ω, while y_{n-1} points to a hash table item which is a pair each of whose components z_{n-1}, z_n have the basing Ω. By converting $z_1,\ldots,z_n$ to the basings $\beta_1,\ldots\beta_n$ we obtain the components x(1),...,x(n) of the desired n-tuple x.

(conv.6.4) s::M(s',β). In this case y, which logically is a pair, points to an element z of s', and the s_field of z points to an element z' with the basing z'::β. We convert z to the basing β_1, thereby obtaining the first component x(1) of the desired element x, and convert z' to the basing β_2, thereby obtaining x(2).

(conv.6.5) s::RM(s',β). This case is much like case (conv.6.4), except that y points to an element z whose s_field contains an index i which defines the component v(i) (of an s-representing vector v) which contains z'. We convert z to the basing β_1, thereby obtaining the first component x(1) of the desired element x, and convert z' to the basing β_2, thereby obtaining x(2).

(conv.6.6) s::MM(s',β). In this case y is a pair of pointers, the first one of which references an element y_1 of s', and the second of which references a hash table entry that contains or points to an element y_2 having the basing β. By converting y_1 and y_2 to the basings β_1 and β_2 respectively, we obtain the first and second components of the converted element x.

(conv.6.7) s::RMM(s',β). This case is identical with case (conv.6.6).

(conv.7) x::{β}. Here we iterate over the elements ey of y, converting each ey individually to the basing β, and incorporating the converted elements into a hashtable.

(conv.8) x::Ω. If y is a tuple, we convert its components individually to the basing Ω, and incorporate the converted components into a new tuple. If y is a set, we proceed essentially as in case (conv.7) (however, tuple insertion in an Ω-based set is handled in a special way.)

(conv.9)x::M(s,β). We iterate over the elements ey of y, converting each ey individually to the basing <∈s,β>. The first component of this pair p locates an element es of the set s; we then set the x_field of this element equal to the second component of the pair p.

(conv.10) x::RM(s,β). The procedure used is similar to that applied in case (conv.9), except that the x_field of the element es contains an index i; we then set the i-th component of an auxiliary vector to equal the second component of the pair p whose first component is es.

(conv.11) x::MM(s,β). We iterate over the elements ey of y, converting each ey individually to a pair p with the basing <∈s,β>. The first component of p locates an element es of the set s; the x_field of this element (is either nil or) points to an auxiliary hash-table in which all the elements of x{es} are to be recorded. We insert p into this hash table, and then continue our iteration.

(conv.12) x::RMM(s,β). The procedure used is similar to that applied in case (conv.9), except that the x_field of the element es contains an index i defining the component (of an auxiliary vector) which contains a pointer to an x{es}-representing hash table. We insert p into this hash table, and then continue our iteration.

(conv.13) x::INT(k), x::STR(k), x::BIT(k). Conversion will only be required if y has the basing y::∈s; in this case, y points to an element of s which in turn points to an element with the desired basing, so that conversion is effected simply by passing from an indirect to a direct reference.

A number of special cases in which conversion can be performed with particular efficiency are worth noting. Suppose that y is to be converted to have the basing of x. If y::LRCs and x::RCs, it is only necessary to copy the bitvector part of y's representation (at high speed using word-length operations) to obtain the representation of x. (If y is dead after x is formed, we have only to drop the list part of the existing representation of y.) A similar remark applies in case y::RM(s,β) and x::RM(s,β), and also if y::RM(s,β) and x::RM(s,β´).

Whenever an object y is converted to a form x having a different basing, the object x will, immediately after its formation, be a logical copy of y. If this logical copy will come into existence anyhow, it may be unnecessary to copy y, even if y is an operation argument which would otherwise have to be copied. A programmer should have no difficulty in signalling such a situation to the redundant-copy elimination mechanisms of the SETL compiler by inserting explicit conversion operations into SETL code with which he is working.

3. Code and execution-time formulae for various SETL primitives.

In the present section, we will discuss a few typical SETL primitive operations, and will describe the code which will be used to realise these primitives. Of course, the code used depends on the way in which the arguments of a particular operation are represented. The most crucial operations are x∈s and x eq y, which enter as subroutines into many other operations and which call each other recursively. But we shall begin our discussion with a simpler operation, namely

(a) s_1 with x.

(a.1) s_1 :: ⊆s . *Code* is as follows:

```
if locate(x,s) is ℓx ne Ω then
     if s1_bit(ℓx) = 1;  /* set flag bit */
     count(s1) = count(s1) + 1;
     /* adjust count field of s1 if necessary */
        hash(s1) = hashplus(hash(s1), hash(x));
     /* adjust hash field of s1 if necessary  */
end if s1_bit;
else
     error('basing violation in s with x operation, x not
                                          present in base');
end if locate;
```

Elisions have been listed in section 1.

Execution *time* formula (not-elided case) is

3.5E + TLOC,

where TLOC is the time needed to calculate the auxiliary function locate(x,s).

(a.2) s::R⊆s. *Code* and *elisions* similar to case (a.1),but s_1_bit(ℓx) is differently located. *Time* formula is

4 E + TLOC.

(a.3) s_1::L⊆s *Code* and *elisions* similar to case (a.1), but ℓx should be added to list that represents s_1 if s_1_bit (ℓx) eq 0. *Time* formula is

4.5E + TLOC.

(a.4) s_1::LR⊆s. *Code* and *elisions* similar to case (a.3), but s_1_bit (ℓx) is differently located.

Time formula is

5E + TLOC.

(a.5) s_1::∈s. This case will never arise, since it cannot be performed destructively, but will force a copy of s_1 to be formed, which will convert s_1 to have the representation used for elements of s; see the discussion of this point given in section 2.

(a.6) s_1::{β}. The code here resembles the code for the with operation found in the SETL run-time library (SRTL). The hash-table representing s_1 is entered using the hash of x. This locates a list, estimated to be 2 elements long, which is searched for an element equal to x. If such an element is found, the with operation is a no-op. Otherwise x is added to the hash table; this may make rehashing necessary. For *code*, see the SRTL listing.

Elisions: Elide test if x known to be different from Ω;
elide hash-value update if hash of s_1 not needed;
elide count update if count of s_1 not needed;
elide equality tests if x∈s_1 is known to be false.

Time formula(assuming no elisions): 7E + 2 * TEQ;
where TEQ is the time required to test x for equality with an element having representation β.

(a.7) s_1::Ω. Assuming no conversions are necessary, this uses the SRTL code. Elisions are possible if some amount of global analysis, type determination, etc., is done.

Elisions: As in case (a.6) above, plus:
elide type test if s_1 is known to be a set;
elide type test if x is not a tuple, or if s_1 is never used as a map.

Time formula: 9E + 2 * TEQ,
where TEQ is the time required to test for equality with an Ω_based element.

(a.8) s_1::M(s,β). *Code* is:

```
if locate (x(1),s) is ℓx ne Ω then
      if mapval(ℓx) eq Ω then
            mapval(ℓx) = x(2);
            count(s1) = count(s1) + 1;
                  /* adjust count field of s1 if necessary */
            hash(s1) = hashplus(hash(s), hash(x));
                  /* adjust hash field of s1 if necessary */
   else if mapval(ℓx) ne x(2) then
         error('multiple definition error in s with x operation'
   end if mapval;
else
   error('basing violation in s with x operation, x not
                                          present in base');
end if locate;
```

Elisions: Elide test if x(1) known to be different from Ω;
elide test if x(1) is known to be a member of s;
elide count update if not necessary;
elide hash update if not necessary.

Time formula (assuming no elisions): 4E + TLOC + TCOMP1 + TCOMP2;
TLOC is the time needed to calculate the auxiliary function locate(x(1), s); TCOMP1 is the time necessary to calculate a representation of x(1) from the representation of x, and TCOMP2 is the time necessary to calculate a representation of x(2).

(a.9) s_1::RM(s_1β). *Code* and *elisions* similar to case (a.8), but mapval(ℓx) is differently located. *Time* formula is

4.5E + TLOC + TCOMP1 + TCOMP2,

where TLOC, TCOMP1, and TCOMP2 are as in case (a.8).

(a.10) s_1::MM(s,β). *Code* is:

```
if locate (x(1),s) is ℓx ne Ω then
      if list_locate(ℓx,x(2)) eq Ω
            /* i.e., x(2) not ∈ s{x(1)}*/ then
            allocate (mapvalspace); /* allocate space for
                                       storage of new map value
            next (mapvalspace) = mapval(ℓx);
            item (mapvalspace) = x(2);
            mapval(ℓx) = mapvalspace;
      end list_locate;
else
      error ('basing violation in s with x operation,
                                       x not present in base').
end if;
```

Elisions: none

Time formula:

7E + TLOC + TLOC2 + TCOMP1 + TCOMP2, where

TCOMP1 and TCOMP2 are as in case(a.8), TLOC is the time needed to calculate the auxiliary function locate(x(1),s), and TLOC2 is the time needed to calculate the function list_locate(ℓx, x(2)) which searches the list whose head is referenced by mapval(ℓx) for an item equal to x(2). The quantity TLOC2 can be estimated as

(# s{x(1)}) * (2E + TEQ), where

TEQ is the time required to test x(2) for equality with one element of this list.

(a.11) s_1::RMM(s,β). *Code* and *elisions* similar to case (a.10), but mapval(ℓx),used in list_locate(ℓx,x(2))) is differently placed. *Time* formula is

7.5E + TLOC + TLOC2 + TCOMP1 + TCOMP2

where TLOC, TLOC2 TCOMP1 and TCOMP2 are as in case (a.10).

Next we consider the related operation

(b) $x \in s_1$, for which the following subcases arise:

(b.1) $s_1 :: \subseteq s$. *Code* is as follows:

```
if locate (x,s) is ℓx ne Ω  then
    return s1_bit (ℓx) ne 0
else
    return false;
end if;
```

Elisions: if $x \in s$ is known, the initial test can be elided.

Time formula (non-elided case) is

$$.5E + TLOC,$$

where TLOC is the time needed to calculate locate(x,s).

(b.2) $s_1 :: R \subseteq s$. *Code* and *elisions* similar to case (b.1), but s_1_bit(ℓx) is differently located. *Time* formula is

$$E + TLOC.$$

(b.3) $s_1 :: L \subseteq s$. *Code, elisions,* and *time* formula same as case (b.1).

(b.4) $s_1 :: L\ R \subseteq s$. *Code, elisions,* and *time* formula same as case (b.2).

(b.5) $s_1 :: \in s$. If s_1 is based in this way, the only possible basings for s are $L \subseteq s'$, $L\ R \subseteq s'$, $\{\beta\}$, and Ω. The procedures used to evaluate $x \in s_1$ in these separate cases, and their analysis, is as follows:

(b.5.1) $s :: L \subseteq s'$. Let *actual* (s_1) denote the element (of the list which represents s) to which s_1 points; this item has the basing $\in s'$. Then $x \in s_1$ is implemented as $x \in$ actual (s_1), requiring a *time* $E + T'$, where T' is the time required to perform the test $x \in sa$ for an item sa having the basing $\in s'$.

(b.5.2) $s :: LR \subseteq s'$. The procedure used in this case, and its analysis, is the same as that used in case (b.5.1).

(b.5.3) $s::\{\beta\}$. Let *actual*(s_1) denote the element (of the s-representing hash table) to which s_1 points; this item has the basing β. Then $x \in s_1$ is implemented as $x \in$ actual(s_1), requiring a time $E + T_\beta$, where T_β is the time required to perform the test $x \in sa$ for an item sa having the basing β.

(b.5.4) $s::\Omega$. This case is handled in essentially the same way as case (b.5.3), except that here β is Ω.

(b.6) $s_1::\{\beta\}$. Here we use essentially the code for the membership test found in the SETL run-time library (SRTL). The hash table representing s_1 is entered using the hash of x. This locates a list, approximately 2 elements long, which is searched for an element equal to x. If such an element is found, true is returned; otherwise false is returned.

Elision: Elide test if x known to be different from Ω.

Time formula (assuming no elision):

$$5E + TEQ,$$

where TEQ is the time required to test x for equality with an element having representation β.

(b.7) $s_1::\Omega$. Assuming no conversions are necessary, this uses the SRTL code.

Elisions: Elide test if x is known to be different from Ω;
elide type test if s_1 is known to be a set;
elide tuple test if x is known not to be a tuple, or if s_1 is never used as a map.

Time formula: $7E + TEQ$,

where TEQ is the time required to test for equality with an Ω-based element.

(b.8) $s_1::M(s,\beta)$. *Code* is:

```
if locate (x(1),s) is ℓx eq Ω  then return false
    else return mapval(ℓx) eq x(2);;
```

Elsisions: Elide test if x(1) is known to be different from Ω;
elide test if x(1) is known to be a member of s;

Time formula: E + TLOC + TCOMP1 + TCOMP2,
where TLOC is the time needed to calculate locate(x(1), s), and TCOMP1 (resp. TCOMP2) is the time needed to calculate a representation of x(1) (resp. x(2)) from the representation of x.

(b.9) s_1::RM(s,β). *Code* and *elisions* similar to case (b.8), but mapval(ℓx) is differently located. *Time* formula is

2E + TLOC + TCOMP1 + TCOMP2,

where TLOC, TCOMP1, and TCOMP2 are as in case (b.8).

(b.10) s_1::MM(s,β). *Code* is

if locate (x(1),s) is ℓx eq Ω then return false ;
else return list_locate(ℓx,x(2)) eq Ω ;;

Elisions: none

Time formula:

2 E + TLOC + TLOC2 + TCOMP1 + TCOMP2,

where TLOC, TCOMP1, and TCOMP2 are as in case (b.8), and where TLOC2 is the time needed to calculate the function list_locate(ℓx,x(2)). For a description of this function, see (a.10) above.

(b.11) s_1::RMM(s,β). *Code* and *elisions* similar to case (b.10), but mapval(ℓx) (used in list_locate(ℓx, x(2))) is differently placed. *Time* formula is

2.5 E + TLOC + TLOC2 + TCOMP1 + TCOMP2,

where TLOC, TLOC2, TCOMP1, and TCOMP2 are as in case (b.10).

The equality testing operation occurs frequently in the preceeding discussion, and we now proceed to consider the way in which it will be implemented in the various cases which can arise. The operation in question is:

(c) x eq y. We consider various subcases reflecting the different possible basings for y.

(c.1) y::⊆s. *Code* is as follows:

```
if type x ne set or (# x) ne # y or hash(x) ne hash(y) then
                                        return false else
    (∀ z ∈ s | y_bit(z) eq 1)
        if not z∈x then return false;;
    end ∀z;
    return true;
end if;
```

Elisions: If the type of x is known, a test can be omitted.

Time formula: the average time required for this operation depends very strongly on the probability PEQ that x and y are equal:

8 E + (6 E * # s + # y * TMEMB) * PEQ,

where TMEMB is the time required for the membership test z ∈ x.

Note that an estimate can be supplied for PEQ in case the equality test x eq y with whose running time we are concerned is being executed repeatedly as part of a search over some compound object, since in this case one can assume that, with probability roughly one-half, x will be to equal to one of the items y of s and different from all the others.

(c.2) Y::R ⊆ s. *Code* and *elisions* are similar to case (c.1), but y_bit(z) is differently located. *Time* formula is:

8 E + (7 E * # s + # y * TMEMB) * PEQ,

where TMEMB and PEQ are as in case (c.1).

Special case: If x and y both have the basing R ⊆ s, then detailed comparison of x and y in the case of suspected equality can be done using bitvector operations. In this case, the *time* formula becomes

8 E + (2 E/NBPW) * PEQ * # s,

where NBPW is the number of bits per word for the machine supporting a given implementation.

(c.3) y:: L ⊆ s. *Code* is as follows:

```
if type x ne set or (# x) ne # y or hash(x) ne hash(y) then
        return false;
else
```

```
    pointer = first (y);  /* the representation of y includes */
                          /*a pointer to the first element of its*/
                          /*                    associated list   */
    (while pointer ne 0)
        if element (pointer) not ∈ x then return false;;
    /* element(pointer) retrieves the element of y associated */
    /*                                  with a particular pointer */
        pointer = next (pointer);  /* advance in list */
    end while;
    return true;
end if;
```

Elisions: If the type of x is known, a test can be omitted.

Time formula:

$$8E + (\#s) * (TMEMB + 3E) * PEQ,$$

where as in case (c.2) PEQ is the probability that x and y are equal, and TMEMB is the time required for the membership test element(pointer) ∈ x.

(c.4) y::L R ⊆ s. *Code, elisions,* and *time* formula identical to case (c.3).

Special case: If x and y both have one of the basings R R ⊆ s or LR⊆ s, then detailed comparison in the case of suspected equality can be done using bitvector operations. See (c.2) for the time formula applicable in this case.

(c.5) y::∈s. Here the way in which we proceed depends on the basing of s. The possible basings are s::L⊆s', s::LR⊆s', s::{β}, s::Ω, s::M(s',β), s::RM(s',β), s::MM(s',β), and s::RMM(s',β). The treatments appropriate in these various subcases are as follows:

(c.5.1) s::L⊆s'. The quantity y is a pointer to an item *itm* in the list s, and *itm* points to an element *actual(y)* in s'. Then x eq y is calculated as x eq actual(y).

Elisions: none

Time formula: $2E + TEQ,$

where TEQ is the time require to calculate x eq $\bar{y}$ for an element $\bar{y}$ having the basing $\bar{y}$::$\in$s'.

(c.5.2) s::LR $\subseteq$ s'. *Code*, *elisions*, and *time* formula identical with case (c.5.1).

(c.5.3) s::{β}. The quantity y is a pointer to an item actual(y) in the hashtable representing s; x eq y is calculated as x eq actual(y).

Elisions: none

Time formula: E + TEQ,

where TEQ is the time required to calculate x eq $\bar{y}$ for an element $\bar{y}$ having the basing β.

(c.5.4) s:: Ω. The quantity y is a pointer to an item actual(y) having the basing Ω.

Elisions and *time* formula: as in case (c.5.3). (Note however that if actual(y) is a tuple the procedure will be more complex and the time formula different.)

(c.5.5) s::M(s,β). The quantity y is a pointer to an element $\bar{y}$ of s'; x eq y is calculated as

x(1) eq $\bar{y}$ and x(2) eq mapval($\bar{y}$).

Elisions: none

Time formula: E + TEQ1 + TCOMP1 + .5 (TEQ2 + TCOMP2), where TEQ1 (resp. TEQ2) is the time needed to test x(1) for equality with an element of s' (resp. to test x(2) for equality with an element having the basing β), and TCOMP1(resp. TCOMP2) is the time needed to calculate x(1) (resp. x(2)) from the representation of x.

(c.5.6) s::RM(s',β). *Code* and *elisions* are the same as in case (c.5.5), but mapval($\bar{y}$) is differently located. *Time* formula is

1.5 E + TEQ1 + TCOMP1 + .5(TEQ2 + TCOMP2).

(c.5.7) s::MM(s',β). The quantity y is a pair consisting of a pointer to an element y' of s', and a pointer to an element y" of the hash table representing s{y'}. The quantity x eq y is calculated as

x(1) eq y' and x(2) eq mapval(y").

Elisions and *Time* formula are as in case (c.5.5).

(c.5.8) s::RMM(s',β). *Code* and *elisions* are the same as in case (c.5.7); *time* formula is the same as in case (c.5.6).

Special case (of case (c.5.6)): If y::∈s and x has the same basing, then x eq y can be tested simply by comparing the pointers x and y for equality. For this important special case, the *time* formula is simply

2 E.

(c.6) y::{β}. *Code* is as follows:

```
if type x ne set or (#x) ne # y or hash(x) ne hash(y)  then
        return false;
else
    (1 ≤ ∀n ≤ table_length(y) | element(table(start(y) + n)) ne 0)
        if element (table(start(y) + n)) not ∈ x then
            return false;;
    end ∀n;
    return true;
end if;
```

Elisions: If the type of x is known, a test can be omitted.

Time formula:

8 E + (9 E + TMEMB) * (# y) * PEQ,

where TMEMB and PEQ are as in case (c.3).

(c.7) y::Ω. *Code* in this case is the equality test code presently in the SRTL run time library. The set-theoretic case of this code rather resembles that of case (c.6), except that a more complex iteration over y is necessary if y is a set which can contain tuples. *Elisions* are possible after type determination;type determination is also necessary if a precise *time* formula is to be stated. Some approximate time formulae for a few cases in which y is a set are as follows:

if y is known to contain no tuples;

8 E + (9 E + TMEMB) * (# y) * PEQ

if y can contain tuples, but need not contain any:

8 E + (12 E + TMEMB) * (# y) * PEQ

where TMEMB and PEQ are as in case (c.3).

(c.8) y::M(s,β). *Code* is:

```
if typex ne set or (# x) ne # y or hash(x) ne hash(y) then
        return false;
else
    (∀z∈s |. y_field(z) ne Ω)
        if not <z, y_field(z)> ∈ x then return false;;
    end ∀z;
    return true;
end if type;
```

Elisions: If the type of x is known, a test can be omitted. If x is known to be defined everywhere on s, a test can be omitted inside a loop.

Time formula (assuming no elisions)

8 E + (8 E * # s + # y * (TMEMB + E)) * PEQ,

where #y denotes the number of elements in y, TMEMB the time required for the membership test <z, y_field(z)> ∈ x, and PEQ the probability that x and y are actually equal.

(c.9) y::RM(s,β). *Code* and *elisions* are similar to case (c.8), but y_field(z) is differently located. *Time* formula

8 E + (9 E * #s + #y * (TMEMB + E)) * PEQ

(c.10) y::MM(s, β). *Code* is

```
if type x ne set or (# x) ne # y or hash(x) ne hash(y) then
            return false;
else
    (∀z∈s)
        pointer = y_first(z);  /* the representation of each z∈s */
    /* includes a pointer to the first entry of a hashed table */
    /*                                  of the elements of y{z}  */
    if pointer eq 0 then continue ∀z;;
    ( 1 ≤ ∀n ≤ numberfield(table(pointer)) |
                          elementfield(table(pointer + n )) ne 0)
    /* numberfield gives the size of the hashed table referenced */
    /* by pointer, elementfield gives the element stored in     */
    /*                          a particular table position     */
        if <z, elementfield (table (pointer + n))>
            not ∈ x then return false;;
```

```
        end ∀n;
    end ∀z;
    return true;
end if;
```

Elisions: If the type of x is known, a test can be elided If x is known to be defined everywhere on s, a test can be omitted inside a loop.

Time formula (assuming no elisions):

8 E + (10 E * (#s) + (5 E + TMEMB) * (# y)) * PEQ,

where PEQ is the probability that x and y are equal.

(c.11) y:: RMM(s,β). *Code* and *elisions* are much as in case (a.10), but one additional level of indirection is required to access *pointer*. *Time* formula is

8 E + (11 E * (# s) + (5 E + TMEMB) * (# y)) * PEQ.

To complete the design of an automatic or even semi-automatic system, we would have to extend the preceeding list to cover every SETL primitive operation. We shall not attempt this now. Instead we will let the preceeding discussion stand as an indication of what a full system would require. However, to round out this discussion a little more, we shall sketch out the implementation of the *locate(x,s)* primitive which appears in several of the preceeding insertion and membership test codes. Here the various cases which can arise are as follows:

(d.1) s:: $\subseteq$ s'. *Code* is:

```
    (∀y∈s | s_bit(y) eq 1)
        if x eq y then return y;;
    end ∀y;
    return Ω;
```

Elisions: none.

Time formula: ½ (C_{adv} E * (# s') + (E + TEQ) (#s)),

where TEQ is the time required to make the test x eq y, and where C_{adv} E is the time required to advance in the iteration (∀y∈s').

(d.2) s::R $\subseteq$ s'. *Code* and *elisions* are the same as in case (d.1), but s_bit(y) is differently located. *Time* formula is:

$\frac{1}{2}((C_{adv} + 1)* E * (\# s') + (E + TEQ) * (\# s))$,

where C_{adv} and TEQ are as in (d.1).

(d.3) s::L $\subseteq$ s'. *Code* is:

```
pointer = first(s);  /* the representation of s includes */
                     /* a pointer to the first element */
                     /* of its associated list         */
(while pointer ne Ω)
    if element (pointer) eq x then return pointer;;
/* element(pointer)  retrieves the element of s associated with*/
/* a particular pointer                                        */
    pointer = next(pointer);  /* advance in list */
end while;
    return Ω;
```

Elisions none.

Time formula:

$\frac{1}{2}$ (4 E + TEQ) * (# s),

where TEQ is the time required to test an element of s' for equality with x.

(d.4) s::LR $\subseteq$ s'. *Code*, *elisions* and *time* formula are the same as in case (d.3).

(d.5) s:: $\in$ s'. Here the way in which we proceed depends upon the basing of s'. The possible basings are: s::L $\subseteq$ s", s::L R $\subseteq$ s", s'::{β}, and s'::Ω. The treatments appropriate in these various subcases are as follows:

(d.5.1) s'::L $\subseteq$ s". The quantity s is a pointer to an item *itm'* in the list s', and *itm'* points to an element *actual(s)* in s". Then locate (x,s) is calculated as locate(x, actual(s)).

Elisions: none.

Time formula: 2 E + TLOC",

where TLOC" is the time required to calculate locate(x, $\bar{s}$) for an $\bar{s}$ having the basing $\bar{s}$:: $\in$ s".

(d.5.2) s'::L R $\subseteq$ s". *Code*, *elisions*, and *time* formula identical with case (d.5.1).

(d.5.3) s'::{β}. The quantity s is a pointer to an item actual(s) in the hashtable representing s', and locate (x,s) is calculated as locate(x,actual(s))

Elisions: none.

Time formula: E + TLOC,

where TLOC is the time required to calculate locate(x $\bar{s}$) for an element $\bar{s}$ having the basing β.

(d.5.4) s'::Ω. The quantity s is a pointer to an item of s'; let actual(s) be the item of s' which s references. Then locate(x,s) is calculated as locate(x, actual(s)).

Elisions: none

Time formula: E + TLOC,

where TLOC is the time required to calculate locate(x $\bar{s}$) for an Ω-based set $\bar{s}$.

(d.6) s::{β}. To locate x within s, we enter the hash table representing s using a hash calculated from x. This locates a list, estimated to be 2 elements long, which is searched for an element equal to x.

Elisions: none.

Time formula: 3 E + 2 * TEQ,

where TEQ is the time required to test x for equality with an element having the representation β.

(d.7) s::Ω. We divide this case into the following subcases, depending on the way in which x is based.

(d.7.1) x::Ω. This uses *code* like that presently appearing in the SRTL library. First x is tested to see if it is a tuple of length at least 2. If not, code very much like that of case (d.6) is used; if x is a tuple, then we use its sucessive components to enter a series of hash tables, each item of which points to a successor hash table , until the desired element is located in a final hashtable.

Elisions. If the type of x is known, a test can be elided.

Time formula: 2 * (6 E + 2 * TEQ) * (PNM + NCX * (1 - PNM))

where TEQ is the time needed to test x (or, if x is a tuple, a component of x) for equality with an Ω-based element, PNM is an estimate of the probability that x is a member of s, and NCX is the number of components of x if x is a tuple, or is 1 otherwise.

(d.7.2) s::<β_1,...,β_n>. The successive components of x are used to enter successive hash tables, the items of which point to successor tables, until the desired element is located in a final hashtable.

Elisions: none.

Time formula: 2 * (6 E + 2 * TEQ_1) * PNM
+ 2 * (6 E + 2 * TEQ) * NCX * (1-PNM),

where PNM is an estimate of the probability that x is a member of S, NCX is the number of components of x, TEQ_1 is the time needed to test a β_1-based element for equality with an Ω-based element, and TEQ is the average time needed to test a β_j-based element, $1<j\leq n$, for equality with an Ω-based element.

(d.7.3) s:: ∈ s_1. Here x is either a pointer to an item actual(x) belonging to the representation of s_1 and representing x directly, or, if s_1 has one of the representations $M(s_2,\beta)$, $RM(s_2,\beta)$, $MM(s_2,\beta)$, $RMM(s_2,\beta)$, x is a pointer or pair of pointers, using which representations of x(1) and x(2) can separately be recovered. If x is a pointer to actual(x), then locate(x,s) is implemented as locate(actual(x),s), and the *time* formula is E + TLOC, where TLOC is the time required to calculate locate($\bar{x}$,s) for an Ω-based set s and an element $\bar{x}$ with the basing that belong to elements of s_1. In the remaining four cases x::$M(s_1,\beta)$ etc., we use x(1) to enter a first hash table, the items of which point to a second hash table in which we search for x(2). In these cases, the *time* formula is

E + 2 * (TEQ1 + TCOMP1) + (TCOMP2 + TEQ2).

Here, TCOMP1 (resp. TCOMP2) is the time needed to reconstruct a representation of x(1) (resp. x(2)) from x, and TEQ1 (resp. TEQ2) is the time needed to test x(1) for equality with an element x_1 having the basing x_1:: ∈ s_1 (resp. x(2) for equality with an x_2 having the basing β).

(d.7.4) Cases other than (d.7.1-3): in these cases, *code*, *elisions*, and *time* formula are the same as in case (d.6).

(d.8) s::M(s_1,β). *Code* is

```
if locate(x(1),s1) is ℓx ne Ω andd x(2) eq mapval (ℓx) then
        return ℓx;
    else return Ω;;
```

Elisions: If x(1) $\in s_1$ is known, a test can be omitted; if x ∈ s is known, two tests can be omitted.

Time formula: 3 E + TLOC + TCOMP1 + TCOMP2,
where TLOC is the time needed to calculate the auxiliary function locate(x(1),s), and TCOMP1 (resp. TCOMP2) is the time needed to calculate a representation of x(1), (resp. x(2)) from the representation of x.

(d.9) s::RM(s_1,β). *Code* and *elisions* are as in case (d.8), but mapval(ℓx) is differently located. *Time* formula is

4 E + TLOC + TCOMP1 + TCOMP2,

where TLOC, TCOMP1, and TCOMP2 are as in case (d.8).

(d.10) s::MM(s_1,β). *Code* is as follows:

```
if locate(x(1),s1) is ℓx eq Ω  then
        return Ω;
else /* search for x(2) in the list to which ℓx points */
        pointer = s_first(ℓx); /* the representation of each */
        /* item ℓx  of s1 includes a pointer to the first */
        /* entry of a hashed table of the elements of y{z}. */
        hashquant = hash(x(2)) // numberfield (table(pointer));
        /* numberfield gives the size of the hashed table */
        /* referenced by pointer;  elementfield gives the */
        /* element stored in a particular table position. */
        /* next gives the position of the next element in */
        /* a particular hash clash list.                  */
        pointer2 = pointer + hashquant + 1;
        (while pointer2 ne 0) /* search down the hash chain */
if elementfield(table(pointer + pointer2)) eq x(2) then
            return <pointer, pointer2>;;
```

```
        pointer1 = next (pointer2); /* advance in hash chain */
    end while ;
    return Ω;  /* element cannot be located */
end if;
```

Elisions: If x(1) is known to be in s_1 a test can be elided.

Time formula: 10 E + TLOC + TCOMP1 + .5 * TCOMP2,

where TLOC is the time needed to calculate the auxiliary function locate(x(1),s_1), and TCOMP1 (resp. TCOMP2) is the time needed to calculate a representation of x(1) (resp. x(2)) from the representation of x.

(d.11) s::RMM(s_1, β). *Code* and *elisions* are as in case (d.10), but the quantity *s_first(kx)* is differently located.

Time formula is:

11 E + TLOC + TCOMP1 + .5 * TCOMP2,

where TLOC, TCOMP1, and TCOMP2 are as in case (d.10).

Additional Special Cases.

There exist various special situations in which the equality test x eq y can be performed with particular efficiency. Some of these have been noted in connection with the individual cases (c.1) thru (c.11) described above. Others are as follows:

i. Since the equality test x eq y is symmetrical in its two arguments, either x or y can be decomposed in the manner described in (c.1)-(c.11) above. A reasonable way to proceed is to work out two complete time formulae, one for the case in which x is decomposed, the other for the case in which y is decomposed, and to use whichever one of these decompositions has the most advantageous time formula. Note that this symmetry can be used to avoid certain particularly troublesome cases, e.g., in making the test x eq y when y has the basing y::Ω we can assume that y will never be decomposed unless x also has Ω basing (in which case the existing SETL equality test routine can be used.) This avoids complications concerning tuples in sets that would otherwise have to be faced.

ii. If x::RM(s,β) and y::RM(s,β) then (if x and y have the same number of elements and identical hashcodes) their elements can be checked for equality by a loop over the bitvectors which represent x and y. This leads to a *time* formula

$$6\ E + (5\ E * \#s + (\#\ x) * TEQ) * PEQ,$$

where TEQ is the time needed to test an element of the range of y for equality with an element of the range of x, and where PEQ is the probability that x and y are equal.

A similar remark applies to the case in which x::RMM(s,β) and y::RMM(s,β).

An important special case for the locate(x,s) function is that in which s has one of the basings $s::L \subseteq s_1$, $s::LR\subseteq s_1$, s::{β}, s::Ω, $s::M(s_1,\beta)$, etc., and in which x has the basing $x::\in s$. In this case, locate(x,s) is simply x.

5. Global Tracing of equality relationships.

We observed in section 1 that when conversions are performed two or more representations of the same SETL value can come into existence, and that by tracking the identity of these two representations we may be able to avoid subsequent conversions operations. This can be done as follows:

a. Whenever an assignment x = y or x = convert(y) appears, surmise the relationship x eq y and prepare to track it globally (by assigning a position in a bitvector to represent it.)

b. The equality relationship is restored by each assignment x=y or x=convert(y), and killed by any other assignment to x or y. This rule defines the action,on the relationships x eq y that will be tracked, of each basic block.

c. Use standard techniques to 'globalise' the local facts defined by rule b (the same technique that is used to determine calculation redundancy can be employed.)