SETL NEWSLETTER # 180

S.C. Liu
E. Schonberg
February 14, 1977

Uncovering Profitable Basing Relations

1. Summary

Based representations provide a systematic mechanism for optimizing set-theoretic operations. The gain they provide is twofold:

a) Operations on based sets and maps can be performed without hash-table searches. The otherwise standard hashing and clash-list scanning is replaced by one or two indexing operations, or fast bit-vector operations.

b) The code for the corresponding SETL primitives can be emitted on-line, eliminating the interpretive overhead imposed by the calls to off-line hash-table accessing procedures.

The cost involved in using based representations is also twofold:

a) When based objects are generated, their base is built simultaneously, "behind the scenes" as it were. Inserting a new element in a set forces its parallel insertion into the corresponding base, an operation slightly more expensive than normal (unbased) set insertion, because an element block, generally several words long, must be allocated.

b) Bases are bulky: each element block must accomodate the value of all based functions that are defined on some subset of the base. If the domains of definition of these functions cover a small fraction of the base, the space waste can be considerable. This is the main reason for allowing the definition of subsets

which are themselves bases: they can then be used more efficiently (in terms of storage) as domains of based maps.

Leaving aside for now the question of storage optimization, it is important to notice that both cost and gains connected with the use of basings can be quantified in terms of the number of insertion operations and hash-searches performed. The cost of set insertions is itself the combination of a hash-search and a storage request, and little will be lost if we disregard the latter. This means that we will disregard the difference between an unbased set insertion, and a base insertion. If we use this somewhat simplified measure, and apply it to a program graph, annotated with frequency information (even as coarse as a nesting depth for each statement), we are led to a rather straightforward algorithm for generating useful basing relations for programs without declared basings.

Outline of a method for inferring useful basings

The operations whose presence will suggest certain based representations for program objects fall into three categories.
a) Operations which involve one or more membership tests. These operations are invariably performed off-line when applied to unbased objects. We can further subdivide these operations into global ones, whose operands are two sets, and differential ones, whose arguments are a set and a (possible) element of it. Global operations include the predicates incs and subset, and the set operations: +, *, -, //. Differential operations include the predicates in and notin, the insertion and deletion ops: with, less, lessf, and the various map accessing operations.

b) The set iterator (e.g. the loop: ∀x∈S) must be examined separately. It does not by itself suggest a based representation for x and S, but favors some linked representation for S. Its optimization in a based environment will be discussed separately.
c) Other SETL primitives which create atomic values: arithmetic and string primitives, etc. Their execution is of course independent of the presence of basings, but if the target (ovariable) of the operation is assumed to be an element of a base, then it must be inserted in this base after creation. Thus operations of this type are seen to bear the expense of enforcing the basing choices suggested by operations of type a).

We start by noting that, for all operations of type a), basings are invariably advantageous. The gain over unbased representations can be very large (as in the bit-vector case of set union and intersection) or marginal (as in the case of membership in a remote bit-vector) but in all cases a gain is available. Therefore, if we use based representations for the objects involved in the operations of highest frequency through the program, execution speed of that program can only increase. We start by postulating basings in terms of these operations, and then propagate these choices by means of the use-definition chaining functions. The costs incurred by the original choices of basings make their appearance when we reach instructions of type c), i.e. the 'value-source' nodes of the use-definition graph.

If we use the total number of hash-searches performed as a measure of program expense, and have frequency information available for each instruction, it is a simple matter to compare

the cost of the based and unbased versions of a program. It is important to note that, if we start from the instructions of highest frequency, this procedure need not backtrack at any point to examine other basing choices: the initial choice can only be profitable. If the instructions of highest frequency are of type c), i.e. create new values that would have to be inserted into bases, then of course no basing choice would be useful, and we can immediately decide to leave such a program unbased.

A few qualifications must be appended to this optimistic claim. We will show below that they lead to various refinement in the choice of basings, without affecting the fundamental structure of our procedure. We must now discuss the nature of the initial basing choices, and the propagation of these choices to all program variables.

Step 1. Initial basing choices.

We examine first the instructions of highest frequency. If they are of type a), some basing choices suggest themselves: all global operations for example, suggest bit-vectors representations for both of their arguments. Membership tests suggest subset and element-of representations, etc. Table I lists the SETL primitives of type a), and the most favorable based representation for their arguments. Each one of these operations introduces a base B, whose exact composition is at this point undetermined.

If several operations of high frequency appear, basings are postulated for the arguments of each. At this point, each

operation is taken to suggest a different base, even if the same variable may appear in several of them. Initially, these bases are only linked to variable occurrences. In the next phase of the algorithm we are sketching, the various bases are merged so that they correspond to bases of program variables instead. For example, if the innermost loop of a program includes the code:

```
L1:    f(x)=y;
L2:    S with x;
L3:    if z in S then y=y+1;;
```

then our initial basing choices are as follows:
(We use the line number as a subscript to specify various variable occurrences)

f_1: map ($\in B_1$)...
x_1: $\in B_1$
S_2: set ($\in B_2$)
x_2: $\in B_2$
S_3: set ($\in B_3$)
z_3: $\in B_3$

It is only later that we will make the obvious identification $B_1 \equiv B_2 \equiv B_3$. The basing chosen for S above has of course been described imprecisely. It could be a local, or a remote subset; or it could be represented as a set of elements of the base. We will start with the optimal choice (in this case, local subset) but consider this as tentative and revise it if necessary, in subsequent phases of the process.

Step 2. Propagation of basing choices.

The initial choices made for a few program variables must now be propagated to the point of creation of the corresponding values. For example, if the basing: $x:\varepsilon B$ has been hypothesized, then all instructions which give a value to x must be forced to be compatible with this choice. The use-definition map *ud* is our basic tool in this phase. We must distinguish two cases:

a) An occurrence x_i of variable x, for which basing b_i has been suggested, is chained to an ovariable occurrence x_o, in an operation O, of type a), or on a simple assignment. For example, suppose that x_i is created by one of the statements:

```
x=y;
:
x = arb S;
```

Then the basing choice for x_i is propagated to the ivariables of operation O. In the case above, y is postulated to have the same basing b_i as x, and S to have basing $\{b_i\}$. These choices are now propagated through the *ud* links of y and S.

b) If the instruction O to which x_i is linked, is an instruction of type c), no further basing propagation is possible. A base-insertion opcode is introduced at that point.

The above suggests that we regard *ud* as defining a basing transmission graph (BTG) on which our cost estimates are evaluated. Terminal nodes of the graph correspond to operations of type c).

The basing transmission graph

The basing transmission graph hypothesized in the preceeding paragraph has the following structure:

a) Its nodes are pairs <instruction, frequency>.

b) Its edges are elements of the use-definition chaining function, *ud*.

The rules for basing propagation are as follows:

a) Basing constraints propagate backwards, i.e. from ivariables to ovariables.

b) Basing constraints are propagated only from instructions of greater frequency to those of smaller frequency. If a basing constraint appears at instruction i_1, with frequency f_1, and if *ud* leads from $<i_1,f_1>$ to $<i_2,f_2>$, with $f_2>f_1$, then we distinguish two cases:

a) i_2 is an instruction of type c). Then it is not advantageous to impose the basing constraint on i_2. Instead, an appropriate conversion operation is inserted at i_1.

b) Otherwise, i_2 transmits an already existing value. The basing constraint is propagated tentatively to i_2, and a depth-first-search starting at i_2 tries to determine whether the cost of that constraint can be pushed below the frequency f_1. If this is not possible, the constraint is dropped and a conversion operation is introduced at i_1. Such a procedure is seen to be linear in the size of the graph, provided that we can convince ourselves that handling loops in the basing propagation graph presents no difficulty.

Loops in the BTG

Apart from trivial cases, such as

```
x=y; ... /*code using x,y */ y=x;
```

loops in the basing propagation graph will reflect the existence

of program loops which repeatedly modify a given variable. For example, consider:

```
(∀x∈S)
   S1 with f(x);
   :
   S1 less g(x);
end ∀;
```

It is clear that both occurrences of S_1 are linked in *ud*, so that the BTG contains a loop L. It is also clear that both instances of S_1 should have the same (based) representation. (This is not always the case and we discuss in the next sections the problem of conflicting basing constraints).

The following code shows another instructive but somewhat more complicated case:

```
(∀x∈S)
  (∀y∈T)
     S1 with f(y);
     :
  end ∀y;
   :
   S1*S2;
end ∀x;
```

Here again, both instances of S_1 are linked by *ud*. However, the choice imposed in the inner loop: S_1:set(∈B), f: map(∈B_2)∈B will propagate outward only (i.e. towards operations appearing at a shallower depth) but the outer basing will not propagate inwards, because of our rule against propagation to higher frequencies. (Note that the bit-vector representation suggested by the

intersection operation will eventually be chosen, as a refinement of the original choice: set(εB)) Loops in the BTG are therefore broken by suppressing edges that go from lower to higher frequencies. If this is done, then basing propagation will proceed monotonically through the graph.

Having in this way made our BTG loop-free, we must now examine the way in which we will handle merging paths in it, i.e. the way in which we will either "fuse or resolve" conflicting basing constraints which arise since the BTG (even if loop-free) is generally not a tree.

Consider the code fragment:

```
x from S;
(∀y∈T)
   z=f(x);
   .
   .
   .
   r=g(x);
   .
   .
   .
end;
```

The scheme that we have outlined starts by setting the following:

f:map(εB_f)gen

x:εB_f

Next, it propagates this choice to S:set(εB_f) and thence to the points of creation of S. After this, we process the other high-frequency instruction, and set:

g:map(εB_g)gen

x:εB_g

but of course x has already been based. At this point we must make one of two choices, and either identify the bases B_f and B_g

(which is clearly reasonable, being that at least some of their elements overlap) or insert a conversion at the point where x is created, so that two copies of x exist:

```
x_f from S;
x_g=locate(x_f,B_g);
(∀y∈T)
   z=f(x_f);
   :
   r=g(x_g);
end;
```

The interesting point to note is that this type of basing conflict can always be resolved by enlarging the base, because bases for a given program are fully defined only after all the based variables have been chosen! The main reason for choosing a double representation for x instead of a common base for f and g, is one of space optimization: f and g might have disjoint domains of definition (except for element x!) and a common base might therefore be wasteful of storage. But this indicates that within our scheme, space and time optimization are disjoint processes. We can start with the optimal choice of basings for speed, and then undo some of our basing choices to bring storage use down to some appropriate level. In any case the two optimizations can be treated independently.

Other possible basing conflicts which might have to be handled by a generalization of this approach are:

a) Remote vs. local representations. It is probably reasonable here to decide in favor of the remote representation.

b) Conflicting bases for composite objects. This is similar to the case just discussed.

c) Different basings for composite objects. The typical case is that of a set, which is also an element of another set:

S:set(εB)

S:εBS

This is actually not a basing conflict, because the structure of BS is not fully determined yet. These two basing constraints can be made consistent by setting BS: base (set(εB)), i.e. relating the two bases which appear in our conflicting basing constraints.

d) We have so far ignored the iteration operator and its relation to basing representations. It is clear that the linked hash-tables used to represent bases and unbased sets are optimal for iteration. Based representations, be they local or remote, are more expensive, because they involve an iteration over the base, and a membership test in the based subset. Furthermore, overhead incurred by iterating over the base will be greater, the sparser the based object is. It will therefore be convenient to generate unbased representations of sparse objects, if they are frequently iterated upon. This conversion is totally dependent on size information, and can only be chosen by some dynamic test (i.e. by comparing cardinalities of base and based object during program execution). We therefore propose to handle sets for which iteration is a critical operation, by introducing conditional conversion operations in the code, at points where the

based object is completely built, and only if the conversion would occur at a point of lower frequency than the iteration we are trying to optimize.

An Example

We examine an algorithm for finding the spanning tree of a graph (K1, L1). See Low's analysis for comments on the technique used.

```
module spantree; var nodes, edges, father;
read nodes, edges;

(∀x∈nodes) father(x)=x;;

(∀e∈edges) f=e(1);
           s=e(2);
           fg=groupof(f);
           sg=groupof(s);
           if fg ne sy then
              treeset with e;
              merge (fg,sg);
           end if;
end ∀;
print treeset;

definef groupof(node);
        while (father(node) ne node)node=father(node);
        return node;
end;
define merge(g1,g2); father(g2)=g1; return; end;
end;
```

The instruction of greater frequency in the main program is the invocation of *groupof*. We consider therefore the loop in that procedure as our starting point. Our first choice is:

father:map(εBN)εBN

node: εBN

This choice propagates to variables f and s, and thence to e and edges, which receive the basing:

e:<εBN,εBN>

edges:set(<εBN,εBN>)

The cost of that choice is eventually propagated to the read statement. Returning to the main loop, we consider the variable *treeset*. The optimization of its lone appearance in the code suggests the basing:

treeset:set(εBE)

e:εBE

Notice that e has already received a basing specification. But the new one is not incompatible with e being a pair of nodes. This new constraint on e propagates to its source, the iteration loop over edges. We now specify that edges:set(εBE) and the base BE receives the description

BE:base(<εBN,εBN>)

All that remains is to propagate our choice for *father* to the loop over nodes. We obtain nodes:set(εBN);

A further optimization becomes available when we put the base insertion statements needed to generate BN and BE into the code. They only appear at the point of input, i.e. immediately after *nodes* and *edges* have been read. This means that *nodes* can

directly be identified with BN. On the other hand, when *edges* is read, it must be converted, to mode set<εBN, εBN>. Then we make the identification of *edges* and the base BE. *Treeset* can be a local subset of that base. *father* is a local map on BN, because nothing requires that it be made remote. The basing choices are now complete.

References

[K1] Knuth, D.E. The Art of Computer Programming, Vol. 1, p. 353 Addison Wesley, Reading, Mass., 1971.

[L1] Low, J. Automatic Data Structure Choice--an overview. U. of Rochester preprint, Nov. 1976 (unpublished.

Table I. Initial basing choices

Opcode	argi1	argi2
in, notin, with, less	εB	{εB}
incs, subset, +, *, -, //	{εB} (remote bit-vector)	{εB}
domain	map {εB}--	
range	map (-) εB (but not useful) (conceivably treat the range of f explicitly as a subs	
f(x)	εB	map (εB)--
f{x}	εB	map {εB}--
f[S]	{εB}	map {(εB)}{$εB_1$}
f(x)=y	same as retrieval.	
f{x}=S	same as retrieval.	
next	Nothing. Suggests hash-table, but not binding: however, if nothing suggested so far, set-of-elements-of may be tried.	
arb	{εB}	