AUTOMATIC DATA STRUCTURE SELECTION IN SETL

Edmond Schonberg
Illinois Institute of Technology

and

Jacob T. Schwartz and Micha Sharir
Courant Institute of Mathematical Sciences

Abstract

SETL is a very high level programming language supporting set theoretical syntax and semantics. It allows algorithms to be programmed rapidly and succinctly without requiring data structure declarations to be supplied, though such declarations can be manually specified later, without recoding the program, to improve the efficiency of program execution. We describe a new technique for automatic selection of appropriate data representations during compile-time for undeclared, or partially declared programs, and present an efficient data structure selection algorithm, whose complexity is comparable with those of the fastest known general data-flow algorithms of Tarjan [TA2] and Reif [RE].

1. Introduction

The level of a programming language is determined by the power of its semantic primitives, which influence the ease and speed of programming in the language profoundly. (See [HA] for an attempt at quantifying these abstract concepts.) Thus a language of very high level should provide high level abstract objects and operations between them, high level control structures and the ability to select data representation in an easy and flexible manner. It is the third property of very high level languages that we address in this paper.

In relatively low level programming languages, data-structures have to be selected in advance, before starting to code the program; the code to be written then depends heavily on this selection and large sections of it are solely devoted to the manipulation of the selected data-structures. These lengthy code sections constitute a significant source of bugs and become deeply imbedded in the program logic to the extent that they have to be replaced or modified when we want to change the data representation.

The programming language SETL, being designed and implemented at New York University, will serve in this paper as a prototype of a very high level language which treats the selection of data structures in a different way. We will describe an automatic technique which enables the SETL programmer to code his program in a high level, relatively independent of specific data-structure, and yet allows a reasonable level of efficiency to be achieved.

In the SETL system, the data representation used to realize an algorithm depends on its code and not vice versa. More specifically, algorithms are coded without specifying any concrete data structures at all. The objects appearing in a program are (dynamically) assigned appropriate abstract data types from among the basic data types supported by the

This work was supported by the National Science Foundation Grant MCS-76-00116 and the U.S. D.O.E. Office of Energy Research Contract EY-76-C-02-3077.

language. In the optimizing version of SETL, each such data type is viewed as a generic indication representing a collection of more specific data-structures, all of which are capable of representing the same abstract data type. Thus the data type 'set' can be represented as a hash table, or a linked linear list, or a bit-string, etc. However the semantic features of the abstract SETL data types, as well as of the operations on them, are independent of any specific data structure selection. Thus program code need not be modified when this selection is made.

Once an algorithm in SETL has been coded, debugged and executed successfully (and in future endeavors, also proved correct), data-representations can be selected in order to improve execution efficiency. Two selection techniques are provided. The more conservative technique is manual. In this technique the program text is supplemented with declarations specifying data structure representation for some (or all) program variables to be used. If these representations are consistent with the abstract data types actually acquired by the declared variables during execution, then the supplemented program will be precisely equivalent to the original purely abstract one, but can run much more efficiently.

In the second, more ambitious data-structuring mode, which is the one to be described in this paper, data-structure selection is performed automatically by an optimizing compiler. This is in general a complex task because each particular data structure will usually be more efficient for some instructions and less efficient for others, so that in order to arrive at a realistic evaluation of the cost of using alternative data structures for a given program variable, we must perform an appropriate global analysis of the way in which program objects are used and related to each other.

Our main aim in this paper is to present an automatic data structure choice algorithm. While this algorithm reflects the particular semantic environment of SETL, and therefore cannot be regarded as a general purpose automatic data structure selection procedure, it does establish the possibility of performing automatic data structure selection in a reasonably efficient manner in a language of very high level.

This paper is organized as follows: In Section 2 we briefly review SETL and its manual data structuring system, known as the 'basing' system. Section 3 will describe our data-structure selection algorithm in detail, including examples illustrating the way in which this algorithm applies to sample programs.

We take this opportunity to thank several members of the SETL project at New York University, namely: Robert B. K. Dewar, Ssu-Cheng Liu and Arthur Grand, for numerous suggestions concerning automatic data-structure selection. In particular, our work has been greatly influenced by earlier work of Liu [LI].

2. The SETL Language and its Basing System

We now summarize the principal features of the SETL language. SETL admits finite set theoretical objects, such as arbitrary finite sets, maps and tuples, and supports most of the operations between these objects. The language also supports the major elementary data types, found in most programming languages, in particular integers, reals and strings. 'true' and 'false' are strings used to represent boolean values.

Tuples in SETL are arbitrary-length, dynamically extensible ordered sequences of component values, which can be either primitive or themselves structured. Tuple concatenation, indexed retrieval and storage of components, subtuple retrieval and storage operations, and a tuple length

operator are provided.

Sets in SETL are unordered collections of elements (these elements being primitive or structured), such that no element can appear in a set more than once. SETL provides the usual set-theoretic operations (union, intersection, etc.), and some special operators, summarized below. SETL also provides general set former expressions, universal and existential quantifiers, compound operators on sets and similar high level constructs.

Maps in SETL are simply sets of tuples of length 2 (called pairs), and can represent both functions and relations. SETL provides functional style constructs for map retrieval and storage, as well as several special operators, e.g. domain and range, which compute the domain set and the range set of a map. Since maps are sets, all set-valued operators can also be used with maps. Data structures such as trees and graphs are represented in SETL using maps which give the relationships between the elements of the structure, without having to specify the detailed storage structure to be used.

A special value OM (for omega) is used to indicate an undefined value. All variables are initialized to OM and certain operations can also yield OM if domain constraints are not met. For example, v(i) yields OM if v is a tuple whose present length is less than i.

The SETL control structures are generally conventional, and include the if-then-else clause, case statements, while loops, numerical iteration, etc., and also a few additional very high level control structures, e.g. iteration over a set, universal and existential quantifiers, etc.

SETL, like APL, has value semantics rather than pointer semantics. This means that the value of each program variable is essentially independent of the values of other variables and will not be affected when other variables are modified. For storage optimization, several variables can share a common value using pointer mechanisms, but care will always be taken to create new copies of that value whenever logically necessary. This value semantics also implies that subprocedure parameters are passed by value rather than by name, and that no explicit reference and manipulation of pointers are allowed.

The following table summarizes the main operations of SETL. For additional details, see [DE] and [SC3].

Table 1. Some of the Primitive SETL Operations.

Operation	Remarks
x + y	integer and real addition, set union, tuple and string concatenation.
x - y	integer and real subtraction, set difference
x * y	integer and real multiplication, set intersection, string and tuple repetition
x / y	arithmetic division
x and y, x or y, x implies y not x	Boolean operations
#x	cardinality of sets, length of tuples and strings
x = y, x /=y	equality and inequality comparisons
x < y, x > y, x >= y, x <= y	arithmetic comparisons, string lexicographical comparisons
x with y	set insertion, tuple incrementing
x less y	set deletion
x in y	set membership test
arb x	select arbitrary element of x
x from y	select and delete an arbitrary element of the set y
{x,y,...}	set with specified elements
[x,y,...]	tuple with specified elements
f(x)	function or subprocedure call, indexed retrieval from a tuple or a string. If f is a map,

	f(x) is the unique y such that [x,y] in f, if such y exists; otherwise f(x) = om (undefined).
f{x}	the set of all y such that [x,y] in f; f must be a map.
domain f, range f	operators producing domain and range of maps
x := y	simple assignment, yielding y as a value
f(x) := y	map storage (which will cause all other pairs [x,z] to be deleted from f) indexed assignment for tuples and strings
f{x} := y	map storage corresponding to the retrieval operator f{x}
f(x...y)	extract subpart starting at component number x and ending at y of the tuple or string f (corresponding storage operator is also available).
newat	generate a new unique atom
type x	the current type of x

In addition to the above primitives SETL provides several high level constructs involving explicit or implicit iterations over sets and tuples. Table 2 summarizes some of these constructs. In this table, the clause-type 'iterator' denotes a construct of the form

$$x_1 \text{ in } e_1,\ x_2 \text{ in } e_2(x_1),\ldots,x_n \text{ in } e_n(x_1,\ldots,x_{n-1}) \mid c(x_1,x_2,\ldots,x_n)$$

where $e_1,\ldots,e_n$ are set-valued expressions, and iteration is carried out by assigning all possible elements in the corresponding sets to $x_1,x_2,\ldots,x_n$. Moreover, $c(x_1,\ldots,x_n)$ is a Boolean valued expression, which defines the sets of values to be bypassed in the iterations.

Table 2. Additional High Level SETL Constructs

Name	Form
iteration over sets	(∀ iterator)
set former	{exp($x_1,\ldots,x_n$) : iterator} where $x_1,\ldots,x_n$ are the free variables of the iterator.
tuple former	[exp($x_1,\ldots,x_n$) : iterator] where $x_1,\ldots,x_n$ are as above (if order is important, all iterations should be numerical.)
existential quantifier	∃ iterator the free variables $x_1,\ldots,x_n$ are set to their first value during the iteration for which the condition c is true, and are otherwise undefined. The expression yields 'true' or 'false' respectively.

The special constant nl denotes the null set (or map).

Let us now describe the data-structures which SETL supports. Tuples are represented as dynamic arrays of consecutive memory locations (which have to be re-allocated if their length increases beyond their current allocation); sets are represented as 'breathing' linked hash tables, so that iteration over a set is fairly rapid, but most of the other set operations involve hashing into a set, an operation which can be expected to be relatively costly. In fact, as will be seen later, the main optimization that our automatic selection of data structures aims to achieve is to minimize the number of hashing operations performed during the execution of a program.

The default representation of maps is either their standard set representation, or, if it can be asserted during compilation that an object will always assume a map value, then it is represented as a linked hash table, hashed on the domain elements of the map, where each entry in the table points to the range value of the corresponding domain element. This expedites map-related operations, such as map retrieval and storage, but makes global set operations, such as set union, slightly more cumbersome than they are for the default set representation.

In order to obtain a coherent extended class of efficient data representations, we introduce new program objects, called bases, which are used as universal sets of program values (cf. [DE] for a more detailed description). Bases enable us to access related groups of program variables in specially efficient manner. This is the only use of bases.

Once one or more bases have been introduced, other program variables can be described by their relationship to these bases. The set of these declarations will determine the run time value of these bases, because bases are constantly updated to maintain the validity of these relationships. For example, we can declare a program variable x to be an element of a base B by writing

x: ∈ B;

If this declaration is made, then during execution, any value assigned to x is automatically inserted into B, unless this value is found to be already in B.

Internally, a base B is represented as a linked hash table of 'element blocks'. Each such block contains a program value (or a pointer to such a value) and as many additional fields as are needed to store program variables which have been declared to be based on B. A variable x declared as ∈ B is represented by a pointer to some element block of B, so that the actual value of x can be obtained simply by dereferencing this pointer. Whenever x is represented in this way we shall say that x is located in B. Computation of such a pointer, which may imply insertion of a new value into B, will be called a base locate operation. A main aim of our data structuring system is to minimize the number of these 'search and insert' operations (normally realized as hashing operations) which are performed during program execution.

In addition to the 'element of a base' representation described above, which can be declared for any program variable, related based representations are available for sets and maps. A set s can be declared as a subset of a base B by writing one of the following declarations:

s: local set (∈ B);
s: indexed set (∈ B);
s: sparse set (∈ B);

If the first declaration is used, a single bit position is reserved in each element block of B; this bit is on iff the element represented in the block belongs to s. This representation of s supports very fast insertion and deletion of elements from s and membership tests, operations that would have otherwise required hashing into s, and it corresponds to the familiar notion of an attribute bit (or flag) in a plex structure.

If the second declaration is used, then all these bits will be grouped into a bit-string, stored independently of B. In order to access this string via B, a unique index is assigned to each element block of B when this block is created. The bits of s are arranged in the order of these indices. Thus, to perform the insertion operation 's with x;' where x is declared as ∈ B, the following steps are taken:

(A) retrieve i(x), the index of the element block to which x points.
(B) turn on the i(x)-th bit of s.

Note that in both cases the set inser-

tion operation 's with x;' is fast only if x is represented as an element of B. In any other case, the value of x must first be hashed into B to find the corresponding element block (or create a new element block if the value of x is not yet in B). This observation indicates that based representations are profitable when consistent basings are given to all the variables involved in hashing operations, using the same base. Even when this is done, hashing operations will still be required to create B; however, they will be fewer than in the unbased case.

The first two declarations have one common disadvantage. To iterate over s we must iterate over B and perform a membership test in s for each element in B. This is certainly less efficient than a direct iteration over s as would have been done in the unbased case, and is especially so if the cardinality of s is much smaller than that of B. When iteration over such a sparse set is performed often, we can use the third representation, in which s is represented by a linked hash table whose entries are pointers to element blocks of B. This representation supports fast iteration over s, and is slightly more advantageous than the unbased representation for search, insertion and deletion from s, since equality of pointers can be checked rapidly whereas equality of general values is much more expensive.

Similar alternative representations are available for maps. A map f whose domain is a subset of a base B can be declared in one of the following ways:

f: local map (∈ B) *;
f: indexed map (∈ B) *;
f: sparse map (∈ B) *;

where * denotes any representation for the range of f. We will temporarily assume that f is single-valued, to simplify the description of its corresponding representations. Thus for example, 'map (∈B) int' denotes a map from some subset of B to some set of integers.

If the first declaration is used, a fixed field within each element block x of B is allocated and reserved for storing the value, or a pointer to the value, of f(x) (or om if f(x) is undefined). This makes map retrieval and storage operations very efficient. For eammple, the instruction 'y := f(x);', where x is represented as an element of B, can be implemented as a simple indexed load, and the need to hash on x in f is eliminated. This local map representation captures the familiar notion of data structures consisting of plexes (base element blocks) within which various fields contain either values (if the range of the corresponding map is unbased) or pointers to other plex nodes.

Local representation of sets and maps have one basic disadvantage. Since they are allocated in static fields within each element block of a base, their number must be predetermined at compile time. Also, no other variable can share their value without violating the value semantics of the language. Thus whenever such variables are assigned to other variables, or incorporated within other composite objects, or are passed as parameters to a procedure, their value must be copied first. Hence, if such a variable is frequently subjected to operations of this kind, its representation may be quite inefficient.

The indexed representation is provided to avoid such problems. If the indexed declaration is used for f, then an array V, disjoint from B, is allocated for f and contains the range values of f in the order of the element-block indices in B. Thus to retrieve f(x), provided that x is represented as '∈ B', two indexed loads are performed, as follows:

(A) retrieve i(x), the index of the element block to which x points.
(B) retrieve V(i(x)).

We will refer to each of the data structures which can be declared in the

above-described basing system as a mode. Modes thus range from primitive modes such as int, real, atom and string, to composite modes which can be nested to any level. For example, a two dimensional map can be represented as

local map ($\in B_1$) indexed map ($\in B_2$) int

A value of a map declared to have this representation can be retrieved using three indexed loads, which is even faster than retrieval of a component of a two dimensional array in FORTRAN.

We now illustrate the preceding considerations by an example showing the process of declarative data-structure selection in SETL. The following program computes a minimum cost path between two nodes in a directed graph (statement numbers are given for later reference):

```
     Program minpath;
(1)  read (graph,cost,x,y);
       $ read in graph and auxiliary infor-
       $ mation.  graph is a set of edges,
       $ cost is an integer-valued map on
       $ these edges, x is the source node
       $ and y is the target.
(2)  prev := nl;  $ prev is a map from
       $ each encountered node to its
       $ 'cheapest' predecessor.
(3)  val := nl; $ val maps each encounter-
       $ ed node to the minimal cost of a
       $ path reaching it from x.
(4)  val(x) := 0; $ start node has zero
       $ cost path
(5)  newnodes := {x}; $ initially, start
       $ node is newly reached
(6)  (while newnodes /= nl) $ while there
       $ exist newly reached nodes
(7)     n from newnodes;
           $ select such a node
(8)     (∀m∈graph{n})
           $ and for each of its successors
(9)        newval := val(n)+cost(n,m);
           $ calculate new path cost
(10)       if val(m)=om or val(m)>newval then
           $ cheaper  path
(11)         val(m) := newval;
             $ note path value
(12)       prev(m) := n;
             $ and cheapest predecessor
(13)       if m /= y then
             $ keep searching if goal
             $ not reached
(14)         newnodes with m;
           end if;
         end if;
       end ∀;
     end while;

if val(y) = om then  $ y is not reachable
                     $ from x
  print('y is not reachable from x');
else
  path := [y]; $ build up reversed path
  z := y; $ starting with the last node
  (while (z := prev(z)) /= om)
          $ chain to preceding node
    path with z;
  end while;  $ and now reverse path
  path := [path(#path+1-i):i:=1...#path];
  print(path);
end if;

end program minpath;
```

This is the 'pure' SETL program in which no specification of data structures has been supplied. After this program has been coded and tested, we can select efficient data structures for its variables. A typical choice might be as follows: introduce a base nodes, which will be the set of all nodes in the graph. Then declare:

```
graph: local map (∈ nodes) ∈ nodes;
cost: local map (∈ nodes) indexed map
                              (∈ nodes) int;
newnodes: local set (∈ nodes);
prev: local smap (∈ nodes) ∈ nodes;
       (i.e. single-valued map)
val: local smap (∈ nodes) int;
x,y,m,n: ∈ nodes;
path: tuple (∈ nodes);
```

In the presence of these declarations, a hash table will be generated for the base nodes. This hash table will be filled in automatically as soon as the graph is read in by our program. Each block in this hash table will contain several fields which store values, bits and pointers to other entries in this table. After the input phase, the rest of the program is executed without performing a single hashing operation. We pay a very small price for this huge saving when we print path, since each component must be dereferenced to its actual value before being printed out. We stress the point that we can map our objects onto lower level data structures and generate code sequences quite close to those which would appear in a lower level variant of our algorithm (such as one written in PASCAL or PL/I) and attain comparable efficiency without recoding the original form of our algorithm at all.

This example indicates that the trade-off between programming language level and execution efficiency need not be as unfavorable as is generally expected. Of course, in our example we lower somewhat the level of 'pure' SETL, by supplying the data-structure declarations. However, this program supplement is very small, compared with the labor that would be involved in an actual recoding of the algorithm in a lower level language. Furthermore, as will be shown in the following sections, the data-structures that we have declared can be selected automatically, thus relieving the programmer of the task of specifying any data structures at all.

3. An Automatic Data-Structure Selection Algorithm

The algorithm to be sketched below is not our first attempt at performing automatic data-structuring in SETL; see [SC], [SL] and [DE]. A strategy common to these approaches, and to our new algorithm as well, is to generate provisional bases and corresponding based representation for variables involved in operations otherwise requiring hashing. These provisional bases initially reflect only local information, and separate bases are generated for each hashing operation. To integrate these "local" provisional basings into an overall basing structure, they are propagated globally. During propagation individual bases are equivalenced whenever logically appropriate. We feel that the efficiency and simplicity of our present algorithm makes it our best candidate for performing automatic data structuring in SETL.

To illustrate the strategy which this algorithm employs, consider the following SETL code fragment (taken from the MINPATH program shown in the previous section):

```
(1)        n from newnodes;
(2)        prev(m) := n;
             ...
(3)        newnodes with m;
```

In this code, instructions (2) and (3) implicitly require hashing operations. Thus we first generate two bases B_2, B_3 and provisionally establish the representations

$prev_2$: map ($\in B_2$) *; m_2 : $\in B_2$;
$newnodes_3$: set ($\in B_3$); m_3 : $\in B_3$;

(where v_i denotes the occurrence of a variable v at instruction i, and * denotes any mode). Note that we generate representations for variable occurrences rather than for the variables themselves; this is because SETL is only weakly typed, so that each variable may assume more than one data-type in the course of execution. (Various problems arising in view of this fact will be addressed later on.) However, since there exists a data-flow link between m_2 and m_3, it follows that m_2 and m_3 can assume the same value, which must therefore be an element both of B_2 and B_3. We therefore merge the representations of m_2 and m_3 into one common representation by identifying B_2 with B_3. We can then

propagate the resulting basing to instruction (1), which is a retrieval operation whose execution speed is virtually independent of the representation of its arguments. Then, in view of the data-flow link between $newnodes_1$ and $newnodes_3$, we give $newnodes_1$ the same representation as $newnodes_3$, and also put $n_1 : \in B_3$. An additional propagation step, using the $n_1 - n_2$ link, gives $prev_2$ the representation '$\underline{map}$ ($\in B_3$) $\in B_3$ '.

Our algorithm mechanizes the strategy that has been sketched above in a relatively efficient and simple manner. Before describing this algorithm, let us sketch the form of input it assumes. We suppose that the program to be analyzed has already undergone several other analyses and optimizations, including a modified version of the definition-use chaining analysis (cf. [AL]) which computes a data-flow map, called BFROM, whose definition is as follows: Let VO_1, VO_2 be two occurrences of a variable V. Then $VO_1 \in BFROM\{VO_2\}$ iff VO_2 is a use of V and their exists an execution path leading from VO_1 to VO_2 which is free of other occurrences of V (cf. [SC_2] for more details).

We also assume that type analysis has been performed using Tenenbaum's approach [TE], so that type information will have been assigned to each variable occurrence, in a manner ensuring that the calculated type of each variable occurrence dominates every data type that the variable can acquire at that program point during execution.

Assuming all this, our algorithm consists of the following phases:

(a) base generation
(b) representation merging and base equivalencing
(c) base-pruning and representation adjustment
(d) name-splitting.

(a) $\underline{base\ generation}$: This phase performs a linear pass through the code being analyzed. For each instruction I we introduce enough bases and corresponding based representations for arguments of I to ensure that execution of I with these based representations for its arguments is not slower than execution of I with unbased arguments. For example,

(i) Suppose that I is 'S $\underline{with}$ x'. Then we introduce a base B_I, and provisionally assume the representations S_I: $\underline{set}$ ($\in B_I$); $x_I : \in B_I$. Note that here the introduction of B_I speeds up the execution of I considerably.

(ii) Next, suppose that I is 'f(x) := y'. Here we introduce two bases, B_I^1, B_I^2, and provisionally represent f_I: $\underline{map}$ ($\in B_I^1$)$\in B_I^2$; x_I: $\in B_I^1$; $y_I : \in B_I^2$; here only B_I^1 is essential, since its introduction eliminates a hashing operation, and we refer to B_I^1 as an $\underline{effective}$ base. The introduction of B_I^2 does not speed up execution of I (but does not slow it down either), and we refer to B_I^2 as a $\underline{neutral}$ base. The utility of neutral bases will become clear in our description of the next phase of the data structure choice algorithm.

(iii) Next, suppose that I is 'x := v(j)', where v is a tuple. Here no speed-up is possible, but nevertheless we introduce a (neutral) base B_I , and provisionally establish the representations v_I: $\underline{tuple}$ ($\in B_I$); $x_I : \in B_I$; .

(iv) Finally, suppose that I is 'x := x + 1'. Here no base can be introduced without running the risk of slowing I down significantly, because of the possible introduction of conversion operations for the newly created values of x , between their $\underline{int}$ mode and their tentative ($\in$ B) mode. Hence we introduce no base for I.

In this first phase we also build up a map EM, mapping each generated base to the mode of its elements. During this phase all these modes are unbased, but they may be transformed into based modes during

phase (b).

(b) representation merging and base equivalencing: This phase executes a linear pass through the map BFROM. For each pair of variable occurrences $(VO_1, VO_2) \in$ BFROM such that both VO_1 and VO_2 have the same type, and supposing that VO_1 and VO_2 have both received based representations in phase (a), we perform the following representation merging operation (recursively):

Let R_1, R_2 denote the based representations of VO_1, VO_2 respectively. Three cases are possible:

(i) Both R_1 and R_2 are base pointers, i.e. R_1 is $\in B_1$ and R_2 is $\in B_2$. In this case, equivalence B_1 and B_2.

(ii) One of these representations, say R_1, is $\in B_1$ and the other is a composite based representation. In this case, merge $EM(B_1)$ with R_2, setting $EM(B_1)$ to R_2 if the former is an unbased mode.

(iii) Both R_1 and R_2 are composite representations. Since the gross set-theoretic types of VO_1 and VO_2 are assumed to be equal, the composite structures specified by R_1 and R_2 must also have the same gross type (i.e. both must be sets, or maps, or tuples, if one is). In this case merge the element-mode of R_1 with that of R_2 (if R_1 and R_2 represent maps, merge their domain element-modes and their range element-modes separately; if R_1 and R_2 represent tuples of the same known length n, each component then having its own type and representation, perform n componentwise merges).

This merging/equivalencing process can be made highly efficient by using a compressed-balanced tree representation for the set of all generated bases (cf. [TA]). This representation allows execution of a sequence of equivalencing operations in almost linear time. The element-mode map EM need be kept only for tree-roots (i.e. we only need to keep one value per equivalence class). Whenever two trees whose roots are B_1, B_2 are merged into one, we also update the map EM of the new root (which is either B_1 or B_2), as follows:

(i) If both $EM(B_1)$ and $EM(B_2)$ are unbased, they must be equal, so that no updating need take place.

(ii) If one of them is based and the other is not, set EM of the new root to the based mode.

(iii) If both are based, then leave either of the two basings at the new root, but merge them with one another.

Note that phase (b) uses the neutral bases introduced in phase (a) to transmit basing information between instructions without actually having to use any global propagation technique. This point is illustrated by the following example:

```
(1)   S with x;        $ S is a set
(2)   V(i) := S;       $ V is a tuple
                         (of sets)
(3)   T := V(j);
(4)   y from T;        $ T is a set
```

Phase (a) will have generated the following provisional representations:

S_1: set ($\in B_1$); x_1: $\in B_1$
V_2: tuple ($\in B_2$); S_2: $\in B_2$
V_3: tuple ($\in B_3$); T_3: $\in B_3$
T_4: set ($\in B_4$); y_4: $\in B_4$

Using the S_1- S_2 link, phase (b) will set $EM(B_2)$ = set ($\in B_1$). In virtue of the V_2- V_3 link we then equivalence B_2 and B_3, setting EM of the new root to 'set ($\in B_1$)'. Then, in virtue of the T_3- T_4 link, we merge $EM(B_3 \approx B_2)$ with 'set ($\in B_4$)', i.e. merge 'set ($\in B_1$)' and 'set ($\in B_4$)' which causes B_1 and B_4 to be equivalenced. If B_2 and B_3 were unavailable, this deduction would have been more difficult, and rather complex propagation of basing information through the code would have been required.

(c) base pruning and representation adjustment: When phase (b) terminates, the set of all initial bases will have been split into equivalence classes. Each such class corresponds to one actual base B, and the map EM maps the root of

this class to the common representation of the elements of B. However, it is possible that such a base B may be useless, in the sense that its introduction cannot speed up the execution of any instruction, or, even if some instructions are made more efficient due to the introduction of B, all these instructions involve only the same composite object s based on B. In this latter case, introduction of B will simply replace the hash table of s by that of B, which gains us nothing. Such cases are detected and eliminated by phase (c), as follows:

(i) We find all actual bases which are useless according to the criterion stated above, and flag them as such.

(ii) We update all based representations of variable occurrences and also the element modes of all actual bases, in the following recursive manner:

(ii.1) unbased modes are left unchanged;

(ii.2) each provisional base B appearing in a mode is replaced by the corresponding actual base $\hat{B}$ if $\hat{B}$ is not useless. If $\hat{B}$ is useless, we replace the submode $\in B$ by $EM(\hat{B})$.

(iii) We enter all useful bases into the symbol table.

It can be shown that the preliminary type-analysis phase of the optimizer can be adjusted in such a way as to guarantee that this recursive adjustment operation will always converge. Note also that it is only after this adjustment that a variable occurrence can have a based representation involving more than one level of specification, e.g.
set (tuple ($\in$ B)).

As an example, consider the case of an integer-valued bivariate map f. In compilation of SETL, each retrieval of the value f(x,y) is expanded into the code sequence

```
(1)        t := f{x};
(2)        z := t(y);
```

Phase (a) of our algorithm will generate the following provisional representations:

$t_1 : \in B_2$; $f_1 :$ map ($\in B_1$) $\in B_2$;

$t_2 :$ map ($\in B_3$) $\in B_4$;

Now we assume that (the equivalence class of) B_2 turns out to be useless (which is the case, e.g., if f is always accessed as a bivariate map) and also that B_4 is useless (e.g. no set manipulation of the range of f occurs in the program being analyzed). Phase (b) will merge the representations of t_1 and t_2 to obtain $EM(B_2) =$ map ($\in B_3$) $\in B_4$. Hence phase (c) will update the representation of f_1 to 'map ($\in B_1$) map ($\in B_3$) $\in B_4$' and again to 'map ($\in B_1$) map ($\in B_3$) int', which is probably the representation that a programmer would have chosen manually, and is also the best representation for sparse multivariate maps defined on arbitrary domains.

(d) Name-splitting: This is the final phase of our data-structuring algorithm. At the end of phase (c), based representations, as well as type information, will have been computed for each variable occurrence, rather than for each variable. We use occurrences rather than variables because the weak typing of SETL allows a variable to assume more than one data-type during execution; moreover, even objects with the same data-type may be represented in different ways at difference occurrences of the same variable. Nevertheless the information collected by our algorithm must finally be stated on a per variable basis, since subsequent compiler phases (e.g. our machine-code generator) cannot support more than one type or detailed representation per variable. Furthermore, the first three phases of our algorithm ignore the operations which actually insert elements into a base B. For example, consider the code

```
(1)        x := x + 1;
(2)        s with x;
```

The first three phases of our algorithm will assign 'int' as the representation of x_1, and '$\in$ B' as the representation of x_2,

where B is some base. This means that instruction (2) can assume that the value of x is represented by a pointer into B, but of course such a pointer must be first created.

These two problems of information integration and base insertion are solved simultaneously in the name-splitting phase of our algorithm. In this phase we first scan all occurrences of each program variable x. For each representation R assigned to at least one of these occurrences, we generate a new symbol-table entry, which we denote by x_R, and which is said to be split from x. We then replace all occurrences of x having the representation R by occurrences of x_R. If two occurrences of the same original variable x having different representations R_1, R_2 are linked by BFROM, then we must of course ensure that conversion of the value of x from representation R_1 to representation R_2 will take place as control advances from the first occurrence to the second one. To enforce this conversion, we insert an assignment '$x_{R_2} := x_{R_1}$' into the code at some optimal (lowest-frequency) place separating these two occurrences. The algorithm which inserts such conversions into the code is rather simple and is based on the interval structure [AL] of the flow graph of a SETL program. For each occurrence VO of a variable V which is linked by BFROM to other occurrences of V which have different representations, our algorithm will insert a conversion to the split variable of VO either just before VO, or just before the head of some interval containing VO. For example, in the code fragment

```
(∀ ...)
    x := x + 1;
end ∀;
    :
(∀ ...)
    s with x;
end ∀;
```

A conversion from the unbased representation appropriate for x within the first loop to the based representation appropriate for x within the second loop will be inserted just before the start of the second loop. The choice of the interval at whose head we insert a conversion involves safety criteria which will not be mentioned here. For a full description of this aspect of our method, the reader is referred to [GS].

It is important to note that the set of all program points at which there occur conversions to representations based on a base B is also the set of all points at which elements are added to B (by hashing operations). In general the number of these conversions will be smaller (and for typical programs, substantially smaller) than the number of hashing operations which would have been required without the presence of B.

A comment on the complexity of our algorithm: Phases (a) and (c) are linear in the length n of the code. Phase (b) is almost linear in the number m of BFROM links, which for typical programs will be linear in n. Phase (d) is $O(m + n)$. Thus the overall complexity of our algorithm is of order $O(n + m\alpha(m))$, where α is an extremely slowly growing function (cf. [TA] for details).

An Example: We will now indicate the way in which our algorithm applies to the MINPATH program given earlier. For simplicity, we consider only the first section of the MINPATH program, which searches through the graph within which a path is sought.

Phase (a) will generate the following provisional based representations:

$graph_1$, $cost_1$, x_1, y_1: general;
$prev_2$: map ($\in B_{21}$) $\in B_{22}$;
val_3: map ($\in B_{31}$) $\in B_{32}$;
val_4: map ($\in B_{41}$) $\in B_{42}$; x_4: $\in B_{41}$;
$newnodes_5$: set ($\in B_5$); x_5: $\in B_5$;
$newnodes_6$: set ($\in B_6$);
$newnodes_7$: set ($\in B_7$); n_7: $\in B_7$;
$graph_8$: map ($\in B_{81}$) $\in B_{82}$;
n_8: $\in B_{81}$; m_8: $\in B_{82}$;

Instruction 9 of the MINPATH code expands into roughly the following sequence

(91) tcost := cost{n};
(92) vcost := tcost(m);
(93) vval := val(n);
(94) newval := vval + vcost;

and thus generates the following provisional representations:

$cost_{91}$: map ($\in B_{911}$) $\in B_{912}$;
n_{91}: $\in B_{911}$; $tcost_{91}$: $\in B_{912}$;
$tcost_{92}$: map ($\in B_{921}$) $\in B_{922}$;
m_{92}: $\in B_{921}$; $vcost_{92}$: $\in B_{922}$;
val_{93}: map ($\in B_{931}$) $\in B_{932}$;
n_{93}: $\in B_{931}$; $vval_{93}$: $\in B_{932}$;

no bases are generated by instruction 94.

val_{10}: map ($\in B_{101}$) $\in B_{102}$;
m_{10}: $\in B_{101}$; $newval_{10}$: int;
val_{11}: map ($\in B_{111}$) $\in B_{112}$;
m_{11}: $\in B_{111}$; $newval_{11}$: $\in B_{112}$;
$prev_{12}$: map ($\in B_{121}$) $\in B_{122}$;
m_{12}: $\in B_{121}$; n_{12}: $\in B_{122}$;
m_{13},y_{13}: $\in B_{13}$;
$newnodes_{14}$: set ($\in B_{14}$); m_{14}: $\in B_{14}$;

Among the above bases, only B_{41}, B_5, B_{81}, B_{911}, B_{921}, B_{931}, B_{101}, B_{111}, B_{121} and B_{14} are effective.

Phase (b) will perform the following merging and equivalencing: In view of the $prev_2$-$prev_{12}$ link, we equivalence B_{21} with B_{121} and B_{22} with B_{122}. In view of the val_3-val_4 link, we equivalence B_{31} with B_{41}, and B_{32} with B_{42}. In view of the x_4-x_5 link, B_{41} and B_5 are equivalenced, etc. Only one additional merge deserves extra comment, namely the step which merges the representation of $tcost_{91}$ and $tcost_{92}$, and which belongs to case (ii) of the general description of merging given above. This merge step causes us to set $EM(B_{912})$ = map ($\in B_{921}$) $\in B_{922}$. The information thereby generated will be used by phase (c) to update representations containing B_{912}, which will turn out to be useless.

At the end of phase (b), the following equivalence classes of bases will have been formed:

$\hat{B}_1 = \{B_{912}\}$
$\hat{B}_2 = \{B_{922}\}$
$\hat{B}_3 = \{B_{32}, B_{42}, B_{932}, B_{112}\}$
$\hat{B}_4$ = a class containing all the remaining bases.

Since only $\hat{B}_4$ is effective, the first three classes are useless. Phase (c) will thus eliminate any reference to bases in these classes, e.g. the representation of $cost_{91}$ will be updated to

map ($\in \hat{B}_4$) map ($\in \hat{B}_4$) int

As for $\hat{B}_3$ we simply replace the submode '$\in \hat{B}_3$' by 'int', since, as is easily checked, $EM(\hat{B}_3)$ = int at the end of phase (b).

Phase (c) will thus assign the representations suggested in section 2 to most of the occurrences of the variables graph, cost, newnodes, prev, val, x, y, m and n. However, the occurrences $graph_2$, $cost_2$, x_2 and y_2 remain unbased. This will cause phase (d) of our algorithm to split each of these variables into two symbol-table entries, a based one and an unbased one, and to insert four conversions from each of the unbased split variables to the corresponding based one before entering the while loop (but the conversion of x will precede instruction 4, as x_4 has already the based representation). The base $\hat{B}_4$ is built precisely at these places, and then remains constant during execution of the remainder of the MINPATH program.

A few final words concerning refinement of the coarse based representations selected by our algorithm: A main goal of our algorithm is to speed up program execution by reducing the number of hashing

operations performed by the program. Having achieved this optimization we can expect a considerable speed-up in the execution of the program. However additional improvement is possible if we refine the based representations thereby selected, choosing suitable local, indexed or sparse representations for based sets and maps. However, to choose refined data-structures effectively may require frequency information, object size estimates, etc., which are unavailable during compile time (but cf. [LO] for various interactive and run-time techniques which can help gather such information). Since our aim is to develop a fully automatic data-structuring technique, we abandon any attempt to collect and use such information, and make do with a coarser and more modest approach, which can be summarized roughly as follows:

(i) If a based object A is iterated upon, choose the sparse representation for A.

(ii) If not, but A is involved in global set-theoretic operations, or is assigned to any variable, or passed as a procedure parameter, or incorporated into other objects or used destructively in a manner requiring value copying, choose the indexed representation for A.

(iii) In all other cases, A can have local representation.

References

[AL] Allen, F. E., "Control flow analysis," Proc. Symp. Compiler Optimization, SIGPLAN Notices 5 (1970), 1-19.

[DE] Dewar, R.B.K., Grand, A., Liu, S.C., Schonberg, E. and Schwartz, J.T., "Programming by refinement, as exemplified by the SETL representation sublanguage," to appear in CACM.

[GS] Grand, A. and Sharir, M., "On name splitting in SETL optimization," SETL Newsletter 206, Courant Inst. Math. Sci., New York, 1978.

[HA] Halstead, M. H., "Elements of software science," Elsevier North-Holland, New York, 1977.

[LI] Liu, S. C., "Automatic data-structure choice in SETL," Ph.D. thesis, Courant Inst. Math. Sci., New York, 1978 (to appear).

[LO] Low, J. R., "Automatic data-structure selection: an example and overview," CACM 21 (1978), 376-384.

[RE] Reif, J. H., Ph.D. Thesis, Harvard University, (to appear).

[SL] Schonberg, E. and Liu, S. C., "Manual and automatic data-structuring in SETL," Proc. 5th Annual III Conference, Guidel, France, 1977, 284-304.

[SC] Schwartz, J. T., "Automatic data-structure choice in a language of very high level," Proc. 2nd POPL Conference, Palo Alto, Calif., 1975, 36-40.

[SC_2] Schwartz, J. T., "Use-use chaining as a technique in typefinding," SETL Newsletter 140, Courant Inst. Math. Sci., New York, 1974.

[SC_3] Schwartz, J. T., "On Programming: an interim report on the SETL project," 2nd edition, Courant Inst. Math. Sci., New York, 1975.

[TA] Tarjan, R. E., "Efficiency of a good but not linear set-union algorithm," JACM 22 (1975), 215-225.

[TA_2] Tarjan, R. E., "Solving path problems on directed graphs," STAN-CS-75-512 Tech. Rep., Stanford University, Calif., 1975.

[TE] Tenenbaum, A. M., "Type determination for very high level languages," Computer Sci. Rep. 3, Courant Inst. Math. Sci., New York, 1974.