Experience with the SETL Optimizer

STEFAN M. FREUDENBERGER, JACOB T. SCHWARTZ,
and MICHA SHARIR
New York University

The structure of an existing optimizer for the very high-level, set theoretically oriented programming language SETL is described, and its capabilities are illustrated. The use of novel techniques (supported by state-of-the-art interprocedural program analysis methods) enables the optimizer to accomplish various sophisticated optimizations, the most significant of which are the automatic selection of data representations and the systematic elimination of superfluous copying operations. These techniques allow quite sophisticated data-structure choices to be made automatically.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*very high-level languages*; *SETL*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; I.2.2 [**Artificial Intelligence**]: Automatic Programming—*automatic analysis of algorithms*; *program modification*; *program transformation*

General Terms: Languages

Additional Key Words and Phrases: Automatic data-structure selection, copy optimization

1. INTRODUCTION

The programming language SETL [7] is a very high-level language whose syntax and semantics are based on the standard set theoretical dictions of mathematics. It is designed to allow complex algorithms and large-scale programs to be written succinctly and rapidly, in a form closely resembling their abstract mathematical statement. The labor of converting one's initial abstract ideas into a runnable program is thereby significantly shortened.

For this paradigm to be successful, the language has to provide high-level abstract objects and operations between them, include high-level control structures, and eliminate part of the need for data-structure selection and manipulation.

The gains in ease of software development that the use of such a very high-level language yields are often offset, at least in part, by the modest execution efficiency of programs written in the language. Both the time and space efficiency

Work on this paper has been supported in part by National Science Foundation grant MCS-76-0116 and by U. S. Department of Energy Office of Energy Research contract EY-76-C-02-3077.
Authors' present addresses: S. M. Freudenberger and J. T. Schwartz, Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, NY 10012; M. Sharir, Department of Mathematics, Tel-Aviv University, Ramat-Aviv, Tel Aviv 69978, Israel.

of high-level abstract programs tend to be less than those of their low-level counterparts. However, one can aim to remove part of this inefficiency by optimization of the program, whereby the level of the program can be "lowered" automatically. In this paper we describe the basic structure of such an optimizer, which has been developed for the SETL language. We explain the principal optimizations that it performs, comment on some of the algorithms used in it which have not been previously reported, and report on the way it performs when applied to a variety of SETL programs.

This paper is written as a sequel to two preceding papers, [5] and [15]. The first of these papers describes a system for improvement of SETL programs by manual selection of proper data representations for the program objects; the second outlines an automatic technique for the selection of such representations by an optimizer. The techniques described in [15] have in fact been implemented in the present SETL optimizer and form the basis of its most significant optimization capabilities. In this paper we review these techniques briefly, describing their current form, and also describe various new optimizations, the most notable among which is elimination and optimization of value copying operations. We also describe and summarize the overall structure of the optimizer, exhibit some of the optimizations which it is able to perform, and comment on its future potential.

This paper is organized as follows: Section 2 contains a very short review of the major features of SETL and lists the major optimizations that one needs to apply to SETL programs. Section 3 describes the overall structure of the implemented optimizer and comments on some of the main algorithms it uses. Section 4 shows the optimizer at work on a series of medium-size SETL programs. A summary of our experience and remarks concerning future possibilities are found in the concluding Section 5.

2. MAJOR FEATURES OF SETL

SETL has been described in a number of recent papers [5, 6, 15]. Here we give only a very short survey of the basic features of the language and those aspects of its implementation needed to understand the way in which one wants to optimize programs written in it. SETL is a procedural language whose elementary data types are essentially those primitive types familiar from other languages, for example, PASCAL (integers, Booleans, strings, and "atoms," for an explanation of which see [5]) and whose composite types are **set**, **map**, and **tuple**. In addition, SETL provides an "undefined value" omega, denoted **om**, which can appear only in equality tests and as the right-hand side of an assignment.

Tuples are dynamically extensible vectors of arbitrary components, to which one can apply the following operations: indexing, iteration, concatenation, subtuple extraction and modification, cardinality, insertion and deletion of elements at the ends, etc. The components of a tuple are stored at contiguous locations within a dynamically managed memory area, so that the area storing a tuple may occasionally have to be reallocated.

Sets and maps are the most important data types of the SETL language. Their semantic properties are those that mathematics specifies for finite sets. The following set operations are provided: insertion and deletion of elements, mem-

bership test, iteration, set former, cardinality, set union, intersection and difference, etc. Maps are sets all of whose elements are length-two tuples (called pairs) and represent set theoretic relations and functions. All set operations apply to maps, but the following operations are also provided: indexing a map by any domain value, updating a map entry, domain and range calculation, and iteration over a map domain. Maps can be either single valued (functions, called smaps) or multivalued (relations, called mmaps). In the absence of optimization or of any data-structure declaration (for which see [5]), sets are implemented as linked hash tables, which will occasionally expand or contract as the size of the set varies; maps are represented in a similar fashion, except that the elements of maps are hashed on their first component so as to support indexed map operations efficiently.

SETL provides the conventional control structures similar to those of PASCAL but also supports certain high-level iterative constructs, including set and tuple formers, and existential and universal quantification over composite objects. The language is weakly typed and requires no variable declarations. The (dynamic) type of a variable is simply that of the last value assigned to it. Most operators are overloaded, that is, can be applied to several different data types. It is also significant that SETL is a "value" rather than a "pointer" language, which means that composite objects often have to be copied when their value is being modified.

As an example illustrating the use of SETL, consider the program shown in Figure 1, which is one of the examples whose optimization is described in Section 4. The program computes the frequency with which individual characters appear in a given string and uses these frequencies to assign Huffman codes (see [13, p. 402]) to the characters. (Line numbers and statement numbers are included for later reference.)

2.1 Potential Improvements of SETL Programs

As can be inferred from the example of Figure 1, unoptimized SETL programs can be very inefficient. The following list describes the most significant ways in which SETL programs can be optimized.

(1) Since without optimization the types of variables are not known at compile time, and since most of the SETL operators are overloaded, the code generated by the compiler will involve many type checks. Instructions involving such checks must therefore be executed by an off-line call to a fairly complex run-time library routine. Compile-time determination of accurate variable types can improve program efficiency both by avoiding type checks and by eliminating the need for library calls in those particular favorable cases in which it is appropriate to execute a (relatively simple, type-specific) operation in-line. For example, the "+" operator applies to integers (addition), to strings and tuples (concatenation), and to sets and maps (union). If the compiler can determine that a certain "+" operation will only be applied to integers, it can emit in-line code for it, thus avoiding a much more expensive call to a general library *ADD* routine.

(2) The default representation of the various sets and maps appearing in the program of Figure 1 uses a collection of linked hash tables. Consequently, retrieval and updating of map entries, and also insertions, deletions, and membership tests on sets, etc., are performed by hashing, which is a time-consuming operation.

```
 1  1  program HUFFCODE;
 2  2
 3  2  $ A test program for the optimizer. It computes the Huffman Code of individual
 4  2  $ characters in a given string.
 5  2
 6  2    var S; repr S: string; end repr;
 7  5
 8  5    read(S); print(S);
 9  7
10  7    $ Compute the FREQ map that counts the frequency of each character in S.
11  7
12  7    FREQ := { };
13  8    CHARS := { };      $ set of all characters appearing in S
14  9    (∀C ∈ S)           $ for all characters in the input string
15 10      if C ∉ CHARS then                $ generate a new leaf:
16 11        CHARS := CHARS with C;         $ – add C to CHARS
17 12        FREQ(C) := 1;                  $ – init. frequency count
18 13      else
19 13        FREQ(C) := FREQ(C) + 1;        $ increment frequency count
20 14      end if;
21 15    end ∀;
22 16
23 16    $ Next build up the Huffman tree by repeatedly connecting two nodes having
24 16    $ minimum frequencies to a new parent node.
25 16
26 16    HTREE := { };      $ parent mapping in the Huffman tree
27 17
28 17    (while #FREQ > 1)
29 18      $ Find the two nodes with minimum frequency.
30 18      [C1, F1] := GETMIN(FREQ);
31 19      [C2, F2] := GETMIN(FREQ);
32 20
33 20      $ Generate a new tree node C, identified by the concatenation of the
34 20      $ strings identifying the children.
35 20      FREQ(C := C1 + C2) := F1 + F2;
36 21
37 21      $ Note the tree connection of C1 and C2 to C.
38 21      HTREE(C1) := [C, "0"];     $ C1 is a "left" child.
39 22      HTREE(C2) := [C, "1"];     $ C2 is a "right" child.
40 23    end while;
41 24
```

```
42 24    $ Finally, compute the actual Huffman codes in a map HCODE that maps
43 24    $ each character to its binary code, decoded as a string. (The actual encoding
44 24    $ of S is not shown.)
45 24    HCODE := { };
46 25    (∀C ∈ CHARS)
47 26      HCD := " ";
48 27      B := C;
49 28      $ Go up the tree from C, collecting all edge marks in the string HCD.
50 28
51 28      (while HTREE(B) /= om)
52 29        [B, LR] := HTREE(B);     $ LR is left/right child.
53 30        HCD := LR + HCD;
54 31      end while;
55 32
56 32      $ Put the final code in HCODE(C).
57 32      HCODE(C) := HCD;
58 33    end ∀;
59 34
60 34    print("Huffman Code is ", HCODE);
61 35
62 35
63  1    procedure GETMIN(rw F);     $ Get and remove a minimum-frequency
64  2                                $ element from F. (Note that F is a
65  2                                $ read–write parameter.)
66  2
67  2      C := om; N := om;
68  4      (∀M = F(A))
69  5        if N = om or M < N then
70  6          [C, N] := [A, M];
71  7        end if;
72  8      end ∀;
73  9
74  9      F(C) := om;     $ Remove C from the domain of F.
75 10
76 10      return [C, N];
77 11
78 11    end procedure GETMIN;
79 12
80 12  end program HUFFCODE;
```

Figure 1

Moreover, the space efficiency of the default representation is low, since it maintains multiple hash tables whose relationship to each other could in fact be used to compress them all. Indeed, a careful examination of the *HUFFCODE* program reveals that there is essentially one "universal domain" accessed by the program objects, namely, the set of characters appearing in S and the set of strings formed by concatenating these characters; these strings are used to represent nodes in the Huffman tree. Consequently, if one arranges to maintain only one hash table representing this universal domain and then represents all other program objects by fields within entries of this table, or by pointers to the table entries, or by arrays indexed by such table entries, it becomes possible both to eliminate many hashing operations and to obtain a more space-efficient representation.

These improvements can in fact be imposed manually by using the declarative SETL data-representation sublanguage, described in [5], to tell the SETL compiler what data structures are desired. To do this we add declarations to the program shown above. These declarations specify the representation desired for particular variables. (Note, however, that this will normally be done only after a program has been debugged in its initial undeclared form. Moreover, once debugged, "undeclared" program text will generally require little or no change other than the addition of these declarations.) The central notion in the SETL representation scheme is that of a **base**. As explained more fully in [5], each auxiliary identifier declared to be a **base** constitutes a "universal domain" for certain of the values accessed by the program. Bases are ordinarily maintained as hash tables. However, each base entry is represented by a block which, in addition to the value of the object x it represents, holds additional fields that can be used to store other values related to x, for example, the values of maps $f(x)$, $g(x)$, etc., or the Boolean values of set membership tests "$x \in A$," "$x \in B$," etc. To be more specific, once having declared B to be a base, we can then use it to make the following typical declarations:

A program variable x can be declared to be an "element of B" (which means a pointer to an element of B).

A set variable S can be declared as "set of elements of B." In this case S is represented either by a collection of attribute bits, one attached to each element of B, indicating membership in S (this happens if S is declared to be a **local** set of elements of B), or as a vector of bits which can be indexed by indices stored at the elements of B (this happens if S is declared to be **remote**[1]), or as a separate hash table, the value of whose entries are pointers into B (if S is declared to be **sparse**). The first two representations eliminate the need for hashing, the second representation is ideal for global set operations, and the third representation is useful when the set is sparse relative to B and when it is used mainly in nonhashing operations, such as iterations.

A map variable whose domain is a subset of B can be declared to be a "map from elements of B," written as **map (elmt** B**)** *, where by * we mean to denote any descriptor specifying the form of the range values of F. If declared in this way, F is represented either by a field attached to each element x of B, holding

[1] **remote** has replaced **indexed** used in earlier papers.

```
Variables for program HUFFCODE
base OPT≠1021:   string;
GETMIN:          procedure(remote smap (elmt OPT≠1021) integer)
                   tuple (elmt OPT≠1021, integer);
S:               string;
Variables for procedure __MAIN
B:       elmt OPT≠1021;
C:       string;
C.1:     elmt OPT≠1021;
C1:      elmt OPT≠1021;
C2:      elmt OPT≠1021;
CHARS:   remote set (elmt OPT≠1021);
F1:      integer;
F2:      integer;
FREQ:    remote smap (elmt OPT≠1021) integer;
HCD:     string;
HCODE:   local smap (elmt OPT≠1021) string;
HTREE:   local smap (elmt OPT≠1021)
           tuple (elmt OPT≠1021, string);
LR:      string;
Variables for procedure GETMIN
A:       elmt OPT≠1021;
C:       elmt OPT≠1021;
F:       remote smap (elmt OPT≠1021) integer;
M:       integer;
N:       integer;
```

Figure 2

the value $F(x)$ (if F is **local**); or by an array of such fields, indexed in the same way as described above (if F is **remote**); or by a separate hash table containing pointers to element blocks of B (if F is **sparse**). We stress again that use of either of the first two representations eliminates many hashing operations involving F.

The reader is referred to [5] and [15] for a full description of these features. Note, however, that this economical library of representations is capable of realizing most of the standard data structures available in low-level languages, including arrays, lists, records, pointers, and tables.

Returning to the *HUFFCODE* program given above, we note that Figure 2 declares a plausible family of data representations that could be used for it. (In fact, these declarations were produced automatically by our optimizer, using algorithms to be reported in the remainder of this article.)

The improvements called out by declarations of this kind can speed up the execution time of SETL programs by a factor ranging between two and ten.

(3) Another significant way of improving a SETL program is to eliminate unnecessary copying operations. As noted earlier, SETL is a "value"-based language. However, for efficiency the current implementation transfers pointers rather than values whenever possible. Nevertheless, to preserve the value semantics of the language, objects may have to be copied when their values are modified, unless an optimizer can ascertain that no other currently live variable shares the

same value. The unoptimized current implementation of SETL uses a rudimentary form of reference counts, called "share bits," to record value sharing. Specifically, each "long" value (e.g., tuple or set value) contains a bit which, if set, indicates that the value may be shared currently. Any modification of a value that might be shared will cause the value to be copied, since other values may contain pointers to the (old) value. This scheme generally performs well, but it has significant shortcomings. For one, once being shared, a value cannot become unshared until it is copied. (Maintaining reference counts rather than simple share bits would make such "unsharing" possible but would probably be too expensive to be worthwhile.) Moreover, if a value is shared only by variables that are dead at a destructive use of that value, it can safely be modified with no need to copy, but our simple "share bit" scheme cannot detect this possibility (even full reference counts could not). Clearly, then, improved ability to eliminate unnecessary copying operations can result in significant program performance improvements. Other less significant but still useful optimizations also become applicable if copy elimination is carried out; for example, the required level of share-bit manipulation can be reduced.

To illustrate the potential benefits of copy optimization, we take the *HUFFCODE* program (Figure 1) as an example. The parameter *F* of the *GETMIN* routine is a read–write parameter and is modified at line 74 by the statement *F*(*C*) := **om**. At the point of execution of this statement, the value of *F* has already been shared by the actual argument *FREQ*, and so the unoptimized version of this program will copy *F* each time *GETMIN* is executed. However, this is not really necessary, because the value of *FREQ* is dead at this point, since the value *F* computed by the *GETMIN* routine will be assigned to *FREQ* when *GETMIN* returns to the main program. This fact cannot be detected dynamically, even by a full reference-count scheme, but can be detected at compile time by an interprocedural live–dead analysis and in fact is detected by our optimizer.

(4) In addition to these major improvements, classical "low-level" optimizations also apply to SETL programs. These optimizations include common-subexpression elimination, motion of code out of loops, dead-code elimination, and constant folding (see [2, 8] for description of these optimizations). While all these are useful, the advantages gained by these transformations are usually smaller than those gained by the higher level optimizations described above. Note also that certain classical optimizations, such as register allocation, which apply at machine-code level are not applied in the current SETL system.

3. STRUCTURE OF THE SETL OPTIMIZER

In this section we describe the structure of our optimizer and explain the major algorithms that it uses. First, we comment briefly on its relation to other components of the SETL system.

The SETL compiler/interpreter consists of five main parts:

(1) the parser, which parses a given source program and translates it into a reverse Polish text;
(2) the semantic analyzer, which performs a semantic analysis of the program and transforms it into an intermediate-level code, called Q_1 code, similar to

the quadruples code described in [3] and [8] but also including a symbol table and several auxiliary tables;

(3) the optimizer, which analyzes the Q_1 code supplied by the semantic pass and improves it according to the goals outlined in the previous section. Its output is also Q_1 code; this allows the system to skip optimization if desired, and go directly to

(4) the code generator, which transforms the Q_1 code supplied either by the semantic pass or by the optimizer into a low-level interpretable code, known as Q_2 code;

(5) the interpreter/run-time library, which executes the Q_2 code produced by the compiler. (This can be done either interpretively or by producing executable machine code. If machine code is produced, we also eliminate library calls when in-line code suffices to perform a required operation.)

3.1 Overall Structure of the SETL Optimizer

The current version of the optimizer consists of several successive analysis and optimization phases which all process the whole Q_1 text representing the program to be optimized. (In what follows we assume familiarity with the basic terminology concerning program flow analysis, which can be found in [2, 8, 14, 19].) These phases are as follows:

(1) *Initial Preparatory Pass.* In this pass over the program code, basic information about the program is collected, and the code is slightly modified to meet requirements assumed by some later phases of the optimizer. This pass constructs several maps holding information about program variables and their occurrences in the program; identifies procedure calls and constructs a program call-graph; builds a flow-graph for each procedure in the program; and renames temporaries for later common-subexpression elimination.

(2) *Call-Graph Analysis.* This phase decomposes the program call-graph into its strongly connected components (recursive cycles) and orders them in a topologically sorted order, which is used later during interprocedural data flow analysis (see also [19]).

(3) *Interval Analysis.* In this phase each procedure flow-graph is analyzed, and its interval structure is computed. The algorithm that we use is based upon Tarjan's fast graph-irreducibility testing algorithm [20], modified in certain ways to handle irreducibilities in a simple manner and to prepare for later code motion. (See [19] for additional details.)

(4) *Common-Subexpression Elimination and Code Motion.* This phase performs available-expression analysis, including code motion out of loops (intervals), on the program flow-graph, using an interprocedural bit-vectoring data flow analysis technique described in [19]. (The optimizer contains a general-purpose bit-vectoring data flow analysis package that is used by several of the optimizer phases. This package supports "forward" and "backward" analysis, both intraprocedural and interprocedural, and optionally will perform code motion as an integral part of an analysis. This package and the algorithms that it uses are fully described in [19].) On the basis of the results of this analysis, redundant computations of common subexpressions are eliminated, and computations of expres-

sions that can be moved out of intervals are inserted into special "preheader" blocks preceding the entry points of these intervals.

(5) *Computation of Modified Use-Definition Links.* This phase performs a modified version of the reaching definition analysis described in [2, 8, 12], which we call "reaching occurrences" analysis. Here, for each program point n, we wish to determine the set of all variable occurrences *VO* that can reach n along a path free of any other occurrence of the variable appearing at *VO*. On the basis of the results of this analysis, the optimizer computes a modified version of the use-definition chaining map [2], which we term "use-use chaining" [18], in which each use of a variable is linked to all the nearest preceding occurrences (definitions and uses) of the same variable.

(6) *Type Analysis.* This phase estimates the types of variables occurring in the program, using Tenenbaum's type analysis method [21]. Types are computed by forward and backward propagation between program points linked by the use-use links computed in the preceding phase. Note that, since SETL is weakly typed, types are most appropriately associated with variable occurrences rather than with variables. (See also [9, 11].)

(7) *Automatic Selection of Data Representation.* This central phase generates useful bases for the program being optimized and selects data representations for the program objects in accordance with the goals described in Section 2. The algorithm that it uses is described fully in [15] and is also reviewed below.

(8) *Conversion Optimization.* In this phase the results of the preceding analyses are entered into the symbol table. Specifically, the set of occurrences of each variable v appearing in the program is partitioned into classes, each of which contains all occurrences of v that have been assigned the same data type and representation by the preceding steps. For each such class a special variable $v.j$ (called a "split variable") is generated and is given the data type and representation associated with that class. The symbol table is then updated as follows: Newly generated bases are entered into the table; new split variables are added to the table; the data type and representation of each variable are updated. In addition, explicit conversions between different variables split from the same variable are inserted at low-frequency code points separating program regions in which one split variable is being used from other regions in which different variables $v.j$ split from the same variable v are used. Program points at which interrepresentation conversions are required are detected using several bit-vectoring analyses which detect "availability" and "liveness" of split variables (see [15], and also below, for a description of these methods).

(9) *Copy Optimization.* This phase eliminates unnecessary copy operations from the code being optimized, and it also eliminates share-bit manipulating operations where possible. The algorithm used is an improved version of Schwartz's value-flow technique [17], which also uses bit-vectoring analyses to find reaching definitions and to determine variable liveness. A more detailed description of these algorithms is given below. By exploiting liveness information we obtain a powerful copy-elimination technique capable of eliminating most of the unnecessary copy operations in the program.

(10) Finally, the improved Q_1 code is written out.

Having thus sketched the overall structure of our optimizer, we proceed to give more detailed descriptions of some of the techniques it uses.

3.2 Automatic Selection of Data Representation

The algorithms used by the optimizer for automatic selection of data representations have been reported in [15]. The current implementation of this algorithm differs only slightly from the description given there. However, for the sake of completeness we include a brief sketch of our method here. The reader is referred to [15] for a full description.

The selection algorithm is composed of four subphases:

(1) *Base Generation Phase.* In this subphase provisional bases and based representations are generated for each instruction I for which the introduction of such representations will not slow the execution of I. (The execution speed of I either may remain essentially the same, in which case the bases introduced are called "neutral," or it may speed up considerably due to the elimination of a hashing operation, in which case the relevant bases are called "effective.") Initially, these provisional representations do not interact across instructions.

(2) *Representation Merging and Base Equivalencing.* In this phase the provisional representations generated by the preceding phase are merged together. Specifically, if two occurrences VO_1 and VO_2 of the same variable are linked via a use-use link, and if both VO_1 and VO_2 are found to have the same type, and both have received tentative based representations, then these representations are merged. This will relate the bases appearing in these representations to each other, either because they are equivalenced directly to each other, or recursively by relating the structure of the elements of one base to that of the elements of the other. (E.g., we may find that the elements of one base are also sets of elements of another base.) This phase uses the Hopcroft-Tarjan equivalencing scheme (see [1]) and in practice attains an almost linear time performance. It should also be noted that this merging scheme, which involves only local merging of representations, achieves a global propagation of basings throughout the program.

(3) *Base Pruning and Representation Adjustment.* In this subphase the final representation of each variable occurrence is computed as follows. Each equivalence class of bases is represented by some base B in it. If such a class contains at least one "effective" base (see [15]), then B is regarded as being effective and will eventually be inserted into the symbol table as an "actual base." Each representation referring to a base B' belonging to this equivalence class is adjusted by replacing B' by B. All equivalence classes not containing any effective base are discarded as useless.

(4) *Conversion Analysis.* This final step, which constitutes a separate phase in our optimizer, can be regarded as a cleanup necessarily following the automatic representation-selection procedure. In this phase actual bases are entered into the symbol table, and variables are split and given detailed representations in a manner already described above.

3.3 Additional Details Concerning Copy Optimization

As previously noted, a significant optimization performed by our optimizer is the elimination of unnecessary copy operations from the code being analyzed. The majority of these copy operations are due to SETL's value semantics, but, as we have seen in the *HUFFCODE* example above, parameter aliasing is handled by

our algorithm as well. A copy operation is potentially required at a program point n if part of the value of a variable v is being modified. It can be eliminated if, when execution reaches n, each program object containing a pointer to the value of v is known to be dead at that point. An object is considered dead at a point n if either (1) no variable points to it directly or indirectly or else (2) along all possible execution paths from n each such variable is going to get a new value before being used.

Such situations are detected by the optimizer via a two-phase analysis. The first phase is known as value-flow analysis and is a simplified and improved version of the original value-flow technique described by Schwartz [17], incorporating improvements suggested by Tsui [22]. This phase performs separate copy-elimination analysis for each "destructive use" *DU* in the code being optimized, that is, each operation in a program which might modify part of a composite value. Normally, there are not too many of these. To be more specific, for each such destructive use *DU*, our analysis determines the set of all preceding variable occurrences *VO* whose value can contain a pointer to the value partially modified at *DU*. To find these occurrences, we employ a two-way propagation scheme of the following kind. The items propagated are pairs of the form [*VO*, *R*] where *VO* is a variable occurrence and where *R* can be any "value-containment relationship" holding between *DU* and *VO*. These symbols *R* represent the relationships being tracked as compositions of elementary relationships, each indicating one level of value containment. For example, the elementary relationship $VO_1 \in VO_2$ indicates that (the value of) VO_1 can be an element of (the composite object) VO_2.

During this phase items [*VO*, *R*] are first propagated from *DU* backward, along use-use links, and from output variables of instructions to their input variables; this finds all preceding occurrences *VO* for which the value *VAL* appearing at *DU* is obtained as a subvalue of the value at *VO* via a sequence of assignments, embeddings, and retrievals, all of which would be executed by the SETL runtime system using pointer transfers. However, these occurrences are not yet all those that might contain a pointer to *VAL*, since it is possible that another occurrence *VO'* which appears after *VO* but before *DU* in the execution flow should get its value from the value of *VO* through a similar sequence of assignments, embeddings, and retrievals, in which case *VO'* might also contain a pointer to *VAL*. To track down these additional relationships we perform a second value-relationship propagation, this time in the direction of execution flow, starting from all the pairs [*VO*, *R*] previously found. It can easily be seen that this finds all occurrences that can contain pointers to *VAL*. Note that, since copy optimization is performed after automatic selection of data representations, it is able to make use of relatively precise information concerning the representation and type of program objects in working out its value-containment estimates.

However, some degree of overestimation takes place, since it is possible that our second (forward) propagation step propagates value relationships relative to some destructive use *DU* from some variable occurrence *VO* to another occurrence *VO'* along a path which has not been traced in the first (backward) propagation step from *DU* to *VO*. When this happens, a spurious value relationship between *DU* and *VO'* might be established. Moreover, it may also happen that *VO'* lies further ahead in the execution flow than *DU*; however, this second

phenomenon would be detected by the following step of copy optimization, which we now proceed to describe.

In the second copy-optimization subphase, we perform live–dead analysis for all variable occurrences encountered during the preceding phase. An occurrence *VO* of a variable *V* is said to be "live" at a program point *n* if there exists a path from *VO* to a use of *V* which passes through *n* and which is free of any modification of *V*.

For simplicity, we break the analysis into two subphases: First, for each point *n* within a potentially destructive use *DU*, we compute the set of all relevant variable definitions which can reach *n* (i.e., those definitions bearing a value-containment relationship to the object being modified at *DU*). Then we compute the set of all variables *V* corresponding to the relevant definitions which are live at *n* (in the usual sense of liveness at a point). Combination of the results of these two analyses gives us the information we need. We caution that a slight overestimation may occur in the interprocedural case (in the analysis of global variables), since the existence of two interprocedurally valid subpaths (from an occurrence *VO* of a global variable *V* to a potentially destructive operation *n* and then from *n* to a use of *V*) does not necessarily imply that the concatenation of these two paths is also interprocedurally valid.

It should be noted that our copy-optimization algorithm solves an important problem that would have arisen if we were to use a simpler approach based on bit-vectoring data flow analysis. This is the issue of "globalization" of share-bit setting. Suppose that a variable which will later be used destructively is passed as a parameter to some procedure *P*. This value transfer already causes *V* to be shared with the corresponding formal parameter, and *V* may also become shared with other local variables of *P*. However, unless the value of *V* becomes part of a global object (or a write parameter, or a return value) during the execution of *P*, the call to *P* should not be viewed as sharing the value of *V* after the call has been completed. This fact, which is very difficult to pick up by using a bit-vectoring scheme, is handled implicitly by the live analysis technique we use.

Having collected this information, we can eliminate value copying at a destructive use *DU*, provided that no prior definition *VO* potentially containing a pointer to the value modified at *DU* is live at *DU*. This information also facilitates several other, though less significant, optimizations. For example, we can detect that certain copy operations are "unconditional," since the value to be copied is shared along all preceding execution paths, and so eliminate share-bit testing during the copy operation. Dually, we can eliminate certain share-bit setting operations, provided that the bits which they set are not going to be used subsequently in some "conditional" copy operation.

4. THE OPTIMIZER AT WORK

We now proceed to observe our optimizer at work on some examples. To this end, we first return to our previous example, the program to compute Huffman codes shown above.

For this program, type analysis will show that *CHARS* is a set and that *FREQ* is a map from strings to integers (this deduction depends on the assignment at line 17 and the addition appearing in line 19). Analysis of line 26 shows that *HTREE* is a set. Moreover, *C* is a string, as follows in view of the assignments to

C in lines 70 and 35. Thus we can conclude that *HTREE* is a map from strings to pairs of strings (see lines 38 and 39). (Our type analyzer regards any set whose elements are all pairs as a map.)

During the next phase, the data-structure selection phase, we scan the program to find instructions for which the introduction of a base would speed up program execution. For the variable *FREQ*, we find that basing could save hashing operations at lines 17, 19, 35, and 74. In line 17 it is advantageous to base *FREQ* on a base B_1 and to give *C* **elmt** B_1 basing. Hashing operations can be saved at lines 38, 39, 51, and 52 by basing *HTREE*. Lines 38 and 39 then suggest that C_1 and C_2 should be given **elmt** basing, and hence that the first component of *GETMIN*'s return value should be given the same basing. This implies that *C* at line 76 and *A* at lines 68 and 70 should be given the same representation. Since *A* appears as an argument of *F*, that is, of *FREQ*, we see that basing *HTREE* on the same base as *FREQ* will eliminate the need for a hashing operation at lines 38 and 39. (Note that, if C_1 and C_2 have **elmt** basing at line 35, we will need to dereference them before we can perform the addition. Even so, since a pointer dereference operation is cheaper than a hashing operation, it still pays to keep C_1 and C_2 as base pointers.) However, a hashing operation will be required for *C* at line 35, since *C* is a newly created value. At lines 38 and 39, *C* becomes part of the range of *HTREE*, which raises the question as to how range elements of *HTREE* are to be represented. The optimizer answers this question by tracing the range of *HTREE*. We find that at line 52 a range element of *HTREE* is assigned to *B*, which is used to index *HTREE* at lines 51 and 52, and from this we conclude that it is also advantageous to give **elmt** representation to the range elements of *HTREE*.

We then find that *C* should have the same representation at line 48, and hence that at line 46 *CHARS* can most appropriately have the representation **sparse set (elmt** B_1**)** (**sparse** since we must iterate over *CHARS*) (see [5] for an explanation of the representation qualifiers **sparse**, **local**, etc.). Occurrences of the set *CHARS* are also found on lines 13, 15, and 16. At lines 15 and 16 local basing of *CHARS* would be most efficient. However, in handling line 46 we are constrained by the fact that the SETL semantics allows an object to be modified while it is being iterated over. Since the optimizer does not yet attempt to detect cases in which the object iterated over is not modified inside an iterative loop, for the moment this requires us to create a copy of *CHARS*. In the current system this is actually implemented by assigning the set *CHARS* to an auxiliary shadow variable *CHARS*.1 and then iterating over *CHARS*.1. But the value of a local object cannot be shared between two variables (see [5] for full details), so that assignments from or to local objects cannot be implemented as pointer transfers but would require copying on assignment. Since dynamic checks made by our current system can avoid copying remote objects which are "logically copied" by transferring a pointer, but cannot avoid actual physical copying of a local object when such an object is copied logically, the optimizer concludes that *CHARS* should not be given local basing at line 46. However, giving *CHARS* sparse representation at line 46 would require hashing operations at lines 15 and 16. In this situation we could either give two different representations to *CHARS* and convert somewhere between lines 21 and 46, or choose some common representation which is reasonably efficient for both occurrences. Since a conversion

between two set modes, like a copy, requires iteration over the set, the optimizer's heuristic determines that a common basing is more advantageous than a conversion and declares *CHARS* to be a remote set.

The next optimization phase, conversion analysis, partitions the occurrences of each variable V into equivalence classes C such that V has the same data representation at each occurrence belonging to the class C, and then it inserts required conversions between representations at low-frequency program points. In our example, this has the following effects. We have chosen to represent C in **elmt** form at line 35. Since the result C of the addition (concatenation) operation which appears in this line is necessarily a string, a split variable $C.1$ is generated and given **elmt** form. Then a conversion between the string C and the base pointer $C.1$ is inserted. At line 67, C and N are assigned **om**, SETL's undefined value. Since N is an operand to the test $M < N$ at line 69, the undefined value would be illegal at line 69. However, since the definition of N appearing in line 67 is connected to line 69 by a path free of other assignments to N, an error arises if execution ever follows this path. Whenever the optimizer encounters a situation where an operand can have a value which is illegal in the context of the operation, it prints a warning message to alert the user to this fact. Note that the redundancy of this warning at line 69 is not detected: If N equals **om**, then the test $M < N$ is not evaluated. However, the present optimizer is not sophisticated enough to detect this fact. Similar warnings are issued for lines 74 and 76 to indicate that the values returned by *GETMIN* may not be **om**.

All in all, the output of the conversion analysis phase is as follows (*__MAIN*.9, etc., refer to the procedure and statement number of the newly created conversion):

Convert *C* from **string** to **elmt** *OPT*#1021 at *__MAIN*.9
Convert *C* from **string** to **elmt** *OPT*#1021 at *__MAIN*.20
Possible error if *N* is **om** at *GETMIN*.5
Possible error if *C* is **om** at *GETMIN*.9
Possible error if *N* is **om** at *GETMIN*.10

Conversion analysis is the last subphase of the optimizer's automatic data-structure selection phase. The data structures produced by the optimizer for the program under consideration have already been presented in Section 2.1.

The final phase of the optimizer is copy optimization. Here we aim to determine the set of all destructive uses of values *VAL* at which *VAL* need not be copied before the operation is performed. We have already mentioned that at line 74 *F* need not be copied. *CHARS* is used destructively at line 16. Since at this point only the components of *CHARS*, but not its actual value, are shared, the optimizer finds that no copy is required. Similar remarks apply to the remaining destructive uses, and, indeed, in the sample program which we are considering no value ever needs to be copied. These facts are all successfully ascertained by our optimizer's copy optimization phase, which is able to produce the following output:

No copy is required for *CHARS* at *__MAIN*.11
No copy is required for *FREQ* at *__MAIN*.12
No copy is required for *FREQ* at *__MAIN*.13
No copy is required for *FREQ* at *__MAIN*.20

```
 1  1  program TOPSORT;
 2  2
 3  2    var G; repr G: mmap{general}set(general); end repr;
 4  5
 5  5    read(G); print(G);
 6  7
 7  7    N := domain G + range G;      $ N is the set of nodes of G.
 8  8
 9  8    $ Compute the number of predecessors for each node C in N.
10  8    NUMPREDS := {[C, #{P ∈ N | [P, C] ∈ G}] : C ∈ N};
11  9
12  9    $ Collect the nodes which have no predecessors.
13  9    NOPRED := {C ∈ N | NUMPREDS(C) = 0};
14 10
15 10    S := [ ];
16 11    (while NOPRED /= { })
17 12          C from NOPRED;  $ Pick a node with no predecessor.
18 13          S := S with C;        $ Add the node to the sorted sequence.
19 14          (∀F ∈ G{C})     $ for all successors of C
20 15            NUMPREDS(F) := NUMPREDS(F) - 1;
21 16        if NUMPREDS(F) = 0 then NOPRED := NOPRED with F; end;
22 19      end ∀;
23 20    end while;
24 21
25 21    print(S);
26 22
27 22  end program TOPSORT;
```

Figure 3

No copy is required for *HTREE* at *__MAIN*.21
No copy is required for *HTREE* at *__MAIN*.22
No copy is required for *HCODE* at *__MAIN*.32
No copy is required for *F* at *GETMIN*.9

As a second simple example we consider the SETL code shown in Figure 3, which represents Knuth's topological sort algorithm [13, p. 262]. From line 19, our automatic data-structure selection mechanism determines that *G* should be based. *C* and *F* get **elmt** representation, and *NUMPREDS* and *NOPRED* are also represented using this base. Our data-structure choice procedure finds that only one base, corresponding to the nodes of the graph *G*, should be introduced for this program.

During automatic data-structure selection we also attempt to determine whether maps are necessarily single valued, or whether we have to represent them by our more general multivalued map structures (i.e., **mmap** rather than **smap**; see [5]). This distinction is important since **mmap** representation requires more storage and slows execution speed for operations such as iteration, insertion of a new element, and retrieval of an element. To make this determination, we observe that the use of *G* in line 19 suggests that *G* is an **mmap**, even though its use in line 7 would allow it to be either an **smap** or an **mmap**. Thus we conclude that *G* must be an **mmap**. The uses of *NUMPREDS* in lines 13, 20, and 21 suggest that *NUMPREDS* is probably an **smap**. However, the set former at line

Variables for program *TOPSORT*

base *OPT#*814: **general**;
G: **mmap**{**general**} **set**(**general**);
G.1: **remote mmap** {**elmt** *OPT#*814}
sparse set (**elmt** *OPT#*814);

Variables for procedure *__MAIN*

C: **elmt** *OPT#*814;
F: **elmt** *OPT#*814;
N: **remote set** (**elmt** *OPT#*814);
NOPRED: **remote set** (**elmt** *OPT#*814);
NUMPREDS: **local mmap** {**elmt** *OPT#*814} **set**(**integer**);
P: **elmt** *OPT#*814;
S: **tuple** (**elmt** *OPT#*814);

Figure 4

10 might form an **mmap**. It is clear upon visual inspection that in line 10 *C* never assumes the same value twice, so that the map formed at line 10 must be single valued. However, our current optimizer does not detect this fact and thus cannot declare *NUMPREDS* to be single valued, as this declaration, if false, would cause the program to terminate abnormally. The results of automatic data-structure selection are therefore as shown in Figure 4.

After the optimizer has partitioned the occurrences of each variable into equivalence classes according to the data structures selected, it finds that only the input variable *G* has been "split" into variables having different representations. This is due to the user-supplied declaration for the form of the input variable *G*. The necessary conversion of *G* to its internal form is then done immediately following the input operation:

Convert *G* from **mmap**{**general**} **set**(**general**)
to **remote mmap** {**elmt** *OPT#*814} **sparse set** (**elmt** *OPT#*814)
at *__MAIN*.5

Note that, at the final implementation level, the data-structure declarations automatically generated by our optimizer cause *G* to be represented as a vector of linked lists. This favorable representation is generated in spite of the fact that, as read in, the elements of *G* are pairs with arbitrary components; that is, nothing is said about the type of the nodes of the input graph. The base introduced by the optimizer serves to map perfectly arbitrary nodes of *G* to small positive integers that are then used to index the vector. Thus we can claim that our optimizer automatically selects the data structures suggested in [13]. Moreover, it also uses an efficient technique for mapping arbitrary graph nodes into small integers, namely, a linked hash table. Consequently, our automatically generated data structures are a bit more sophisticated than those which Knuth suggests. (There are, however, certain finer points that our optimizer fails to handle optimally. Those are discussed below in the concluding section of this paper.)

Following the course of optimization to its final phase, we note that copy optimization shows that none of the destructive uses requires a copy before the modification:

No copy is required for **domain** *G* at *__MAIN*.7
No copy is required for *NOPRED* at *__MAIN*.12

No copy is required for *S* at *__MAIN*.13
No copy is required for *NUMPREDS* at *__MAIN*.15
No copy is required for *NOPRED* at *__MAIN*.16

5. SUMMARY AND CONCLUSIONS

The examples reported in the preceding section indicate that in many cases our optimizer is able to choose data structures that do not lie too far from those that a trained programmer, operating in a straightforward spirit, might select. Once the types of the input variables are declared, the types of all remaining variables are found with high accuracy. Conversions between different data representations for the same variable are inserted at appropriate points, and a large fraction of all logically unnecessary copying operations are eliminated. However, some of the data-structure decisions that would be made routinely during manual transcription of a SETL program into a lower level language elude our optimizer. The data structures chosen for the topological sort example presented above illustrate this observation.

As we have already noted, the data structures chosen by the optimizer for this program are surprisingly close to those initially prescribed by Knuth [13] for this algorithm, and even surpass these low-level structures in the efficient conversion of graph nodes into small indices used to access various parts of these data structures. (It is instructive to note that the implementation of this algorithm in PASCAL given by Wirth [23] uses linear search to locate each newly read graph node in a list of nodes already read. This makes his algorithm into a quadratic rather than linear algorithm.) However, this example also highlights certain issues which our optimizer fails to handle optimally.

One of these appears if we consider the set *NOPRED*, which is only used as a list from which an arbitrary element is selected. Here **sparse** representation is preferable to **remote**. However, our optimizer's failure to detect the advantage has a rather deep root. A programmer familiar with the topological sort algorithm will realize that a node is never added to the *NOPRED* set twice, so that no check for duplicate elements is necessary: *NOPRED* can therefore be represented as a simple list. This fact lies hidden far too deeply in the logic of the *TOPSORT* code for an optimizer to have any chance of discovering or exploiting it. Consequently, **remote set (elmt** *B*) representation is chosen: This speeds up the test for duplicate elements which the optimizer (wrongly) treats as if it were necessary, but at the cost of slowing down the operation *C* **from** *NOPRED* (line 17 of *TOPSORT*), which an experienced lower level programmer would choose to optimize. To obtain something like the preferred "list" representation in SETL, we would have to treat *NOPRED* as a tuple rather than a list. (This can be accomplished by making a small (but significant) change to the *TOPSORT* source code; namely, we can change the set brackets in lines 13 and 16 of *TOPSORT* to square brackets and the **from** operator in line 17 to **frome**.)

Once this is done, the standard SETL implementation of tuples, which leaves a certain amount of growth space at the end of each tuple, turns the uses of the *NOPRED* variable (lines 17 and 21 of *TOPSORT*) into stacklike operations not conspicuously inferior in efficiency to list manipulations. Note, however, that SETL's present data-structure representation language does not allow us to

specify a list rather than a tuple representation for *NOPRED* even if we handle it as a tuple. To achieve this effect declaratively, we would have to support a presently unsupported internal data representation, namely, tuples represented as lists. This, and tuples represented as B-trees, seem the two most desirable presently unsupported representations. If made available, they could be represented by declarations of the form

T_1: **list tuple**; T_2: **btree tuple**

in the representation language. The optimizer could then detect some, but not all, cases in which these representations are preferable to the array representation of tuples employed presently.

As already noted, our optimizer fails to detect that the map *NUMPREDS* used in *TOPSORT* is single valued, rather than multivalued. This is not hard to detect, but to do so we would have to perform a closer symbolic analysis of the set former appearing in line 10 of *TOPSORT* than is now carried out.

All in all, we can say that

(1) the simple data-structure choice heuristic employed by our optimizer captures a surprisingly large part of the information which shapes lower level data-structure design decisions, at least in their broader lines, but
(2) more refined aspects of data-structure choice, especially those which depend on an a priori knowledge of set sizes or operation frequencies, or bounds for the sizes of some program objects in terms of others, and of logical relationships of identity, inclusion, and disjointness are missed. Consequently, the optimizer is not able to choose accurately between closely related data structures; moreover, the final code it generates continues to contain tests that a programmer working in a lower level language would know could be omitted. Thus
(3) the optimizer improves code efficiency substantially: generally by a factor of two or more, in some particularly favorable cases by as much as eight; but the final efficiency obtained still falls an order of magnitude short of the efficiency that can be obtained by lower level recoding of a SETL source program. We estimate that a refinement of our optimization techniques could reduce the efficiency advantage of lower level programming to a factor well under ten.

The current prototype of the optimizer, which has been used to derive the results presented above, has been written in SETL, requiring approximately two man-years to write some 24,000 lines of source and documentation. To optimize the Huffman coding program presented in this paper required 26 seconds on an Amdahl 470 V8 computer, while the topological sorting program required 15 seconds. Our experience to date indicates that our prototype, running in an 8 Mbyte virtual address space on an Amdahl 470, optimizes about 100 statements per minute for programs of 75 SETL statements, and about 55 statements per minute for programs of 300 statements. We have not yet attempted to optimize larger programs.

Optimization increases the execution speed of the Huffman coding example by a factor of 2.1, and the topological sorting example by 1.2 (see Table I). The poor

Table I. Relative Performance (>1 Indicates Faster)

Program	Size (exec. stmts.)	Ratio to default system	
		Manual decl.	Optimizer
Heap Sort	35	1.772	1.274
Topological Sort	19	1.129	1.190
Huffman Coding	43	1.694	2.069
Sieve	17	2.794	2.387
File Difference	116	4.026	2.254
Available Expressions	222	2.590	2.646

Note. Sieve computes the first N prime numbers. File Difference compares the records of two files and creates a listing of the lines that do not match. Available Expressions is an implementation of the standard data flow algorithm, taken from our optimizer.

result in the latter example is due to the dominance of the **read** statement. If the results are adjusted by elimination of the time required for the input operation (which we cannot optimize), the code which remains is seen to run 3.5 times faster than the unoptimized code. In this example, running time is dominated by the inner set former of statement 8. If we replace this statement by a loop over the graph edges to count the number of predecessors for each node (rather than forming the sets of predecessors), execution speed increases by a factor of 5.6.

The overall space savings in the Huffman coding example is 33 percent, while the topological sorting example requires roughly the same amount of space. The result in the latter case is again determined by the dominant role of the **read** statement contained in the topological sort code; in fact, if we modify this program to eliminate this input operation, the optimized code which remains is seen to require only 6.7 percent of the storage required by its unoptimized version. This indicates that the optimized program allocates almost all storage statically.

The prototype optimizer itself makes use of manually selected data-structure declarations. For small programs (approximately 75 statements), the inclusion of these declarations increases the execution speed by a factor of two, while for larger programs (more than 250 statements), the speedup factor exceeds eight. The measurement tools available to us are not yet sufficient to determine precisely the amount of storage saved by using the more compact based data structures; however, we have observed that the total number of words recovered by the SETL garbage collector is decreased by a factor of five in the presence of data-structure declarations.

A current aim of our work is to recode the optimizer itself, thereby making it efficient enough to be usable routinely as a component of the SETL compiler. In parallel, we are investigating the use of optimization techniques for detection of common programming errors.

REFERENCES

(Note. References [4, 10, 16] are not cited in the text.)

1. AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
2. AHO, A.V., AND ULLMAN, J.D. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1977.

3. ALLEN, F.E. Program optimization. In *Annu. Rev. Autom. Program. 5* (1969), 239–307.
4. COCKE, J., AND SCHWARTZ, J.T. *Programming Languages and Their Compilers.* Courant Inst. of Mathematical Sciences, New York Univ., New York, 1970.
5. DEWAR, R.B.K., GRAND, A., LIU, S.-C., SCHWARTZ, J.T., AND SCHONBERG, E. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM Trans. Program. Lang. Syst. 1*, 1 (July 1979), 27–49.
6. DEWAR, R.B.K., AND SCHONBERG, E. The elements of SETL style. In *ACM 79*: Proceedings of the 1979 Annual Conference, Detroit, Mich., Oct. 29–31, 1979, pp. 24–32.
7. DEWAR, R.B.K., SCHONBERG, E., AND SCHWARTZ, J.T. *Higher Level Programming: Introduction to the Use of the Set-Theoretic Programming Language SETL.* Courant Inst. of Mathematical Sciences, New York Univ., New York, 1981.
8. HECHT, M.S. *Flow Analysis of Computer Programs.* Elsevier North-Holland, New York, 1977.
9. JONES, N.D., AND MUCHNICK, S.S. Binding time optimization in programming languages: Some thoughts toward the design of an ideal language. In Conference Record of the Third ACM Symposium on Principles of Programming Languages, Atlanta, Ga., Jan. 19–21, 1976, pp. 77–94.
10. KAPLAN, M.A. Relational data flow analysis. Tech. Rep. TR-243, Dep. of Electrical Engineering and Computer Science, Princeton Univ., Princeton, N.J., Apr. 1978.
11. KAPLAN, M.A., AND ULLMAN, J.D. A general scheme for the automatic inference of variable types. In Conference Record, Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Ariz., Jan. 23–25, 1978, pp. 60–75.
12. KENNEDY, K.W. A survey of data flow analysis techniques. In *Program Flow Analysis*, S.S. Muchnick and N.D. Jones (Eds.). Prentice-Hall, Englewood Cliffs, N.J., 1981, pp. 5–54.
13. KNUTH, D.E. *The Art of Computer Programming*, vol. 1: *Fundamental Algorithms*, 2d ed. Addison-Wesley, Reading, Mass., 1973.
14. MUCHNICK, S.S., AND JONES, N.D. (EDS.). *Program Flow Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1981.
15. SCHONBERG, E., SCHWARTZ, J.T., AND SHARIR, M. An automatic technique for selection of data representations in SETL programs. *ACM Trans. Program. Lang. Syst. 3*, 2 (Apr. 1981), 126–143.
16. SCHWARTZ, J.T. *On Programming: An Interim Report on the SETL Project*, 2d ed. Courant Inst. of Mathematical Sciences, New York Univ., New York, 1975.
17. SCHWARTZ, J.T. Optimization of very high level languages. *J. Comput. Lang.* (1975), 161–194, 197–218.
18. SCHWARTZ, J.T. Use-use chaining as a technique in type-finding. SETL Newsl. 140, Courant Inst. of Mathematical Sciences, New York Univ., New York, 1975.
19. SCHWARTZ, J.T., AND SHARIR, M. A design for optimizations of the bitvectoring class. Courant Computer Science Rep. 17, Courant Inst. of Mathematical Sciences, New York Univ., New York, 1979.
20. TARJAN, R.E. Testing flow graph reducibility. *J. Comput. Syst. Sci. 9* (1974), 355–365.
21. TENENBAUM, A.M. Type determination for very high level languages. Courant Computer Science Rep. 3, Courant Inst. of Mathematical Sciences, New York Univ., New York, 1974.
22. TSUI, W.H. A reformulation of value flow analysis. SETL Newsl. 181, Courant Inst. of Mathematical Sciences, New York Univ., New York, 1977.
23. WIRTH, N. *Algorithms + Data Structures = Programs.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

Received June 1981; revised July 1982; accepted July 1982