THE UNIVERSITY OF ROCHESTER
THE DEPARTMENT OF COMPUTER SCIENCE

AUGUST 1974

TECHNICAL REPORT NO. 1

AUTOMATIC CODING:

CHOICE OF DATA STRUCTURES

James Richard Low

(Originally published as
Stanford Computer Science
Department Memo STAN-CS-74-452)

ABSTRACT:

A system is described which automatically chooses representations for high-level information structures, such as sets, sequences, and relations for a given computer program. Representations are picked from a fixed library of low-level data structures including linked-lists, binary trees and hash tables. The representations are chosen by attempting to minimize the predicted space-time integral of the user's program execution. Predictions are based upon statistics of information structure use provided directly by the user and collected by monitoring executions of the user program using default representations for the high-level structures. A demonstration system has been constructed. Results using that system are presented.

A dissertation submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

The research reported here was supported in part by the Advanced Research Projects Agency of the Department of Defense under Contract DAHC-15-73-C-0435.

ACKNOWLEDGEMENTS

I am truly grateful for the help and support of the Artificial Intelligence Laboratory and Computer Science Department here at Stanford. I would be negligent if I did not also mention my appreciation to the following people: Donald Knuth, who strongly influenced my ideas towards software monitoring during the summer of 1970 when I was part of his FORTRAN optimization study, and who later directed me in reading and research in the field of data structures; Terry Winograd and Forest Baskett, members of my reading committee, who made many useful suggestions about this dissertation; Dan Swinehart, Russ Taylor, Hanan Samet and Bob Sproull, the SAIL hackers who acted as sounding boards for my ideas; and lastly, and most importantly, Jerry Feldman, my advisor, who was always there when I needed him, ready at any time to think about, and talk with me about, any problems I was having.

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

SECTION 1

INTRODUCTION

Many of the processes used in producing computer programs may be performed with less user effort by using specialized computer programs. There has been a continuous stream of developments which decrease the effort of a human programmer in producing a computer program, including symbolic assemblers, macro assemblers, algorithmic languages, text editors, debugging systems and so forth. An important part of this series of developments has been the development of higher level programming languages. These allow the user to ignore low level details of implementation and have the system provide for them.

Automatic Coding research involves the design and efficient implementation of very high level programming languages. Examples of features available in such programming languages include pattern matching, extensible data types, associative retrieval, and complex control structures including backtracking, coroutines, multiprocessing, message passing etc. Current research ([Bobrow73b,Smith73,Tesler73]) is aimed at developing techniques for efficient implementation of these features.

Automatic Coding includes such things as algorithm transformation, optimization and compilation. In general, it takes one form of a program and translates that form into another which can be executed more efficiently. Traditional *optimizing* techniques involve transformations of the execution flow structure of the program, such as moving computations out of loops and elimination of redundant computations. Until recently there has been little work done on having a compiler optimize the representation of data. The reason for this is clear. Most programming languages offer only data types which have explicit implementations. If other data structures such as variable length strings are provided, their internal representations are also fixed. Common languages may allow the user to aggregate words into arrays or records, but do not provide the user with a representation-free way of specifying his data structures with the generality needed for a translator or compiler to choose a suitable, tailor-made data structure. The compile-time data structures (PL/I structures, ALGOL 68 structs) some algorithmic languages allow are very detailed and maintain a close tie with their implementation. They are usually equivalent to the assembly language representations such as DSECTS in 360-ASSEMBLER ([IBM69]) which merely define the fields of a contiguous block of storage.

Recently many languages, including QA4 ([Derksen72]), PLANNER ([Sussman70, Baumgart72]), SETL ([Morris73]), MADCAP[Morris73]), VERS2([Earley73b]), CONNIVER([Sussman72,McDermott72]) and LEAP ([Feldman69]) have incorporated high-level abstract data structures (information structures) based on relations and sets. These information structures provide not only the representation independence we desire, but also give the user good abstract models for his data. A programmer can think in terms of such abstract information structures as sets rather than in terms of linked record structures provided by languages like PL/I. Usually this is much simpler and the programmer is able to design and debug his programs more quickly. Unfortunately, users have often been reluctant to use these information structures to their fullest in production programs because of runtime inefficiencies. These inefficiencies are caused by the suboptimal low-level implementations of these information structures; the implementations being a compromise over all intended usages. Thus, in most existing programming systems the user must choose between the conceptual efficiency of expressing his data using high level

information structures, and the runtime efficiency available using lower level data structures. We believe these runtime inefficiencies can be remedied by providing a library of representations for the information structures and a user assisted automatic system to choose appropriate elements from this library for the user's programs.

The main problems in such an automatic approach are: 1). Building a library of representations; 2). Characterizing the properties of these representations; 3). *understanding* how the user's program actually uses the abstract information structures; and 4). efficiently matching the user program needs with the properties of the known representations.

The research reported here is an attempt to demonstrate the feasibility of such automatic representation selection. We allow the user to express data in terms of relations, sets, and sequences. Our system then obtains information by analysis of the program, execution statement profiles and interrogation of the user. Using that information, the system selects efficient (in terms of time and space) low level implementations for the user's information structures from a fixed library of information structure representations. Though we have considered only a few abstract information structures (those available in our programming language, SAIL) we believe the techniques used in this system are generally applicable.

We have constructed a demonstration system which we have used to process several example programs. Example programs and the structures selected for their information structures are included in Section 6. A marked improvement in execution time (over using a default representation) is shown in several of these. Others gave us some surprises and indicate areas for future research.

1.1 TRADITIONAL OPTIMIZATIONS

We are concerned with optimization in our choice of representations for information structures in that we wish to decrease the cost of running the finished program. This is the same goal as that of classical compiler optimization. In this research we have emphasized solving problems in representation selection rather than using standard optimization techniques.

Traditional optimizations ([Allen69, Cocke70, Geschke72, Kildall72, Wulf73]) are concerned with reducing the execution time necessary for arithmetic calculations by performing equivalence preserving transformations on the program being optimized. Many of these optimizations are also applicable to operations involving information structures. Our system does not include such optimizations but it should not be extremely difficult to add such optimizing strategies to future data structure selection systems.

CONSTANT FOLDING AND PROPAGATION

Expressions involving only functions of constants may be evaluated at compile time if the functions do not have side effects and given the same argument always return the same value. If a constant valued expression is assigned to a variable and that variable is not changed following such assignment then we may treat that variable as if it were the constant expression itself. Thus in the sequence:

```
ROWSIZE ← 10;
I ← 1;
J ← 5;
INDEX ← I*ROWSIZE + J;
A[INDEX+X] ← 4;
```

we can realize that *INDEX* will be 15 and that there is no need to compile the code to compute its value since its value may be computed at compile time. If the array *A* were statically allocated the address of *A[INDEX]* could also be computed at compile time. Thus, we would only need to generate code that will add the value of *X* to the computed value of the address of *A[INDEX]* to obtain the address of *A[INDEX+X]*.

Information structure constants such as constant sets or constant sequences do not appear to be as common as simple arithmetic constants but when they are present the above technique may beneficially be applied.

COMMON SUB-EXPRESSION ELIMINATION

We often see the same expression being computed several times without its arguments being changed. Common subexpression elimination is designed to recognize such occurrences and avoid the redundant computations by saving the expression in a temporary. For example:

```
A[i + j] ← B[i + j] * A[i + j];
```

The expression *i+j* must be computed only once. Similarly the addressing function for subscripting array *A* need only be computed once.

Information structures such as sets and sequences usually take more than a single word to store. Therefore application of common sub-expression elimination must be carefully weighed to see if the time saved in avoiding redundant computation is more important than the added storage needed to save temporary results.

CODE MOTION

Often, computations can be moved from portions of the program which are executed very often (such as inside loops) to places where they would be executed fewer times without changing the meaning of the program. For example:

```
for I ← 1 step 1 until N do
        A[I] ← J * X + I;
```

The expression *J * X* need only be computed once before the loop and then saved in a temporary. Thus we might compile the above as if it were:

```
TEMP ← J * X;
for I ← 1 step 1 until N do
        A[I] ← TEMP + I;
```

Application of these techniques to multi-word information structures involves tradeoffs similar to those in common sub-expression elimination. We must determine if the execution time saved is more important than the space used to store the temporary result.

REGISTER ALLOCATION and DEAD VARIABLE ANALYSIS

Modern day computers often have a number of fast working registers. It is often beneficial to keep the most commonly used variables and common subexpressions in these working registers rather than memory. There are usually very few of these registers so a compiler wants to know when it no longer needs to retain the value of an expression. For example in the sequence:

```
B ← A + 1;
J ← X * B;
B ← 5;
```

we would calculate the value of $A + 1$ in a register and would notice that the use of B in the multiplication is the last use of B before B is given a new value. B is said to be *dead* after the multiplication. Therefore any register that was known to contain B may be reused. In the above example we would be able to compute the value of the multiplication in the same accumulator as we computed $A + 1$. (In this example, a smart compiler would realize that the value $A + 1$ need never be actually stored into the memory location B.)

The notion of dead variables is quite important in operations involving high level structures. To make a copy of a simple arithmetic value is usually quite cheap, but making a copy of a set can be very expensive. Consider the statements:

```
A ← B ∪ C;
B ← { x, y, z };
```

The subroutine implementing the union operator might simply insert all the elements of its second argument in the first argument. Thus, in the first statement above, we would normally have to make a copy of B and pass the copy to the union subroutine. This, however, is the last use of B before it is given a new value. Therefore the copying operation is unnecessary. This copy avoidance has tremendous payoff ([Schwartz74a]).

We will not discuss classical optimizations in any more detail. They are very important and should be included in compilers which select data structures. We imagine new optimization techniques (including interactive application of the above techniques [Knuth74]) will be found which are even more helpful to optimizing programs involving high level data structures, than the traditional ones mentioned above.

1.2 INFORMATION STRUCTURES

Examples of common abstract information structures which would be nice to have in our programming languages include simple queues, stacks, priority queues, sequences, sorted sequences, sets, disjoint sets, ordered sets, relations and mapping functions. Each has some particular semantic properties which make it conceptually appropriate for expressing certain algorithms. The right information structure provides the important properties of the data, and the essential primitives necessary for manipulating the data. At the same time, non-essential (and implementation restricting) details are suppressed. For example, consider an information structure which is logically a set; that is, an unordered collection of unique objects. In the absence of the set data-type a programmer would probably use a sequence for that data, or end up programming his own set representation. While the operations on a set may be easily coded as appropriate operations on sequences, there is a conceptual overhead (and chance for bugs) in making sure that only a single copy of each object is in the sequence. It should be much easier to prove properties of the program knowing that the information structure is a set rather than having to derive that knowledge from the representation of the set in terms of a sequence or other construct, and from the way the representation is updated and accessed.

An important attribute of a system having the *right* information structures, and the attribute on which our system is based, is that optimizers can do a much better job of selecting data representations when they are given a *free hand* and are not encumbered by non-essential details. With our set example above, we can see that an optimizer given the program in terms of sequences, would find it extremely difficult to recognize that the information structure was really a set. Therefore it would not be able to consider such other set representations as boolean arrays or hash tables. The optimizer would be forced to represent the data in some sequence representation which might be inferior to available set representations.

A commonly held view is that the user should have originally expressed his information structure in terms of its final representation, say a hash table. This opposes many high level language principles. As stated earlier it is much more difficult to prove properties of (and to debug) programs at the representational level. What may be the best representation when the program is designed may not be the best later when such attributes as number of data objects change. Thus, enormous reprogramming costs may occur in order to change representations during the lifetime of a program, or increased computer costs from using inefficient representations may be incurred. Similarly the best representation for the data on one machine, might be decidedly inferior on another machine. Programming at the representational level, rather than at the information structure level, is therefore costly in terms of debugging, maintenance, and portability of programs.

Our final rationale for use of abstract data structures is that we are working towards the day when computer programs are mechanically generated by other programs ([Balzer72, Feldman72a]). It should be much easier to generate programs at the abstract level than at the representational level.

1.3 CODE GENERATION

To generate code for manipulating information structures, we must consider which primitives are necessary in the generated code to accomplish a high-level operation. These primitives normally will be either closed subroutines, or in-line code. In some representations certain primitives may essentially be null operations. For example, certain representations may not need explicit *copy* or explicit *release storage* primitives because these operations will be performed inside other primitive operations.

Consider an assignment statement involving a set variable and a set expression

SETVAR ← SETEXPR;

A single assignment primitive could take the two sets as arguments and do the assignment. Another implementation might make a copy of the right side (*SETEXPR*), if necessary; release the storage occupied by the set on the left side, if necessary; and then put the descriptor for the copy in the set variable. This implementation has the advantage that it *knows* whether it has to make a copy of the right side (*SETEXPR*) or whether the right side is simply the result of a computation. In an optimizing compiler, we could also recognize special cases where the space occupied by the set variable on the left (*SETVAR*) need not be released because it is known to be empty or previously released. To get the same effect with an assignment primitive, we would really need four assignment primitives, each called depending on knowledge of whether the left side needed to have its storage reclaimed or the right side needed copying.

Alternatively, an assignment primitive is sometimes more efficient in expressing the concept of assignment than lower level primitives. Consider the representation of two sets as fixed length arrays. An assignment primitive could simply copy from one array to the other, but if we had expressed the assignment in terms of copy and release primitives we would have to create and destroy some temporary array. Allocation and deallocation of temporary arrays can be quite costly in execution time.

The design of a set of low level primitives is an art very similar to the design of the instruction set of a computer. We feel that much benefit could be derived from studying the ways information structures are used in order to decide which primitives are most beneficial. (See appendices A and B for the description of the primitive operations which we use to describe sets and sequences.)

1.4 RELATED WORK IN HIGH LEVEL ALGORITHMIC LANGUAGES

There are several research projects which are investigating the use of abstract information structures in high-level programming languages. The following are projects which have recently been conducting such research.

SETL([Morris73]). This language is being developed at the Courant Institute at NYU. It is based on set-theoretic principles. Data structures are expressed in terms of finite sets and tuples of heterogeneous objects, where objects are elementary types like integers and character-strings, or more complicated objects such as sets and tuples. Mapping functions are expressed as sets of

ordered pairs of function argument and function value. Most of the work on the SETL project seems to have been spent in designing programming language constructs which are closely related to mathematical set constructs. Major effort has also been spent on a series of progressively more efficient implementations. Recently ([Schwartz74a, Schwartz74b]), work has been done on optimization techniques involving data-type inference (there are no variable declarations in SETL). SETL researchers are also interested in computation avoidance. For example in the expression $x \in (A \cup B)$ there is no need to actually construct the union of A and B as we can simply test whether x is an element of either set. SETL researchers have also investigated other more classical optimizations including dead variable analysis. As mentioned earlier dead variable analysis has a large payoff in avoiding unnecessary copy operations.

MADCAP([Morris73]).The latest of the series of MADCAP languages (MADCAP VI) being developed at the Los Alamos Scientific Laboratory by Morris and Wells is very similar to SETL. There are minor differences in some semantics, e.g sets must contain objects of a single type, and some not so minor differences. One such difference is that MADCAP is a pointer language rather than a value language. For example, the sequence:

```
A = <1,3>
B = A
B(1) = 2
```

causes A to have the value $\langle 2,3 \rangle$ rather than the expected value of $\langle 1,3 \rangle$. The emphasis again has been on the language design and not in optimizing the implementation.

ELI. This is the first of a family of extensible languages being developed by the ECL group at Harvard ([Wegbreit71]). The base language itself does not include constructs of higher level than tuples. It does, however, include extension mechanisms with which the user can define his own abstract data-types and operators to act upon these types (See VERS2 below). The goal of this research has been to provide an extensible language which has several levels of implementation from LISP-like interpreters (highly useful in debugging programs) to very complex optimizing compilers. One important feature of these compilers is the *CLOSURE* mechanism ([Webreit73]). To understand this mechanism we must realize that ELI operators are defined in terms of user written subroutines. These routines normally will do different things depending on the type of their operands. At the lowest level, for example, the operator "+" will do a floating point addition if its arguments are both real, a fixed point addition if its arguments are both integer, and a floating point addition preceded by changing an integer argument to a real value if one argument is integer and the other is real. The purpose of closure is to tailor special versions of routines (both operator definitions and user defined procedures) which *know* the data-types, and perhaps even the values of certain parameters and *free* variables. With such knowledge unnecessary type checking or computation of constant-valued expressions may be avoided. Thus, for the operator "+" it would often be possible for a compiler to generate the single add instruction in-line rather than to generate a call to a generic routine which does the type checking of arguments and then eventually an add instruction. As closure is sometimes a very expensive operation for a compiler to perform, closure is currently invoked by user requests which are very specific as to what quantities are bound. Future research will likely try to automate these decisions. Work is also being done to be able to include invariants in the closure mechanism. For example, we may be able to prove that because of certain types and values being constant, other relationships will hold between certain expressions. We can then often improve the code. In a trivial example we might be able to prove that some set is always empty at a given point in a program. If there is a

conditional expression based on that property, we could transform the expression into an unconditional one.

VERS2([Earley73b]). This is a language being developed by Earley at the University of California at Berkeley. It is actually being implemented as an extension of EL1 (above). It has much of the flavor of SETL, sharing many of the same constructs. It includes relations and sequences of objects (both values and variables). One of the features of this language of great interest to us is the *implementation facility*. With this facility the user can tell the programming system how he wants various data-structures implemented. For example, he may state that he wants a particular set implemented as a sequence, run his debugging tests using the default sequence representation, and then later specify that he wants that sequence implemented using doubly linked lists for production use. Earley has also been interested in high level concepts of loops which he calls *iterators* ([Earley74a, Earley74b]). These iterators are at a high enough level that algorithm transformation can often be used to circumvent actual execution of loops. For example, consider the iterative operation $\{x \in S \mid P(x)\}$, which constructs the subset of set S whose members satisfy the predicate P. Normally this would be implemented using a loop. An intelligent optimizer could realize that no loop needs to be present if we keep an auxiliary set SP which contains all those elements of S which satisfy P. If P is sufficiently tractable (e.g. $P(x) \equiv x > 0$) then to maintain SP we need merely to check P every time we add (remove) an element to (from) S, to decide if we should add (remove) that element to (from) SP. The interesting problems of such *iterator inversion* include deciding when it is the more efficient thing to do, and how to handle complex iterators. The whole concept of iterator inversion, though not currently implemented in VERS2, should strongly influence automatic coding research.

LEAP([Feldman69]). This language was originally implemented at MIT's Lincoln Laboratory by Feldman and Rovner. Apart from normal ALGOL-like features, it contained sets of objects and a single ternary relation between these objects. The original implementation was geared to handling large data bases, much larger than could fit into a single core image. It was used with much success in diverse applications including interactive graphics. The data structure elements of LEAP have since been incorporated into an ALGOL-60 based artificial intelligence language called SAIL. Here the emphasis on handling very large data bases was dropped, and the current implementation allows only small core-resident data bases. Recently the LEAP subset of SAIL was used as part of a basis for adding powerful control structures to SAIL including multiprocessing, coroutines, and message passing as well as a limited form of backtracking ([Feldman72b]). In our demonstration system we use a subset of SAIL as the language in which a user expresses his programs. A more detailed description of the important LEAP features will be given in the next section.

Other related research projects ([Anderson72, Bobrow73a]) involve the development of programming languages for use in Artificial Intelligence research. These programming languages usually have associative data retrieval and complex control mechanisms including call by pattern match and backtracking.

All the projects described above have some central notions as to what types of data structures should be available to the user. Data structures should be expressed in very general terms to rid the programmer of unnecessary implementation details. The programmer, at least at first, should only have to worry about his algorithm and not about details such as bit masks and hash-tables. It is much easier to debug a program and prove it correct if we are dealing with sets rather than some linked structures. Portability of programs is becoming more significant. When we move a program from one computer to another it should be much simpler to change the representation of high level data-structures such as sets and sequences to those more suitable to the new machine than to reprogram application programs using low level structures. Even when we do not consider portability we still derive great benefits from using the higher level data structures. It is a rare production program that does not get modified as its requirements change. A programmer usually finds it simpler to understand (and thus be able to modify) programs written in ALGOL or PL/I than the same programs written in assembly language; similarly, he finds it easier to understand set and sequence manipulations than the corresponding pointer structures which might have been used to represent sets and sequences.

1.5 LEAP

In order to explore the problems involved in doing automatic selection of representations, we decided to build a demonstration system. The programming language we chose to work with is a subset of SAIL ([VanLehn73]). SAIL is a good choice for such a demonstration system because of the LEAP sublanguage. LEAP contains sets, sequences and a ternary relation. Sets and sequences are common information structures and the problems in selecting representations for them are very similar to the problems in selecting representations of other abstract structures such as simple queues, priority queues and stacks. The LEAP ternary relation presents problems similar to those found in partial mapping functions and n-ary relations. The version of LEAP we use is core-resident, so we also restrict ourselves to relatively small data bases. Problems involving large (disk size) and very large (tape library size) data bases are left for future research.

LEAP consists of items (variable names), each of which may have typed datum; sets of items; linear lists (sequences) of items; and a single ternary relation between items (also called the associative store or triples).

The important properties of items are:

1. They are allocated either statically (declared) or from a heap (using the *NEW* generic function). Their lifetime does not follow ALGOL block structure. A given item ceases to exist only when it is given as the argument to the *DELETE* procedure.

2. Each item may have a *DATUM*. A datum is a algebraic, set, or list variable. The datum of an item has the same lifetime as the item itself. We often classify items according to the data-type of their datum. Thus we may speak of type-less or blank items (which have no datum), *STRING* items (whose datum is a string variable), *STRING ARRAY* items (whose datum is a *STRING ARRAY*) etc.

3. Items may be referred to by their name (if declared), or as the contents of an item variable (**itemvar**). In ALGOL-68 notation an **itemvar** would be known as a **ref item**. Itemvar's may receive values by assignment of item expressions, or by pattern matches against the associative store, sets or lists.

SETS

LEAP has finite sets of items. Normal mathematical considerations, such as $\{a\ a\ b\} = \{a\ b\} = \{\ b\ a\}$, hold. The empty or null set is denoted by the name **phi**. Set expressions may be stored in set variables. It is important to note that the semantics of set assignment is to make a copy of the set expression. Thus in the code sequence:

```
XSET ← { a, b, c, d };
YSET ← XSET;
put e in XSET;
```

the resulting sets will be

```
YSET = { a, b, c, d }
XSET = { a, b, c, d, e}
```

LISTS

Linear lists of items (sequences) are also available. They behave much like variable length one dimensional arrays of items. The notation used for an explicit list is {{ *a, b, c, d* }} . "{{" and "}}" are called list brackets.

```
XLIST ← {{ a, b, c, d }};
XLIST[1] ← b;
```

will result in *XLIST* = {{ *b, b, c, d* }}. Other operations on lists include concatenation, removal of items from a list either by index position within the list or by giving the item to be removed, and insertion of items into lists either by index position or after or before named items. The null list is denoted by the name NIL.

TRIPLES

The most powerful abstract data structure in LEAP is the single ternary relation also known as the associative store. The relation instance (a,o,v) is denoted by:

$$a \circledast o \equiv v.$$

The first component is called the attribute; the second, the object; the third, the value. When we are indicating an unspecified element (which might be returned from a search of the relational data base) we will mark that as "?".

Normally we fix one of the elements (usually the first) and use that as a binary relation name. Even when this is done, we may still do searches of the form "? ⊛ o ≡ ?". For example, assume we have a number of relation instances involving a certain item *A* and we wish to copy them to another item *B*. The simplest way of doing this is to search the associative store (see FOREACH's below) for all triples whose second component is *A* and for each such triple, create a new relation instances which differs from the original only by having B as its second component.

In order to take advantage of the high level data structures we must have loops which sequence through sets of items, lists of items and items which satisfy pattern matches on the ternary relation. The mechanism provided by LEAP is called a *FOREACH* statement.

FOREACH STATEMENTS

A foreach statement consists of three parts: a binding list of itemvars (also called foreach locals, or local itemvars) whose elements are analogous to the loop control variable of an ALGOL FOR statement; an associative context; and a statement to be iterated. An associative context consists of elements separated by "∧", where an element is a boolean expression, a set iterator, a list iterator or an iterator based on a pattern match on the associative store.

Each element of the associative context, other than boolean expressions, is said to bind one or more of the foreach local itemvars. That is, the iterator successively will give various item values to the local itemvar. The first element of the associative context which refers to a given local itemvar, binds it. Later uses of the local itemvar in the associative context will use the item previously bound to the local itemvar.
For example:

foreach par,grand | PARENT ⊛ JOHN ≡ par ∧ PARENT ⊛ par ≡ grand **do**

will iterate thought all the pairs of (parent,grandparent) of *JOHN*. The first element, *PARENT* ⊛ *JOHN* ≡ *par*, binds the foreach local *par* and then that binding is used in the second element to find bindings for *grand*.

FOREACH ITERATORS

Set iterators are written in the form:

local ∈ setexpression

These successively bind the local to each element of the setexpression. Since a set is conceptually unordered, the order in which the items of the setexpression are bound to the local is also undefined.

List iterators are written in the form:

local ∈ listexpression

These will successively bind the local to the first element of the list, second element of the list and so on.

The FOREACH statement:

```
foreach X | X ∈ LIST1 do
    S;
```

is thus equivalent (in absence of changes to *LIST1* within the loop) to:

```
for I ← 1 step 1 until LENGTH(LIST1) do
    begin
        X ← LIST1[i];
        S;
    END.
```

Relation iterators bind one or two locals.

```
foreach X | a ⊗ X ≡ b do
```

will iterate through all items in the associative store which are the object component of an relation instance with *a* as the attribute component, and *b* as the value component.
Thus, if the associative store contained:

```
a ⊗ d ≡ b
a ⊗ e ≡ b
a ⊗ f ≡ b
```

then the above FOREACH would be equivalent to

```
foreach X | X ∈ { d, e, f } do
```

As with sets, the order which the bindings are given by this iterator to the local(s) is undefined.

DECOMPOSITION OF FOREACH's

When a FOREACH statement has more than a single element, it behaves as if it were a nest of FOREACH statements each containing a single element. A boolean expression element acts much like an IF statement.
Thus:

```
foreach X, Y, Z | A ⊗ X ≡ Y ∧ (datum(X) > datum(Y)) ∧ Z ∈ ZLIST do
    S;
```

will be semantically equivalent to:

```
foreach X, Y | A ⊗ X ≡ Y do
    if (datum(X) > datum(Y)) then
        foreach Z | Z ∈ ZLIST do
            S;
```

CHANGES TO THE INFORMATION STRUCTURES DURING ITERATION

A difficulty in the semantics of FOREACH statements (inherent in any data structure iterator) concerns changes to an information structure which is currently being processed by some iterator.

For example consider:

```
foreach X | X ∈ SET1 do
    begin
        remove F(X) from SET1;
        CNT ← CNT +1;
    end;

foreach X | X ∈ SET1 do
    begin
        put F(X) in SET1;
        CNT ← CNT +1;
    end;
```

Should changes to the set affect which items are returned by the future iterations of this loop?

If *SET1* in the first *FOREACH* were { *a, b, c, F(a), F(b), F(c)* } and the *FOREACH* would return the items in that particular order, would the statement be executed 3 times, 6 times or some number in between. Similarly if the *SET1* in the second *FOREACH* were executed with the set { *a, b, c* } would the statement *CNT ← CNT +1* be executed only 3 times, or maybe an *indeterminate* number of times as the foreach produced the set:

{ a, b, c, F(a), F(b), F(c), F(F(a)),.....

It seems desirable to minimize the differences in program execution caused by the ordering which *FOREACH* iterators give to to semantically unordered quantities. Therefore the best semantics would have the FOREACH not be affected by changes to the data structure during its executions. Two ways of doing this are apparent. The first is to outlaw operations which alter a data structure which is being processed by an iterator. This unfortunately removes many highly useful constructs such as:

```
foreach X | X ∈ SET1 do
    if pred(X) then remove X from SET1;
```

The other way of solving this problem is to define the semantics as if the set were copied before the FOREACH was entered and then the copy was used to produce the items for the iteration. Thus the above FOREACH would be equivalent to:

```
TEMP ← SET1;
foreach X | X ∈ TEMP do
    if pred(X) then remove X from SET1;
```

We see that any changes to *SET1* would not change *TEMP* and thus alter which items

are returned by the FOREACH. Note that an optimizing compiler might be able to determine that there is no possibility of a set or sequence variable being changed inside a given FOREACH and could then suppress the copy operation. In our demonstration system we always form copies of sets and lists being iterated.

1.6 EXAMPLE OF LEAP SUBPROGRAM

Now let us look the procedure *TRANSCLO* (below), written in SAIL, which uses the LEAP features. We will use this procedure to demonstrate some of the problems and techniques of representation selection. This procedure computes the transitive closure of a reflexive binary relation (*REL*) upon some set of items (*BASE*).

```
set procedure TRANSCLO(itemvar REL; set BASE);
begin "TRANSCLO"
      set RELATED, NEWLYRELATED, FOUND; itemvar X,Y;
      RELATED ← phi; NEWLYRELATED ← BASE;
      while (NEWLYRELATED ≠ phi) do
         begin
               FOUND ← phi;
               foreach X,Y | X ∈ NEWLYRELATED ∧ REL ⊗ X ≡ Y do
                     put Y in FOUND;
               RELATED ← RELATED ∪ NEWLYRELATED;
               NEWLYRELATED ← FOUND - RELATED;
         end;
      return(RELATED);
end "TRANSCLO";
```

The binary relation is represented by a LEAP triple, whose first component is the relation name. *REL* ⊗ *A* ≡ *B* means that *A* is related to *B* by the relation *REL*. The input to this procedure is the relation name, *REL*; and the original set of items, *BASE*. The set *RELATED* will be used to collect all those items which are directly or indirectly related to the original *BASE*. The set *NEWLYRELATED* consists of those items which have been found on the previous iteration of the while loop to be related directly or indirectly to the base. The set *FOUND* is used to collect all those items found to be directly related to the items of the set *NEWLYRELATED* during a single iteration of the while loop. At the end of the while loop, we add all those elements of *NEWLYRELATED* to the collection *RELATED*. The set *NEWLYRELATED* is then given all the objects which were found in this iteration yet were not processed by some previous iteration.

Now let us attempt to select efficient implementations of the information structures of this procedure. We must consider how the various structures are used and their sizes in making such selections. In the absence of global knowledge of how the relational store is used outside the procedure and how the input and output sets are used, we would probably not be able to choose the best representations, but let us see how we might approach the selection process.

First of all we must determine what abstract data structures appear in the procedure. We notice that the only such structures are the four set variables: *BASE*, *RELATED*, *NEWLYRELATED*, and *FOUND*; and the associative store.

Next we must determine which operations are performed on the individual sets. These are assignment, element insertion (*PUT*), set union, set subtraction and foreach iteration. The only operation performed on the relational store is the foreach search with the first two components bound.

We notice that *BASE* and *NEWLYRELATED* are both arguments to a single assignment statement. As a simplification let us assume that this will influence us to choose a common representation for *base* and *newlyrelated*. (Motivation for this will be presented later in Sections 2, and 3). Similarly the statement:

RELATED ← *RELATED* ∪ *NEWLYRELATED*;

will cause *RELATED* and *NEWLYRELATED* to have the same representation. In fact, all the sets of this procedure will end up with a common representation.

With this information, we will attempt to choose a representation for these sets. We immediately realize that we still do not have enough information to choose the best representation. We need to know how large the sets are on the average, and the relative frequency of the various operations. Let us assume that each set is potentially very large though its average size is a small proportion of its potential maximum.

We probably would eliminate set representations such as binary trees and hash tables because set union and set difference operations are time consuming using these representations. If there were a fixed maximum number of items which could be elements of these sets we might consider using a fixed length bit vector, since bit vectors are efficient for union and difference operations. However, depending on set density they may not be very efficient in terms of space, or time needed for the foreach search. Without more concrete information we cannot really decide. It may turn out that the insertion operations so dominate the execution time of the program that we really should consider use of a binary tree or hash table set representation. The savings from using them for insertion may make up for their added costs in doing the other set operations.

We have similar considerations in choosing an appropriate representation for the relation. If we find that the program in which this procedure appears does not have other types of searches on the associative store, we will not have to provide for such searches. We will be able to choose a very specific representation which is sufficient and efficient for this program yet which is not capable of handling all possible operations on the associative store.

In the next section we will examine the overall design of a system which automatically chooses appropriate representations of the high level information structures of LEAP.

SECTION 2

OVERVIEW OF THE SYSTEM

We have implemented a data structure selection system to demonstrate the feasibility of our ideas. The system consists of several computer programs written in SAIL and PDP-10 assembly language. The assembly language parts of the system have been abstracted from the standard SAIL compiler, and are used to parse user programs and do the machine code emission in the final compiler. All the rest is written in SAIL. We make extensive use of the LEAP information structures in the SAIL coded portions. Communication between programs is accomplished by having each program write the contents of the LEAP information structures to a disk file which is later read by the next program in the series. The system we have designed to perform selection of low-level data structures logically consists of three major components: *information gathering*, *structure selection*, and *final compilation and execution* (See Figure 1). We will note other techniques which we feel are applicable to a selection system which are not part of our system.

2.1 INFORMATION GATHERING

The information our selection phase needs about the abstract information structures of the user's program includes such things as their size, the primitive operations performed on them, and values of other parameters which affect the execution time of the primitive operations as implemented for the various representations.

We do not want to require that all of the structures of the same abstract type, such as sets, be represented the same way. We therefore need information as to how to partition the abstract structures into equivalence classes; the members of each class having a common representation. Classes will contain individual information structures of the same type which are *connected* to each other in some way. Such *connections* include being the same positional parameters to some procedure, and being operands to a single instance of some operator. A single representation for a class is necessary to avoid dynamic checking of representation, as well as to avoid potentially costly translations of representation. It also eliminates the need for coding implementations of the primitive operations which take arguments with different representations. Consider the example of a set assignment statement:

```
BASE ← NEWLYRELATED;
```

It may, because of other uses of the sets in the program, be more efficient to have different representations for *BASE* and *NEWLYRELATED* and have the assignment do a translation between them. We realize that this flexibility requires a much larger library of set manipulation routines. If there are n different set representations we will need n^2 assignment routines if we implement this directly, or $2n$ routines if we translate into a single intermediate set representation. In order to decrease the library size, our system sacrifices this flexibility and insists that both the arguments to an assignment be in the same representation, thus needing at most n different assignment routines. Similarly we will insist that both operands to any other binary set operation be in the same representation and that the results of set union, intersection, and

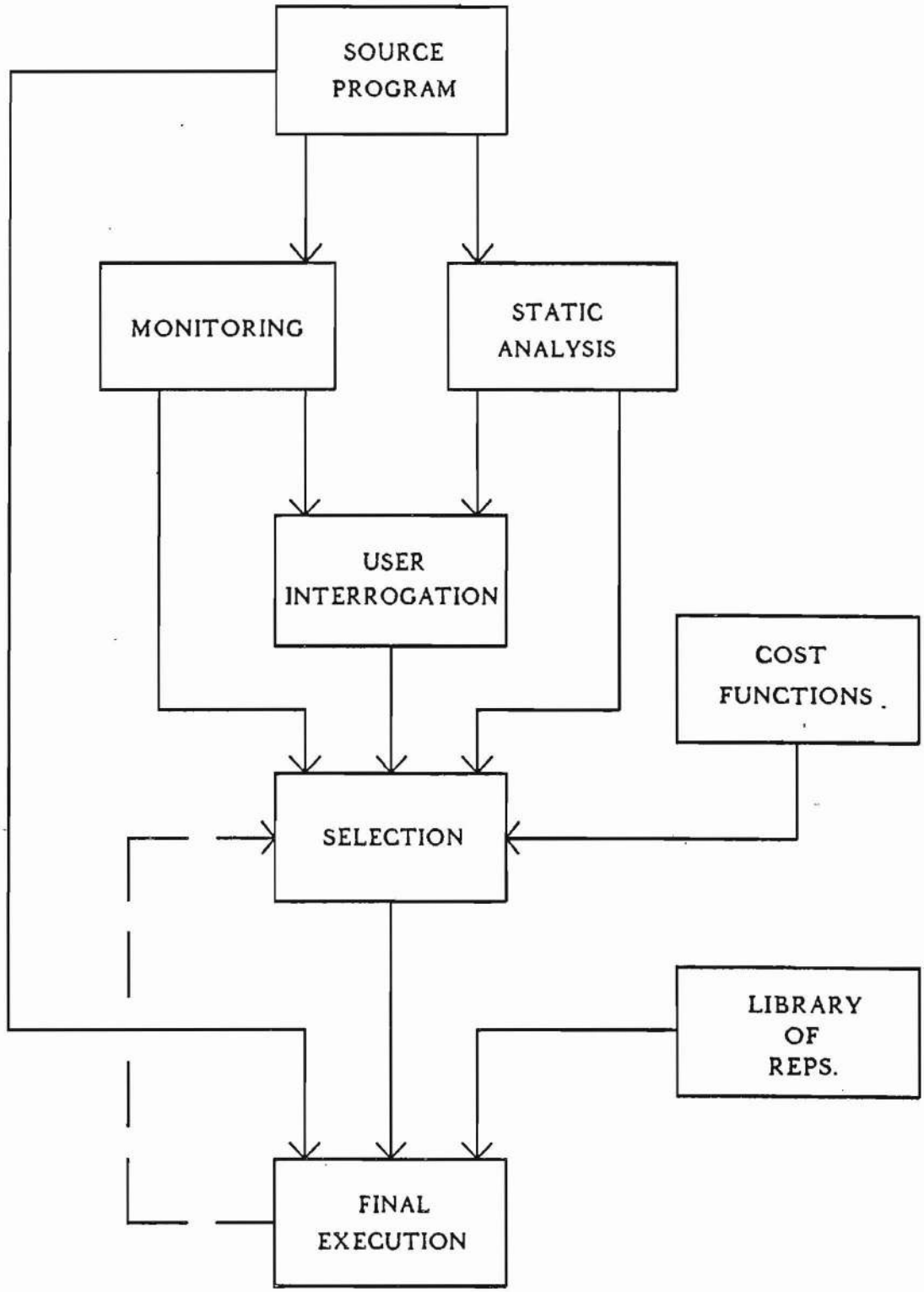
SOURCE PROGRAM
MONITORING
STATIC ANALYSIS
USER INTERROGATION
COST FUNCTIONS
SELECTION
LIBRARY OF REPS.
FINAL EXECUTION

Figure 1

difference be in same representation as the operands to those operators. These representation constraints will usually produce several disjoint classes of set variables, each of which will be forced to have a single representation. We note that this decision to avoid translation of representation may well not be optimal, as is the decision to disallow codings of the primitive operations taking arguments of different representations. We feel that these restrictions were reasonable in a first implementation.

There are many ways of obtaining the required information, including assertions or declarations by the user; monitoring the execution of the user's program (using default representations); static analysis of the program; and interactive interrogation of the user. In the demonstration system we have constructed, we collect statement counts by monitoring the execution of the program. Other statistics of the data use are gathered by asking the user. Partitioning information is obtained by a static analysis to be described later.

2.2 SELECTION

TIME AND SPACE COST FUNCTIONS

A prime prerequisite for making intelligent choices between alternative ways of representing an information structure is a knowledge base containing information about the properties of different representations. We need to know when a representation may be used (applicability), how much space a given representation will require as a function of the number of data objects (storage cost), and the expected time necessary for the primitive operations on this representation as a function of the size of the information structure and other parameters (execution time cost).

The attributes of the various representations are not independent of the programs in which they are used. For example, assume we have a set represented as a binary tree. The time needed to do an insertion into this set is (on the average) proportional to the logarithm of the length of the set. However, if the program inserts elements into this set in ascending order, the binary tree degenerates into a linear linked list and the time needed to do an insertion becomes proportional to the number of elements in the set rather than the logarithm of the number of elements. As a first approximation, though, we consider such attributes of representations as invariant over the programs in which they are used. Thus, the time required for element insertion into a binary tree will be approximated by the average time (i.e. proportional to the logarithm). The predicted execution time is thus a function only of number of elements in the set and not the order in which elements are inserted. Note that the deficiencies of the model for this particular example may be overcome by noticing that elements are inserted in ascending order. In this example the programmer might have used a different information structure if he considered order an important property (perhaps an *ordered* set). We would expect a good programming language to include a multitude of abstract data types or facilities for creating them. As extension mechanisms become more powerful we would hope users would be also be able to define their own abstract information structures, supply the appropriate cost evaluation functions and let the system select representations from a user supplied library for the new information structures.

The number of computer words used to store references to n objects can usually be expressed as a simple function of the maximum number of objects, and the current number of objects. The expected time required for a primitive operation such as union of two sets, is a function of the maximum and current sizes of the abstract structures as well as certain other parameters unique to the primitive operation. With set union we would expect the time for union to be dependent on the percentage of overlap between the two sets.

The attributes of the possible representations are: the applicability predicate; the space function for the representation; and the individual time functions for the primitive operations. These would usually be calculated once by the person who has constructed the selection system. These cost functions may appear either as input data to a structure selector, or actually be explicitly contained in the structure selection component of the system. It is, of course, easier to add new representations if we need only update a data set and not a program, but it is simpler to construct an automatic selection mechanism if the information is explicitly present (so we may make ad hoc adjustments) rather than being present only as data. For our implementation, we chose to obtain the best of both techniques. All information about the representations of information structures is present in the form of procedures. To add a new representation of some information structure to our system, we need only add a new set of procedures to the system which indicate when the representation is applicable, how much storage it requires, and how much execution time is required for each primitive operation. When the attributes of data structures are expressed either as procedures or data to the structure selector, it is a very simple matter to change the structure selector to select structures for a different implementation of the primitive operations (such as when we move the program from one computer to another) by merely changing the cost functions.

PRELIMINARY PROCESSING

The first thing the selection process must do is partition the set and sequence variables and expressions into equivalence classes, members of which have a common representation. The selector then computes which primitive operations are performed on the classes.

The selection process next does some preliminary filtering to weed out obviously undesirable representations. Some representations are incomplete. They may be used only when specific primitive operations are not performed on the class of information structures. If we find that the user has performed such primitive operations we can immediately eliminate those data representations. Another reason for discarding certain representations is that their implementation may depend on knowing certain information at compile time, such as the maximum size of the information structure. In cases where this information is not computable at compile time, but is a function of data at runtime, we must discard these potential representations.

Next, the selector predicts, using time and space prediction functions for each remaining representation, how much time and space would be needed for each information structure equivalence class using each of the remaining representations. If the system notices that with two representations for the same class, one requires both more time and space, the system removes that representation from further consideration. The preliminary selection phase uses this heuristic to filter out representations unlikely to be chosen by the final selector. It also ranks (according to some cost dependent criteria), the representations which have not been discarded as to their likelihood of being the *best*. In our system the possible representations are ranked in increasing order of the products of their expected space and execution time requirements.

SELECTION

After the preliminary selection we should have only a small number of representations which are still candidates for any given class of information structures from the user's program. The Selection phase must worry about the second order effects which arise from having more than one information structure. For example, assume the user's program creates two sets and that our measure of cost is simply the space time product. Also assume that the two sets quickly achieve their maximum size and thereafter remain constant in size throughout the remainder of the program.

let S_{1J} = the space occupied by set *1* using representation *J*
let T_{1J} = the time used by the primitive operations on set *1* using representation *J*
let S_{2K} = the space occupied by set *2* using representation *K*
let T_{2K} = the time used on set *2* using representation *K*
let S_0 = the space used by the non-set operations of the program
let T_0 = the time used by the non-set portion of the program

The cost of the program is thus approximately

$$(S_0 + S_{1J} + S_{2K}) * (T_0 + T_{1J} + T_{2K})$$

The preliminary selector has ranked representations to minimize the expected space time product (such as $S_{1J} * T_{1J}$) for a single abstract data structure, but because of the cross terms (such as $S_{1J} * T_{2K}$) above, this may not be the best choice for minimizing the total cost function. These cross terms indicate that the Selection Phase must consider the representations for all the structures together, and cannot simply approach the individual structures independently. Our final selection phase uses, as an initial approximation, those representations which provide the minimum space time product for the individual structures. It then proceeds to attempt to change individual representations to minimize the predicted *TOTAL* cost. When it can not improve the *TOTAL* cost by changing the representation of a single structure it returns the best set of representations it has found.

2.3 COMPILATION AND EXECUTION

REPRESENTATION DEPENDENT OPTIMIZATIONS

The final stage of the system prepares the user's program for final compilation. In a production system, representation dependent optimizations may be performed during this stage.

For example, consider a program which has statements of the form:

```
if length (SET1) = 0 then
```

It might be more efficient for certain representations to check the expression:

```
if SET1 = phi then
```

and vice-versa. Another example is expressions of the form:

```
SET1 ← SET1 ∪ { a, b, c }
```

With some representations this might be more efficiently implemented as:

```
put a in SET1;
put b in SET1;
put c in SET1;
```

In our implementation no such representation dependent optimizations are performed.

POST-SELECTION MONITORING

Once the system has compiled the user's program it should run it with a special runtime package which gathers statistics to see if there are biases in the way the data structures are used which were not apparent originally. For example, assume the system represents a set as a binary tree (not balanced). It is possible that, because of biases, the tree always degenerates, resembling a linked list. We should be able to go back to the structure selection phase with this added information and see if the system might better select some other data structure to represent the set. We did not have the time to include post-selection monitoring in our demonstration system, so its benefits have not been fully determined.

SECTION 3

ABSTRACT DATA STRUCTURES AND THEIR REPRESENTATIONS

In Section I, we mentioned many abstract data structures which we feel should be available in high level programming languages, including various kinds of queues, stacks, and mapping functions. This list is far from complete. We would imagine other programmers to have their own lists. The ones we chose hopefully form a basis for others. We hope that future systems will not only include the information structures mentioned, but will provide extension mechanisms to allow the user to define new information structures. These extension mechanisms should allow representations for user defined structures to be chosen much the same way as for built-in information structures. In our demonstration system, we have limited ourselves to those abstract data types already available in LEAP: sets, sequences and a single ternary relation. Let us now consider these information structures.

In the following, the reader should remember that a LEAP item is essentially the name of a variable (arithmetic, set or sequence) allocated from a heap. Its internal representation will normally be a contiguous block of storage in the computer's memory (in our implementation this means each reference to an item is an 18-bit pointer). Thus, an item in our implementation is a tuple (PL/I type structure) with a type field, possibly a datum field, and perhaps various other fields which are used for representing certain information structures Figure 2 shows the layout of our items in storage. These other fields are used in the attribute bit representation of sets and the offset representation of triples mentioned below. In certain representations of sets it is beneficial to have an alternate representation of an item such as a small integer index. Translation functions are required for these set representations to take the full 18-bit representation of an item to the small integer index representation and vice-versa. Often we will refer to the item as if it were a value (as in a sorted list of items). Here we are really referring to the integer value of the pointer to the item.

The representations we use do not pack more than one piece of information per computer word, except as explicitly noted below. Thus, even though our representations of items take no more than 18 bits and a PDP-10 computer word consists of 36 bits, we make no attempt to pack two items per word in any representation. In our storage requirements below we will count the number of computer words actually used. Storing a single item with nothing else in the same word will cost one word in storage, not one half a word.

3.1 SETS AND THEIR REPRESENTATIONS

Sets in LEAP are conceptually unordered collections of items. There is no restriction that these items be of the same datum type. We have created a collection of seventeen primitive operations, which are sufficient to perform any of the high level LEAP constructs involving sets, such as assignment, set insertion, removal of items from sets, set union, set difference, set intersection, and FOREACH iteration through a set. (See appendix A for descriptions of each of the primitive operations.)

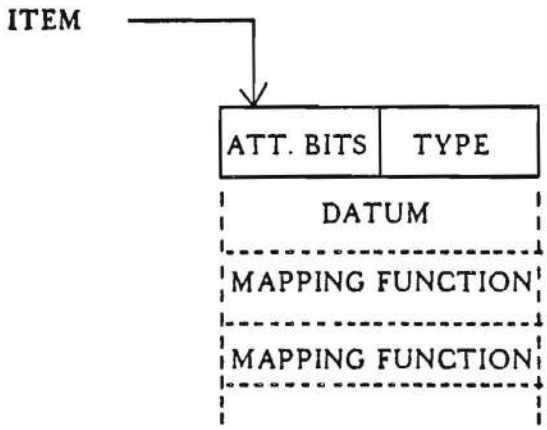

Figure 2

ONE-WAY LINKED LIST

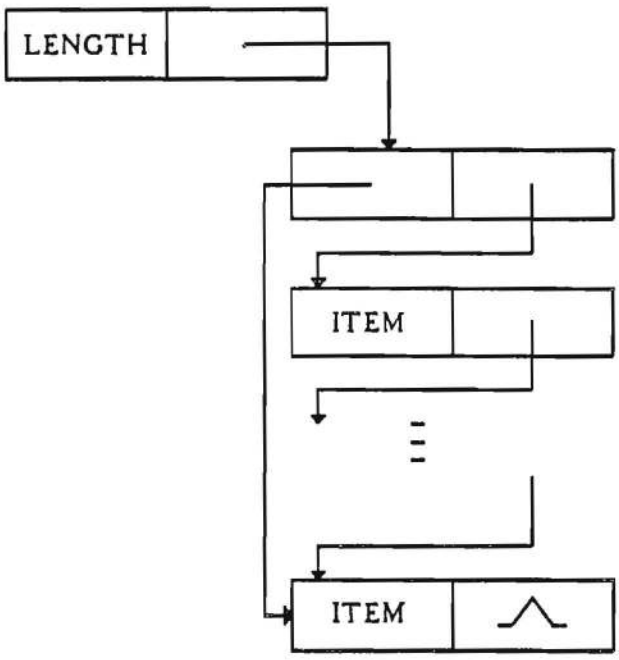

Figure 3

REPRESENTATIONS OF SETS

In each of the representations we have implemented, a null descriptor (0) is always a valid representation of the empty set. For certain representations, as noted below, there may also be additional valid representations of the empty set. In the following, the storage requirements are those for our implementation on the PDP-10 computer. Other implementations are likely to have similar storage requirements.

a. Sorted one-way linked list (See Figure 3). The descriptor contains the length of the set and a pointer to the first of a chain of one-word nodes. The first node contains pointers to the last node in the chain, and the next node in the chain. The remaining nodes in the chain each contain the 18-bit internal representation of an item and a pointer to the next element in the chain. The pointer field of the last element of the chain contains a null pointer (0). An empty set is uniquely represented by a null (0) descriptor.

   The additional storage (in words) occupied by a set is one more than the number of elements in the set unless the set is empty, in which case no additional storage is required.

b. Height Balanced Binary Tree (See figure 4) - The descriptor contains the length of the set and a pointer to the root node of an AVL (for Adel'son-Vel'skii, and Landis) tree; a binary tree with the property that at any node, the height of the left subtree differs from the height of the right subtree by at most 1 ([Foster65, Crane72, Knuth73]).

   Each (two-word) node contains the reference to an item, the balance factor (Left Heavy, Balanced, Right Heavy), and the pointers to the left and right subtrees (perhaps null). An empty set is uniquely represented by a null (0) descriptor.

   The storage required is twice the number of elements in the set.

c. Fixed Length Bitstring (also known as boolean array, bit array, bit vector). The descriptor contains the number of words making up the bit string and a pointer to a contiguous block of storage of that size. We pack 32 bits per word, rather than the available 36 so that indexing operations may be performed using shifts rather than divisions. The empty set is represented by a null descriptor or by a bitstring of all 0's.

   The storage required is ⌈MAXSIZE(SET)/32⌉.

d. Hash Table (See figure 5). The descriptor contains the length of the set and a pointer to a block of 33 words. In our implementation, we restricted ourselves to a single sized bucket hash table ([Morris68]). In a more advanced system we would expect a number of different sized tables to be available. In this implementation a hash function maps each item into a number between 0 and 31 corresponding to a word within the the block of 33 words. This word is a list head containing pointers to the first and last nodes of the chain of items (conflict list) that hashed to the same bucket. The other word of the 33 word block contains a mask (signature) ([Harrison72]) with a 1 bit on corresponding to every bucket with a non-empty conflict list.

   The storage required is zero for an empty set; otherwise, 33 plus the number of elements in the set.

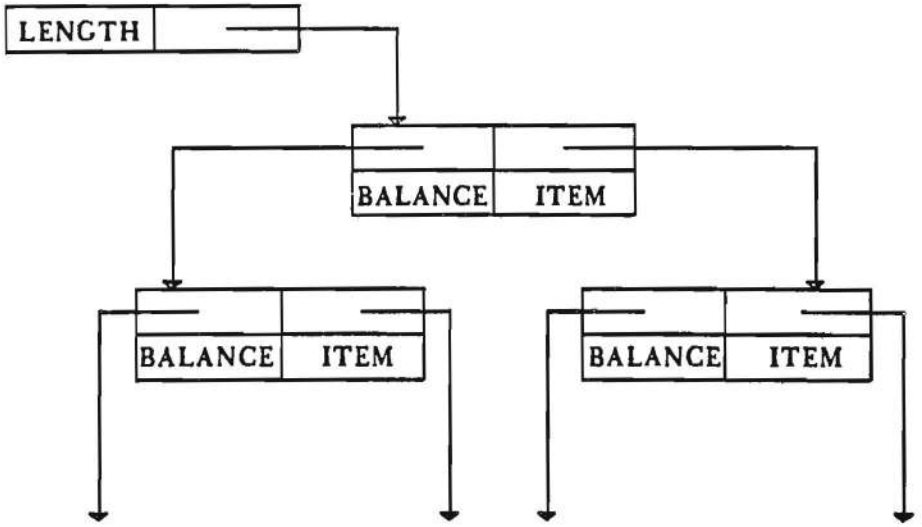

Figure 4

HASH TABLE WITH SIGNATURE

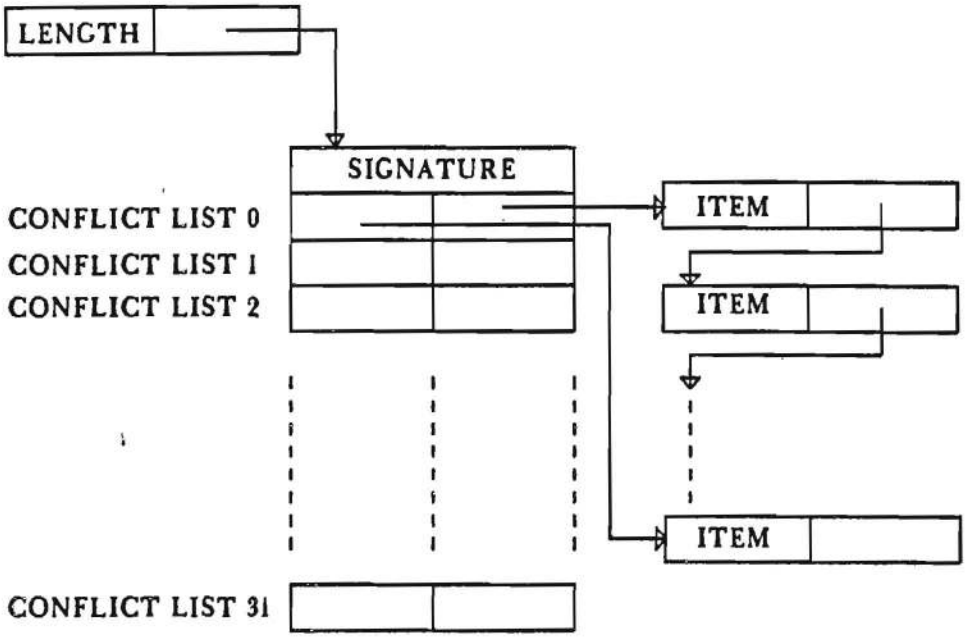

Figure 5

e. Fixed Length Sorted Array. The descriptor contains a pointer to a block of words (multiple of 8 in length) and the length of that block. The first word of the block contains the current length of the set. The next n words (where n is the length of the set) contain the items of the set in ascending order according to their 18-bit representation.

   The storage required is zero for an empty set; otherwise, five plus the number of elements in the set. This is not always accurate since our implementation will never decrease the size of an array. Thus, if the set becomes very large and then decreases in size it will continue to take up the larger amount of space.

f. Attribute Bit of item. The descriptor contains a number between 1 and 18. This corresponds to a bit position in the left half of the word pointed to by the internal representation of an item. This is an incomplete representation and can not be used if operations other than inserting an element into a set, removing an element from a set, or testing an element for set membership are required. Since our implementation uses a single halfword to contain attribute bits (see figure 2), a maximum of 18 sets per program may use this representation. There is no explicit representation for a null set. A null set is indicated by every item having a zero in the bit position corresponding to the set.

   As the storage occupied by this representation would go to waste (in our implementation this half word in each item is normally unused) if it were not used, we have associated a zero storage cost to this representation.

g. Combination of Fixed Length Bitstring and Unsorted Linked List (see figure 6). The descriptor contains a pointer to a two word block. This block contains descriptors of the form for representations (c) and (a) above, with the exception that the linked list is not necessarily in descending order of internal representations of items. A null set is uniquely represented by a null descriptor.

   The storage required is zero for an empty set; otherwise, three plus $\lceil$MAXSIZE(set)/32$\rceil$ plus the number of elements in the set.

In all of the representations above, except the boolean array, and the combination of boolean array and linked list, we use the full 18-bit pointer to reference items. With the boolean array representations we use small integer indices to reference items. Two translation functions are necessary to translate from the full 18 bit reference of items to the bitstring index (between one and the maximum size of the set) and vice-versa.

OTHER SET REPRESENTATIONS

There are many other representations for sets which we have not implemented. Any sequence representation may be used, since we may represent a set as an ordered or unordered sequence of items. Other possible representations include 3-2 trees (*B Trees*) ([Knuth73]) and linked items. The last is similar to our standard linked list representation, but the list actually runs through the items rather than through nodes pointing at them. With our standard representation we have a linked list of nodes, each of which points to an item. With the linked items representation, each item (represented much like a PL/1 based structure) contains a field which is a pointer to the next element of the set. Thus, each item tuple would have to have a field (see

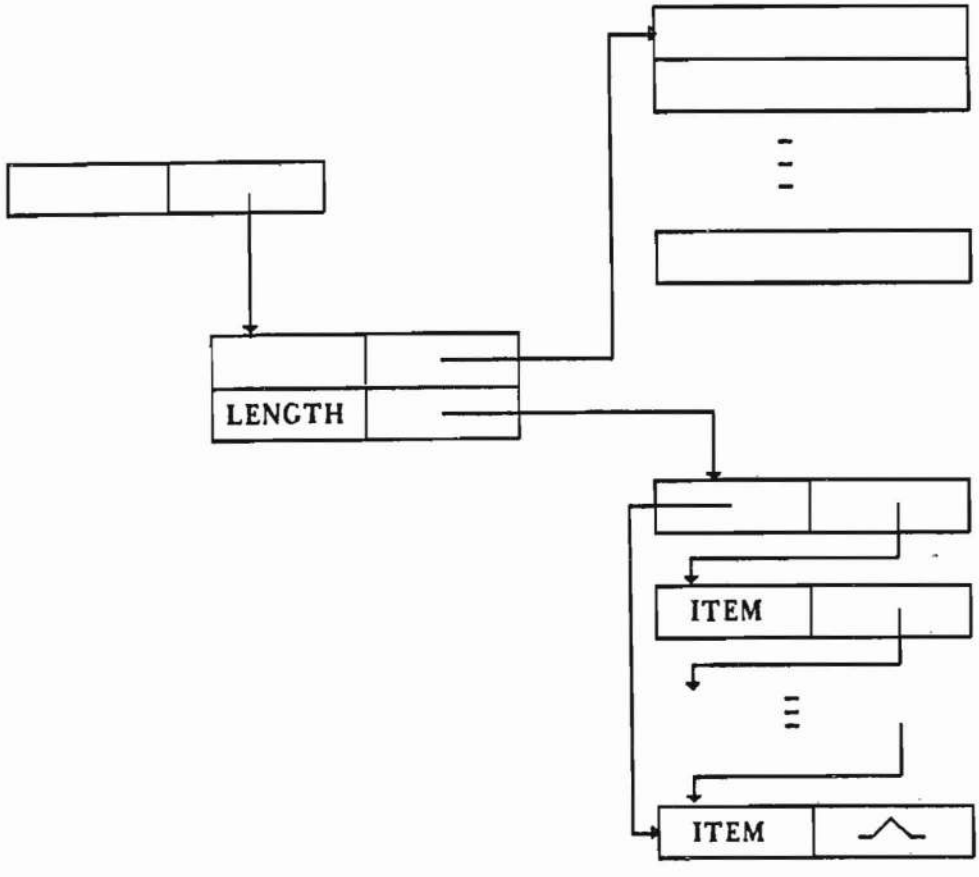

Figure 6

figure 2) for each set of which it might potentially be a member. Other forms of hash coding ([Morris68, Maurer68, Brent73, Feldman73, Knuth73]), such as linear probing or quadratic hashing may also prove beneficial for set representations.

3.2 LIST REPRESENTATIONS

LEAP lists are sequences of items. The same item may appear an arbitrary number of times in the list. The order of items is that imposed by the user's program. As with sets, we have chosen a set of primitive operations which are sufficient to do all the list manipulations available in our subset of SAIL, e.g., assignment, FOREACH iteration, concatenation, selection and removal based on index position. There are twenty primitive list operations (See appendix B for their description).

REPRESENTATIONS FOR LISTS

1. One-Way Linked List (see Figure 3).This is the same data structure as the one-way linked list used for set representation, except of course the order of items is not necessarily in descending internal representation, but is that imposed by the user's program.

   The storage (in words) required is zero for the empty list; otherwise, one plus the length of the list.

2. Two-way linked list (see figure 7). The descriptor contains a pointer to a two-word header node. The header node contains the length of the list and pointers to the first and last nodes of the list. Each two-word node contains an item, a forward pointer to the next node in the list, and a back pointer to the previous node in the list. The forward pointer of the last node and back pointer of the first node both point to the header node.

   The storage required is zero for an empty list; otherwise, two plus twice the length of the list.

3. Variable Length Array. This is the same as the data structure used for sets except again the order of items is that prescribed by the user's program and is not dependent on the internal representation of items.

   The storage required is zero for the empty list; otherwise (on the average) five plus the length of the list.

OTHER LIST REPRESENTATIONS

In addition to the representations mentioned above, we may use most of the representations used for character strings since our lists are really strings of items ([Madnick67]). Thus, we might also use fixed length arrays. A circular buffer is another representation which may be useful. In a circular buffer, we have a block of storage and two pointers to the beginning and end of the list. The block of storage is thought of as circular. That is, conceptually, the next element after the last element of the block is the first. Another interesting list representation is that of a height balanced binary tree ([Crane72]).

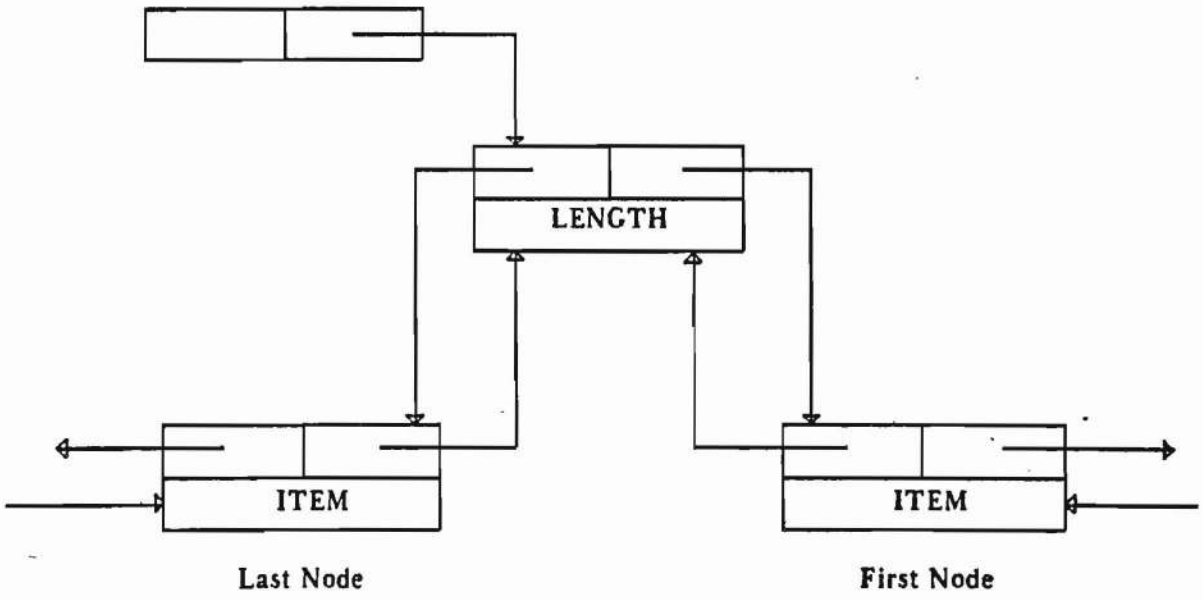

Figure 7

3.3 TERNARY RELATION

LEAP contains a single ternary relation. Relation instances are written $A \otimes O \equiv V$. A ternary relation can be represented by eight mapping functions.

1. Given	A , O, V	→→	true, false (does relation instance exist)
2. Given	A , O	→→	{V} such that A ⊗ O ≡ V
3. Given	A , V	→→	{O} such that A ⊗ O ≡ V
4. Given	O , V	→→	{A} such that A ⊗ O ≡ V
5. Given	A	→→	{<O,V>} such that A ⊗ O ≡ V
6. Given	O	→→	{<A,V>} such that A ⊗ O ≡ V
7. Given	V	→→	{<A,O>} such that A ⊗ O ≡ V
8.		→→	{<A,O,V>} such that A ⊗ O ≡ V

"<a,b>" is a meta notation the ordered tuple (a,b). The eighth function which produces the universe of ordered triples has not been implemented.

It is not common for the ternary relation to be used in the most general sense (i. e. needing all of the mapping functions). For most programs, any given triple (relation instance) may be referred to by only a small subset of the mapping functions. This is even more likely when we partition the single ternary relation into several disjoint ternary relations (See description of partitions of the ternary relation in Section 4)

To simplify the selection process by eliminating some redundancy, we wish to implement only the most specific mapping functions which can not be easily derived from other mapping functions which will be present. Let MF be the subset of the mapping functions which are needed for the user's program (this can be determined by a static analysis of the source program). Now remove from MF those functions which can easily be derived from others in MF. (1) can be derived from any of the others. (2) can be derived from (5) or (6), (3) can be derived from (5) or (7) and so forth. We now have MF containing three, two or one mapping functions. We will have a representation for each of these mapping functions. These mapping functions may be classified by the number of *bound* arguments they have. (5), (6), and (7) each have only a single bound argument. (2), (3), and (4) have two bound arguments, and (1) has all three arguments *bound*.

We immediately notice that by considering permutations of the components of the ordered triple representing the relation instances, we can always act as if the first; first and second; or all three positions within the triple were *bound*. This may mean we are actually keeping track of up to three different permutations, one associated with each mapping function within MF.

Before we actually list those data structures which we will use to implement these mapping functions, let us note that there are other ways of implementing relations which involve a single data structure which is good for several mapping functions. ([Rivest74, Delobel73]). Some hardware associative processors ([Minter72, Minsky72, Parhami72]) have even designed to directly implement associative retrieval.

PRIMITIVE OPERATIONS ON THE RELATION

The primitive operations necessary for the associative store are:

1. MAKE - create an instance the ternary relation. If more than one mapping function may potentially involve this triple MAKE will have to modify more than a single data structure. For example assume that mapping function (2) (A ⊗ O ≡ ?) and mapping function (7) (? ⊗ ? ≡ V) were required by the user's program. A make statement would have to update the data structures corresponding to each mapping function.

2. ERASE - delete an instance from the ternary relation. If more than one mapping function may involve the triples matching the pattern of the ERASE, ERASE will have to update more than a single data structure as with MAKE.

3. EXISTENCE - does a given relation instance exist. We will use the most specific mapping function which can answer the question. Again assume that we have mapping functions (2) and (7) available. To evaluate the boolean $A \otimes O \equiv V$ we would use the more specific mapping function, mapping function (2). Note that this is sometimes suboptimal. If there were only one $<A,O>$ pair for each V, but many V's for each $<A,O>$ we would be better off using mapping function (7).

4. FOREACH iteration - as with existence testing, we use the most specific applicable mapping function.

REPRESENTATIONS FOR THE MAPPING FUNCTIONS

The following are descriptions of how the various mapping functions may be implemented. Due to time constraints, this implementation was not completed in our demonstration system.

ALL THREE ITEMS BOUND

We have a block of storage 128 words long which we use as buckets for a hash table. We take all three items, hash them together to form an index between 0 and 127. This gives us an position in the hash table of a conflict list. Each two-word node in the conflict list contains the three items of a relation instance and the pointer to the next node in the conflict list.

The storage (in words) required for this representation is thus 128 words plus twice the number of relation instances.

TWO-THINGS BOUND

Hashing

We hash the two *bound* items together to get an index into a 64 word bucket hash table. This gives us a linked list of three word nodes, containing the two bound items, the pointer to the next node in the conflict list, and a set descriptor for the set of items which are the third component of relation instances with the two bound items.

In our design, there are three variants of this which differ in the representation used for the set of *third components*. Apart from the storage used for the set, the storage requirement is 64 plus three times the number of different pairs <first component, second component> in the data structure. Using a sorted linked list as the set representation, we need an additional word per <first component, second component> pair and one word per relation instance. Using an height balanced binary tree, we additionally need two times the number of relation instances. Using a fixed length bitstring, we need an additional n words, where n is $\lceil$number of possible third components/32$\rceil$ per distinct <first component, second component> pair.

Field Selection

We use the second item to select a field (offset) to the structure of the first item (See figure 2). The field contains a descriptor of standard set representation. We need a translation function which translates the 18 - bit representation of the second item into a field index. (In our design this function must be executable at compile time).

The base storage requirements are the number of possible first components times the number of possible second components. The additional storage needed for the set of third components for each active <A,O> pair may be calculated the same way as was done with the hashing representation above.

ONE-THING BOUND

This mapping function has an ordering to the components of the triple. The *first* item is the one which is always specified, the *second* item will be the one next most often specified, and the *third* will be the least often specified.

We use the *first* item to find (via hashing, or sorted linked list) the head of a sorted linked list of ordered pairs consisting of the second item and a set of the third items.

The storage required is the number of distinct *first items* plus initial hash table (if any), plus twice the number of *second items* per *first item*, plus the storage necessary for the set of *third items*.

3.4 ESTIMATING THE EXECUTION TIME OF PRIMITIVE OPERATIONS

We have previously stated that there are three classes of information that our selection phase needs to know about the representations available. The first, applicability, is found by looking at which primitive operations have actually been implemented using this representation. If any primitive operation is used by the user's program which has not been implemented, then this representation is not applicable. The second piece of information, the storage cost function, has been treated in the previous sections. Here we will discuss the third class of information, prediction of the execution time which would be used by the routines implementing the primitive operations.

We believe in precise analysis of program segments. It is not enough to know that one routine takes time proportional to the size of a data structure and another takes time proportional to the log of the size of the data structure. The proportionality constants and any other constant overheads are important in making a wise choice between the two routines.

To demonstrate the importance of knowing the precise proportionality constants let us consider two subprograms which are logically equivalent. The expected execution time of the first is

$$19 + 12 * \log2(\lambda)$$

time units, where λ is the number of data objects , and the expected execution time of the other is

$$40 + 5 * \lambda/32$$

time units. Which should we choose? The answer clearly depends on λ, the number of data objects. If λ were only 2, the first program would be expected to take 31 time units and the second slightly over 40 time units, so we would choose the first. If λ were 32 the first program would be expected to take 79 time units and the second would take only 45, so here the second is superior. If λ were 1024 the first program would be expected to take 139 units, and the second , 200 so the first program would again be superior. The only way we can determine this analytically, is to have precise knowledge of the constants 19 and 12 for the first algorithm and 40 and 5/32 for the second. The problem to be solved now is, how do we determine the constants, and the very dependencies on $\log2(\lambda)$ or λ?

Two methods are apparent. One is to simulate the routines on a large number of collections of random data of different sizes then use statistical techniques to derive the dependencies on functions of size and the corresponding constants. This technique is valid only for a large enough sample, and we would not expect to be able to simulate in a large range. The results would thus be valid only in a small range.

The other method, which we have chosen, is to analyze mathematically the various subprograms which are used to implement the primitive operations using the techniques of Knuth (vol I-III).

We are interested in the application of the analysis of algorithms techniques to the concrete subprograms to determine both the order dependence (an order n^2 or order $log\ n$ algorithm) and the proportionality constants of all terms in the execution time cost function.

Clearly any results we obtain are not of theoretical importance (i.e. no claim is made that they reflect the minimum amount of time necessary to accomplish the primitive operation on the given data structure) but reflect only the expected time using our particular encodings of the primitive operation. A better encoding of a given routine that may reduce the proportionality constants, or even order dependency may be found in the future.

We wish to stress again the importance of precisely determining the proportionality constants as well as the order dependency. When the size of the data structures is small a $1000\ n$ encoding will be worse than a $10\ n^2$ encoding. The running time of a program which uses these subprograms depends on the actual number of machine instructions executed within these subprograms not on some order dependency such as n^2.

Our basic method is to weight each subprogram statement with its expected execution time, multiply that by the number of times it is expected to be executed (normally a function of size of data structures etc), and then sum these numbers over all the statements of the subprogram.

We have chosen to use assembly language to implement all the primitive operations on our representations. We have a much better idea as to how much time a machine instruction is expected to consume than how much time a statement in a higher level language will take (See appendix H). There has been some work on predicting execution time of ALGOL, FORTRAN and LISP programs ([Wichman72, Ingalls71, Knuth71, Wegbreit74]) but current methods are still very crude. As demonstrated above we must know precisely the relevant proportionality constants in order to determine whether given representations are better or inferior to others.

Our method of calculating execution time cost functions is thus to associate with each instruction a weight proportional to its average execution time. We also associate with each instruction the number of times the instruction is expected to be executed as a function of the size of data structure and perhaps other parameters. Finally we take the sum of the products of instruction weight and number of times the instruction is executed, to construct the time function.

For example, consider the routine for fetching the Nth item in a sequence using the one-way linked list representation. (My apologies to those who are not familiar with PDP-10 assembly language. Hopefully, the comments will help). Let *INDX*, *LPTR*, and *RESULT* be symbolic names for accumulators. Also, assume that the list index is in the variable named *N*, and the list descriptor is in the cell named *THELIST*. The numbers to the right are the weights we have assigned to the individual instructions.

```
        move indx,n          ;LOAD N INTO ACCUMULATOR INDX          ; 3
        move lptr,thelist    ;LOAD THE DESCRIPTOR INTO              ; 3
                             ;   REGISTER LPTR
lp:     hrrz lptr,(lptr)     ;GET POINTER TO NEXT NODE IN LIST      ; 3
        sojg indx,lp         ;DECREMENT INDEX, IF GREATER THAN      ; 2
                             ;   ZERO GO TO LP.
        hlrz result,(lptr)   ;FETCH THE DESIRED ITEM                ; 3
```

We immediately see that the two *MOVE* instructions and the *HLRZ* instruction are each executed only once. The two instruction loop will be executed *N* times, where *N* is the list index. As it is not clear here what *N* is, we will assume that it is equally likely to be $1, 2, \ldots \lambda$, the length of the list. The average value of *N* is thus easily determined as $(1 + 2 + \ldots \lambda)/\lambda = (\lambda+1)/2$. Therefore, the average number of executions of the *HRRZ*, *SOJG* loop is also $(\lambda + 1)/2$.

Our predicted time for this routine is thus:

$$3 + 3 + 3 * (\lambda + 1)/2 + 2 * (\lambda + 1)/2 + 3 = 11.5 + 2.5 * \lambda$$

Our analysis was dependent on the assumption that the indices were randomly distributed over all possibilities. However, if the user were actually using the sequence as a model for a stack, he might be biased toward the lower or higher indices only, thereby invalidating our assumption. Other forms of analysis of programs can give us worst case estimates if need to worry about the maximum program execution time. , perhaps because of real time constraints. Our analyses give us average estimates which we feel are the most generally useful. We feel that the real answer to problems involving biases in the use of data structures, either explicit semantic level ones as above, or hidden ones that result from interactions involving internal representations, is careful monitoring. Both pre-selection monitoring (using default representations) and post-selection monitoring should help us alleviate problems involving such gross biases. We would expect future execution time estimators to have more parameters. In the above case we would want the estimator to depend at least on the average index value, if not the distribution of index values.

We have performed analyses (average case) like the one above on all the routines implementing the primitive operations on our representations. (See APPENDIX I for our execution time cost functions.)

SECTION 4

INFORMATION GATHERING

A system for selecting data representations must have information about representations and about the use of the abstract data structures for which it is going to choose representations. In the previous chapter we discussed two techniques for obtaining information about the time functions of the primitive operations as implemented for given representations: simulation and mathematical analysis. We will now discuss techniques for obtaining information about the use of the abstract structures of the user's program.

The information we wish to obtain includes which primitive operations are performed, the expected sizes of the abstract structures when the various primitive operations are performed, the values of the other parameters of the time functions of the primitive operations, and partitions of the information structures into classes.

It is common within a program to have many different abstract data structures of the same type. Often, it is not necessary or desirable to represent all these structures the same way. We may find that a program uses two structures of the same abstract type quite differently, even employing different sets of primitive operations on them. If we are forced to represent both structures the same way, we end up with a compromise representation that is probably inferior to ones we would have chosen if we had approached each structure independently. However, there are also good reasons for representing some groups of data structures the same way. We find that with many operators we can take advantage of the internal structure to obtain more efficient implementations. For example, assume we wished to add the elements of one set to another, e. g.,

$$SET1 \leftarrow SET1 \cup SET2;$$

An obvious way of implementing this would be to iterate though *SET2* and insert those elements into *SET1*. If we were representing *SET1* by a sorted (on the internal representation of data objects) linked list, this implementation would take a time proportional to the product of the number of elements in *SET1* and the number of elements in *SET2*. If, on the other hand, both sets were represented the same way, as sorted linked lists, we could traverse both lists in parallel and accomplish the union in time proportional to the sum of the lengths of the two sets. In theory we could have a different union implementation for every pair of possible input representations, and every output representation but the cost of this is very large in terms of the size of the library of implementations of the primitive operations. If we had ten set representations we might need five hundred (ten cubed divided by two since union is commutative) union routines. The other alternative is to have only one union routine per representation, and translation functions which change the representation of a set from one representation to another. To perform the union, we would make sure both arguments were in the same representation and then use the representation dependent union routine. These translations are usuallly expensive operations to perform, yet we wish to use the highly efficient representation-dependent routines. Our solution, though admittedly suboptimal, is to avoid the expensive translation operations by insisting that both operands to an operator be kept in the same representation. Note that even if we were willing to have such large libraries or translations of representations we might still wish to insist that certain information structures be represented the same way. One reason might be to avoid runtime representation checking of procedure parameters when they appear as operands to primitive operations.

We have chosen to require that all information structures which are operands to an instance of an operator, or structures acting as the same positional parameters to a procedure, be in the same equivalence class. This gives us the smallest classes possible that retain the property that there is no conversion necessary from one representation to another. This also gives us the property that no dynamic determination of representation is necessary and the proper routine for a primitive operation can be totally specified at compile time. Determinations of the tradeoffs involved in having translations of data structures from one representation to another, as well as benefits versus costs of dynamic representation checking are left to future research.

One of the goals of the information gathering phase is thus to find the information necessary to partition the set and sequence variables into equivalence classes.

There is another important use of partitioning. This occurs when we wish to split a single information structure into several. Let us consider the ternary relation as simply a set of triples. It is often possible to view this set as the union of several disjoint sets of triples. This separation is useful if each update or access of the associative store in the user's program refers only to one of the disjoint subsets. Now, instead of having one large data structure we have logically several disjoint data structures which are independent of one another. It is quite likely that each of these is *less general* in terms of the primitive operations performed on it than the original data structure. We will thus be able to tailor a representation for each of the smaller structures that is likely to be more efficient than the compromise representation we would have had to choose for the original structure. Note that *splitting* in general may be a useful concept in the representation of high level data, and often it may not even be necessary that the results of splitting be disjoint (No attempt is made in this system to *split* sets or sequences). One of the goals of our information gathering phase will be to find information which will let us later compute a natural disjoint *split* of the ternary relation.

In general, several techniques are applicable to the problem of information gathering. First, we may let the user give us the required information. This may be done by requiring the user to make declarations or assertions about his program (this is not done in our demonstration system). This information can be augmented by an interactive session in which the system asks the values of additional parameters which had not been declared. This method has the obvious drawback that the user often does not have such information. With the current state of the art there are many programs which can not be analyzed mathematically to the detail we require. Our choice of data representation will be determined by this information. If dependent only on crude *guesstimates* our choice will be just as crude. Another reason for not depending on the user is that a voluminous amount of information is required. An enormous amount of patience is required to sit at a terminal for many minutes (hours?) to answer detailed questions. A good strategy is to ask the user only when the system is not able to determine a piece of information in any other way. During the evolution of a system like ours, this should require the user to provide less and less information directly.

A prime means of obtaining information is the use of monitoring. The user runs his program with his own sample input data with the system providing default representations for the abstract information structures. A special compiler and runtime environment is used which is geared to collecting statistics about the use of the information structures. The only statistics we gather in our system concern the number of times each construct of the user's program is executed. Other information which we could easily obtain in a production system would be the distribution of sizes of the information structures at particular places in the user's program (e.g every time a

primitive operation is performed), and the parameters of our time functions for the primitive operations. In our current system we ask the user to provide this information during a question-answer session.

The other technique which we depend on is a *static analysis* of the program. In our system this takes the form of a *meta-evaluation* of the program by actually following all possible paths of control and computing the possible contents of variables and the associative store.

4.1 EXAMPLE OF INFORMATION GATHERING

Before we describe the last technique of information gathering, static analysis, let us see how the information gathering portions already described would process our transitive closure procedure.

```
set procedure TRANSCLO (itemvar REL; set BASE);
begin "TRANSCLO"
      set RELATED, NEWLYRELATED, FOUND; itemvar X,Y;
      RELATED ← phi; NEWLYRELATED ← BASE;
      while (NEWLYRELATED ≠ phi) do
         begin
               FOUND ← phi;
               foreach X,Y | X ∈ NEWLYRELATED ∧ REL ⊗ X ≡ Y do
                     put Y in FOUND;
               RELATED ← RELATED ∪ NEWLYRELATED;
               NEWLYRELATED ← FOUND - RELATED;
         end;
      return (RELATED);
end "TRANSCLO";
```

We first have a monitoring phase. The above procedure is compiled with a special compiler which inserts counters before every statement and every FOREACH *iterator*. We obtain a count of how many times each construct was executed by monitoring a sample run of the program using the user's own input data with the system supplying default representations for all the LEAP data structures.

The system then asks the user many questions concerning the average size of various data structures at particular program points. In the above example the system asks for the average size of *BASE* when used in the assignment statement, the average size of *NEWLYRELATED* in the equality test and the probability of it being empty, the average size of *NEWLYRELATED* in the foreach search and again the probability of it being empty, the size of *FOUND* at the *PUT* statement, the size and probable overlap of *RELATED* and *NEWLYRELATED* at the union statement, and so forth. Most of these statistics could be more easily obtained if there existed special versions of the data structure manipulation routines used during the monitoring phase which actually recorded the information necessary to compute these statistics. Even in the absence of such mechanisms we could possibly decrease the number of questions the user is asked by doing some inference on the program. For example, above we mentioned that the system asks the user the probability of *NEWLYRELATED* being empty at the

FOREACH search. This question is clearly superfluous because the probability can be inferred to be zero from the conditional at the top of the while loop. Other inferences could be made on the basis of certain operations not changing the size of their operands. All these inference techniques are left to future research.

4.2 STATIC ANALYSIS OF LEAP PROGRAMS

PRIMITIVE CLASSES OF ITEMS

A primitive item class consists of either a single declared item or all the items potentially allocated from a single source language call to the heap allocator (*NEW*). This is the finest grain to which we can partition all the items in the user's program. There is essentially no way, at present, to distinguish between different items produced by the same source language call to *NEW*. We will use these primitive classes to model the contents of variables and the associative store. A relatively small number of primitive classes, can be used to take the place of the usually much larger (and sometimes indeterminate at compile time) number of items actually present during the execution of the user's program.

Meta-evaluation

Meta-evaluation, as we use the term, means a simulation of the user's program symbolically rather than with real data. In our system we express the values of LEAP variables as sets of primitive item classes.

Let us consider the transitive closure procedure we looked at earlier and note how we would process it, before we give the actual static analysis algorithm in detail.

```
set procedure TRANSCLO (itemvar REL; set BASE);
begin "TRANSCLO"
      set RELATED, NEWLYRELATED, FOUND; itemvar X,Y;
      RELATED ← phi; NEWLYRELATED ← BASE;
      while (NEWLYRELATED ≠ phi) do
         begin
               FOUND ← phi;
               foreach X,Y | X ∈ NEWLYRELATED ∧ REL ⊗ X ≡ Y do
                     put Y in FOUND;
               RELATED ← RELATED ∪ NEWLYRELATED;
               NEWLYRELATED ← FOUND - RELATED;
         end;
      return (RELATED);
end "TRANSCLO";
```

To begin processing we form a flow graph of the program which we will then analyze. (See figure 8 for the flow graph corresponding to this procedure.)

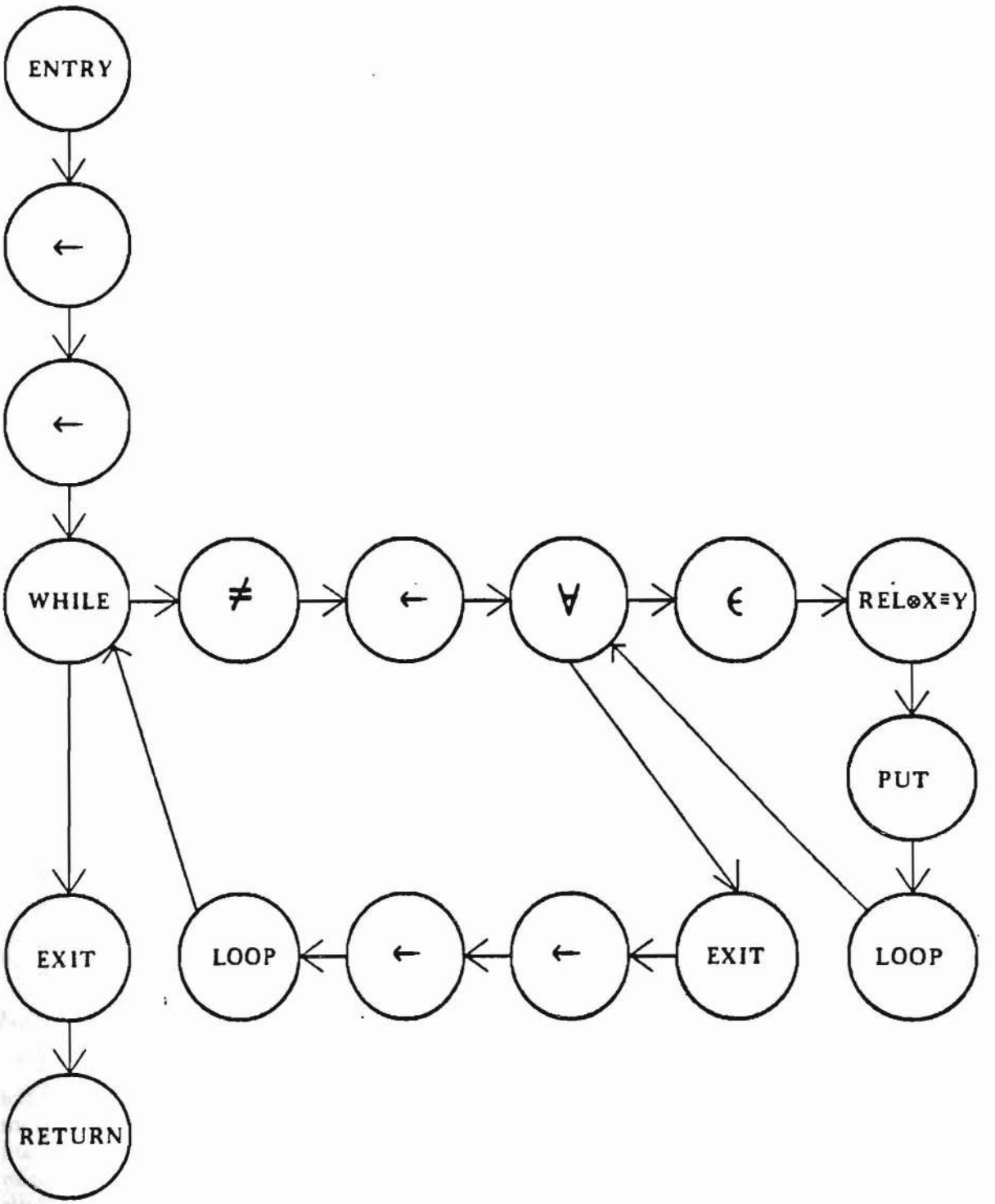

Figure 8

We only arrive at the procedure entry point via having encountered a call to this procedure in the flow graph of the program in which this procedure is found. We will therefore know what the possible values of *REL* are in terms of the primitive item classes. Similarly we will know the possible elements of the set *BASE*, as well as having a model about what associations can possibly exist, all in terms of the primitive item classes.

When we encounter the first two assignment statements, we attach to the corresponding flow graph nodes, the appropriate information. In this case, at the first assignment we know that the set *RELATED* will be given the value **PHI**, i. e. that its set of possible values in terms of the primitive item classes is empty. Similarly, at the second assignment node we can determine the possible set of values for *BASE*, and we will attach that same set as the potential set of values for *NEWLYRELATED* at this node. In general we keep track of the nodes at which either changes to the value sets of some variable occur, or where the value sets of all variables are known. We will explicitly compute all the value sets at control points. This will allow us to determine the possible set of values for any variable we encounter in the program.

We next encounter the while loop node. This is a control point so we will attach to this node our knowledge as to the possible contents of all the variables. In this case it means merely attaching the value set of the variable *REL*, as copied from the entry node; the value set of *BASE*, as copied from the entry node; *RELATED*, as copied from the first assignment node; and the value set of *NEWLYRELATED* as copied from the second assignment node.

We now encounter the equality test, *NEWLYRELATED* ≠ **PHI**. This statement has no effect on the value sets of any variables so it is essentially treated as a null statement. In a more advanced system, other booleans could possibly alter our views as to the value sets of certain variables. For example, if we had the expression $X \in SETVAR$ then on the *true* path (if this were an if statement conditional), we would know that the possible value set for *X* could only be our previous value set for *X* intersected with the value set of *SETVAR*. Our system as currently implemented, makes no such use of boolean expressions.

The next node encountered is the assignment *FOUND* ← **PHI**. We treat this the same way we did the first assignment node.

Now we come across the FOREACH loop. We treat the search $X \in NEWLYRELATED$, much like an assignment. The value set for *X* at the foreach node is thus the same as the current value set of *NEWLYRELATED*. The search $REL \otimes X \equiv Y$ is slightly more complicated. We know the value sets of *REL*, and *X*. We then use our model of what possible associations exist, to compute the possible value set of primitive items which could be assigned to *Y* by this pattern match.

The put node will take the union of the value set of *Y* and the value set of *FOUND* and make that the new value set of *FOUND* at this node.

We now encounter the continue node for the foreach. We notice that during the execution of the foreach node, value sets for certain variables were changed. We must therefore simulate the loop again until a fixed point is reached; that is, until no new primitive items are added to any of the value sets of the variables at the various nodes within the loop (The reason why a fixed point is always reached will be discussed with the description of the static analysis algorithm). This guarantees that we will correctly compute the possible sets of values for variables

after loop exhaustion. As an example of why simulating until reaching a fixed point is necessary consider:

```
SET1 ← { a };
COUNT ← 1;
SET2 ← { b };
while COUNT < N do
      begin
            COUNT ← COUNT + 1;
            SET2 ← SET2 ∪ SET1;
            SET1 ← { c };
      end;
```

If we had merely simulated the while loop once, the possible set of values for *SET2*, when the loop is exhausted, would have been computed as { *a* , *b* } which of course is incorrect. By insisting that we have a fixed point we will simulate the loop three times and obtain the correct result { *a*, *b*, *c* }.

The union node is easily handled. The new value set for *RELATED* is the union of the old value sets of *RELATED* and *NEWLYRELATED*. We cannot tell if the set difference will find any values in common, so we take the conservative approach and assign as the new value set of *NEWLYRELATED* the current value set of *FOUND*. In general if we err in a computation of a value set, we want it to be on the side of being larger than necessary. This may force the system to later choose a slightly less efficient representation, but it also restrains the system from choosing a representation which is not applicable (i. e. doesn't provide essential primitive operations).

We now come across the continue node of the while loop. Just as in the case of the FOREACH loop we must continue processing until we reach a fixed point. So we again return to the while node. At this point we have to insert in the value sets of the variables *FOUND*, *RELATED*, and *NEWLYRELATED* all the elements of the corresponding value sets at the end of the while loop.

Eventually we will obtain a fixed point and can then reach the return node. Here we will take the current value set of *RELATED* and make it the value set of the procedure. This is the value set that the caller of the procedure will use in its meta-evaluation.

4.3 STATIC ANALYSIS ALGORITHM

The first step is to form a graph of the user's program. As our system is only dealing with the choice of data structures, we do not include constructs from the user's program which are totally devoid of LEAP statements and expressions in this graph. For simplicity, we have also outlawed the *go to* in this system, thus giving our program graphs a nicely nested structure.

We will implicitly associate with each node (expression, or statement) of the program graph, a set of primitive item classes (the value set) for each variable of the user's program which is accessible at that node. The value set for a given variable at a given node will eventually contain all the primitive items which that variable could possibly have at that point of the user's

program. We also maintain a model of the associative store (ternary relation) in terms of what relation instances can exist between the primitive item classes. This model contains all associations which could exist at any point in the program. We do not have separate models of the contents of the associative store at every node of the program.

Before describing the details of the meta-evaluation process, let us define a some terms. A *value changing* node is a node where assignment is done to some LEAP variable: itemvar, set or list. This may either be an explicit assignment statement, or some FOREACH search. A *control node* is a node representing a control point in the user's program. This can be the beginning of a loop, exit of a loop, if-then-else node, case node, join node (node immediately following case or if-then-else), procedure entry node, and so forth. We keep a list (*NODELIST*) consisting of the most recently encountered control node, preceded by all the value-changing nodes encountered since that control node. The value sets (as constructed so far) of every known LEAP variable are associated explicitly with each control node. With each value changing node we associate the value sets for all LEAP variables whose values were possibly changed by that node. Thus, as we encounter any node in the program, we can find the value-set associated with a given variable known at that point by chaining back the *NODELIST* until we find the first node which has a value set for that variable. If there are no nodes in the *NODELIST* which have value sets for the variable, then we know that this is the first encounter with that variable so it has a null value set so far.

To begin the meta-evaluation, we initialize *NODELIST* to contain the program entry point. Now we traverse the program graph nodes in the natural order. As we come to any set, list or item expression we can compute the possible value set of that expression by substituting the value sets of the constituent components of the expression and using some special rules.(See META-EVALUATIONS appendix C) For example we might encounter the expression,

SET1 ∪ datum(SETITMVR)

We can directly compute the value set for the set variable *SET1* by the algorithm given above. To evaluate the value set for datum(*SETITMVR*) we must first compute the value-set for *SETITMVR*, and then form the union of all the value-sets of the datums of the primitive item classes in the value-set for *SETITMVR*. Now that we have the two value sets for *SET1* and datum(*SETITMVR*) we simply take the union of them to get the value-set for the entire expression.

At any value changing node we will compute the new value set and either merge that with the existing value-set (from *NODELIST*) for this variable (s) or make it the value set for the variable at this node. We then add this node to the front of *NODELIST*.

At a *MAKE* node we evaluate the three item-expressions and insert the appropriate ternary relation instances into our model. *ERASE* nodes are ignored during this phase.

When we encounter any *control node*, we form explicit value-sets for each variable known at that point. We do this by stepping through the *NODELIST* and finding the value sets for each variable known. We then merge these value-sets with those already present (if any) for these variables at the control node. After this is complete we throw away *NODELIST* and make a new one consisting only of this control node.

When we encounter a branching structure in the program (case, if-then-else) we stack the current status, follow one branch to its completion (the join node following the case or if). Then we pop the status (*NODELIST* etc) and traverse any remaining branches at this level. Join nodes are control nodes, therefore, our model of the contents of each variable contain the union of the models resulting from traversing each branch.

Loops are handled in a similar way to branching structures with one exception. When we reach the end of a loop, we note whether any value sets have been changed during the simulation of this loop, or whether any associations have been added to our model of the associative store. If there have been any such changes then we simulate the loop again. Note that this process is guaranteed to terminate because: we never remove things from the value-set of a variable at any node; we never remove associations from the associative store; there are only a finite number of variables (standard, datums of primitive item classes); there are only a finite number of primitive item classes. These all combine to give us the knowledge that we can only add primitive item classes to the value-sets, a finite number of times. Thus the loop simulation always terminates. Though this process is finite, and in most of the test cases we have processed the execution time is not more than we're willing to spend, static analysis of loops is potentially very costly. We continue to look for ways in which the cost of this analysis can be reduced.

Procedure calls are handled in a straight-forward manner. We treat value parameters as if they were variables which had been assigned as their initial values, the values of the actual parameters at the procedure entry point. Reference parameters are slightly more complex. We must keep track of the set of variables which they could represent. Except for these minor distinctions and the fact that we have to remember where to continue simulation after procedure exit, we treat procedure calls just as if they were simple in-line blocks of code. This method does not allow recursive procedures. There is no condition implemented which would cause the meta-evaluation process to terminate for recursive procedures, as each time it encountered a self call to the procedure it would suspend its current evaluation and start to evaluate the recursive call. This process would continue indefinitely. Thus our demonstration system outlaws recursion. One condition which could cause termination in future systems is to require that we not simulate any procedure if there is a pending simulation of the same procedure with the same state (all parameters have the same value sets, and no associations have been added to our model of the associative store).

We have now mentioned all the information gathering which is performed in the system, monitoring, user interrogation and static analysis. Each of these provides essential information which will be used in the next phase, selection of representation for the individual information structure classes.

SECTION 5

SELECTION OF DATA STRUCTURES

5.1 CRITERIA

Whenever we pick one representation of data over another, we need to have reasons why we consider that representation to be superior (or at least not inferior) to the other representation for the given purpose. The most common considerations used in such decisions involve the amount of storage space (working set size in a virtual memory) occupied by the data structures, and the execution time (cpu time) necessary for performing all the access and updates to the data structures. Also involved are the programming costs (original design and debugging) of implementing the representation. In this system we are totally ignoring these latter costs, because all the representations which are candidates for selection are fixed in advance, and we have already programmed and debugged the library of their implementation.

We need some way of quantifying how good a given set of representations are in comparison with other sets of representations. If one takes both less space and less time it is clearly superior. However, what if no set of representations satisfies this? We must have some way of predicting a *cost* for running the program with the different representations, and then we shall pick the representation whose expected cost is least. One way of associating such costs is an objective cost function whose parameters are time and space occupied by a program.

5.2 COMMON COST FUNCTIONS

Let SMAX be the maximum amount of storage which may be necessary during a program's execution.

Let TTOT be the total running time of the program

Let S(T) be the actual storage in use by the program at time T, $0 \le T \le$ TTOTAL

1. Minimum space

$$\text{COST} = \text{SMAX}$$

Choose the set of representations which will use the least space.

2. Time *(function of space)

$$\text{COST} = \int F(S(t))\, dt$$

A. F(S) = if S < CORESIZE then 1 else ∞

Cost of program is the time required as long as it fits in the partition allotted to it.

B. F(S) = if S < CORESIZE then S else ∞

Cost of program is the time * space required

C. F(S) = IF $0 < S \leq S_1$ THEN C_1
ELSE IF $S_1 < S \leq S_2$ THEN C_2
.
.
.
ELSE IF $S_{N-1} < S \leq S_N$ THEN C_N
ELSE ∞

We have here a step function, $C_1 \leq C_2 \leq C_3 \ldots \leq C_N$ are constants

D. F(S) = IF $0 < S \leq$CORESIZE THEN S^2 ELSE ∞

Quadratic in memory size.

3. Time * function(maxspace). Multiprogrammed systems will often require the user to specify in advance the maximum storage size he will use. The cost is then the cpu time multiplied by a function of maximum storage size.

In addition, other constraints may be placed on the representations. For example, in a real-time system (e.g. process control), we may have the restriction that certain operations must never take more than some fixed time.

Virtual memory systems with their pages and segments lead to other cost functions which may be described in terms of average working set size, maximum working set and so forth. These quantities are very difficult to predict with the current state of the art. In the sequel we are therefore considering only *real memory* or single-segment systems.

Our computer runs under an operating system whose costs are related to space multiplied by time so selection 2 (B) above will be the objective cost function which we will attempt to minimize. An important fact to note is that we have not assumed in the rest of the system the form of our objective function. It is therefore rather easy to change this to some other cost function without modification to the other parts of the system.

5.3 PARTITIONING THE INFORMATION STRUCTURES

After we have exhaustively processed the program graph in the information gathering phase we make one more pass to partition the set and list variables into their appropriate classes. Recall our requirement that two variables be in the same equivalence class if they are either operands to the same instance of a binary operator, or they are the actuals to the same formal parameter of some procedure. We also mark every set or list binary operation node of the

program graph with the *name* of the partition to which its arguments belong.

We now partition the primitive items into disjoint item classes. Just as we wish to avoid dynamic representation checking with sets and lists we do not wish to have to dynamically check the representation of items. In our system all items are represented by dynamic records, but the length of these records may differ depending on whether the item in question has a datum, if we use the offset representation for associations involving the item, and so forth. Our criteria for putting two primitive items in the same class is that they are in the same value-set for some variable at some program node or they are elements of the different value-sets for the same list or set variable. Having these disjoint item classes will make the job of implementing representations involving transformation of item representation (such as in the bit vector representations of sets) much easier. We will not have to worry about having different transformations for different uses of a single item.

After partitioning the items, we partition our model of the associative store into several different ternary relations each of which will contain associations between the disjoint item classes. For example, consider:

item class1: { *ITEM1 , ITEM2, NEW(scan 20)* }
item class2: { *ITEM3* }
item class3: { *ITEM4, ITEM5* }

where *ITEM1, ITEM2, ITEM3, ITEM4, ITEM5* are declared items and *NEW(scan 20)* is the primitive item class for the call to *NEW* at the 20 th token in the source program.

If we see that there are possible relation instances:

ITEM1 ⊗ ITEM3 ≡ ITEM4
ITEM1 ⊗ ITEM3 ≡ ITEM5
ITEM4 ⊗ ITEM1 ≡ ITEM3
ITEM4 ⊗ ITEM2 ≡ ITEM3

we can classify them into two classes, modeled by

class1 ⊗ class2≡class3
and
class3 ⊗ class1≡class2

Each operation on the associative store within the user's program can refer to only one of these classes (otherwise we would have had the merger of two item classes into a single item class). Therefore we have achieved the split of the ternary relation into *smaller* disjoint ternary relations.

5.4 APPLICABILITY FILTERING

Our system now has partitioned the set and list variables into equivalence classes and *split* the ternary relation so we must now begin the selection process. We can immediately eliminate some representations from further consideration because they do not provide certain of the primitive operations required by the user's program for that class of abstract data structure.

5.5 COST PREDICTION

We now will predict the time requirements and space requirements for each data structure using each possible representation. It is clear that the size of a high-level data structure varies over time. That is, the space for a data structure increases and decreases over the execution of a program. However in this first approximation let us act as if the size of the abstract data structure were just the average size over its lifetime. This average can be approximated by taking the average of the average sizes of the data structure at each primitive operation on this data structure. In our demonstration system we have these average sizes of the data structure from information given to us by the user. In a production system this information would be gathered by monitoring as well.

The execution time required by primitive operations on this information structure class can be predicted by simply processing each primitive operation node of the program graph referring to this class in the following manner. First use the values of the size and other parameters in evaluating the time function using this representation for the given primitive operation. Then multiply the result by the number of times the node is executed (from the monitoring of the program). Now by summing up the time costs of the individual primitive operations we can get an estimate of the time cost for representing this class of high-level data structure using this representation.

For each equivalence class of information structures we now have a table consisting of how much time and space would be required using each applicable representation. If one representation dominates another, that is, both its predicted space and time costs are less, then we drop the dominated representation from any further consideration. Note that this is a heuristic rather than an absolutely optimal thing to do. This is not necessarily optimal because of cross terms in the objective function (time spent in procedure for manipulation times space occupied by other structures) and is similar to the reason which prohibited us from selecting each information structure representation independently.

Now let us order the remaining representations by our objective function, in this case space-time product. The first of these representations (the one with the smallest space-time product(ignoring other data structures)) will be our initial guess as to the best representation for this class of information structures.

5.6 FINAL SELECTION

The total time-space product cannot be minimized by simply minimizing the time-space product of each data structure, because of the cross-terms of the form, time of operations on structure *A* multiplied by the space occupied by structure *B*. We therefore need a technique of minimization which suitably treats these problems.

The quantity we are attempting to minimize is the space-time integral. We will approximate this quantity by taking the sum of the terms of the form: average space in use during a procedure multiplied by the average time spent inside the procedure, the summation being performed over all procedures.

In this research we have restricted ourselves to non-recursive procedures. Thus we can construct a simple sequence of all the procedures of the user's program with the property that if procedure *A* calls procedure *B* then a node representing procedure *A* precedes the node representing procedure *B* in the sequence. Another way of stating this, is that without recursion there is a partial ordering of the procedures where the relation *calls or calls indirectly* is used to provide this partial ordering. We construct a sequence which realizes this partial ordering.

Once we have constructed such a sequence we now have the property that the time of execution for a procedure depends only on itself and the time of execution of procedures later on in the sequence. Thus, to estimate the space-time integral for a given representation set we do the following:

1. Processing the procedure list in reverse order, we estimate each procedure's execution time by using the already predicted average execution time of all procedures which it calls, the execution time required for non-leap constructs (provided by information gathering), and the estimates of the LEAP constructs of the procedure found by using the primitive operation time cost functions associated with the representation under consideration.

2. We then multiply the predicted time cost of a procedure by the storage costs of all the variables allocated within the procedure. Global constructs such as triples and the datums of items are counted as variables allocated in the outer block. The storage cost of a given construct is estimated by taking the average size of the construct and using that as a parameter to the storage cost function associated with the given representation.

Using the above algorithm we can obtain a prediction (admittedly crude) of the space-time integral using any given set of representations for the data-structures of the user's programs. We now shall state how we use these estimates to pick the final set of representations.

Our first guess of the set of representations consists of all those representations which minimize the local space-time product (that is only the average space used by a class multiplied by the execution time of the primitive operations on the class using a single representation). Using this set of representations we can form a preliminary guess as to the minimal achievable space-time integral.

We now iterate through all the classes of data structures. For each class and each possible representation we estimate the new space-time integral if that particular representation were chosen rather than the one currently chosen. If the new estimate of the space-time integral is better than the best seen so far, we will record the new *best* representation for the data structure class, and then continue by processing the next data structure class. We continue to iterate through all the data structure classes until we no longer get any improvement in the predicted space-time integral. We now have our final set of representations for the data structures of the user's program.

5.7 FINAL COMPILATION

The system has selected representations for all the LEAP data structures in the user's program, the system now compiles the SAIL program substituting calls on the appropriate primitive operations to handle the LEAP constructs. In our demonstration system, all the primitive operations were implemented as closed subroutines, but there is no inherent reason why the final Compiler could not generate in-line code for these primitive operations.

SECTION 6

RESULTS

We have implemented a demonstration system to test out many of these ideas. It works on a subset of SAIL which includes LISTS and SETS. It does not fully handle triples. All of the phases up to the user interrogation phase (monitoring, static analysis) process the operations on triples; only the interrogation and selection phase and final compiler would have to be modified. Also, of course, the library of primitive operations on associations (which has only been partially implemented) would have to be completed as well as analyzed.

The system consists of several programs, most of which are written using the SAIL language.

The first program is a trivial modification to the Stanford SAIL compiler and is written in assembly language. The only difference between this and the standard compiler involves the insertion of statement counters into the object code. We needed more precise knowledge of statement and expression counts and so we insert more counters into the object file than the standard compiler does. This first phase is used to obtain the statement frequency counts by compiling and executing the user's program using the user's data and our own default representations.

The next phase takes as input the user's source file and the statement counter file produced by the monitoring phase above. Its basic function is to parse the user's program into a flow graph and associate with each node in the flow graph the corresponding statement count. This program was formed by taking the parser and scanner from the standard SAIL compiler and substituting our own routines for the code generators. Thus part of it is written in assembly language and part in SAIL. The flow graph is in the form of LEAP triples. Other data that will be used in the successive phases is stored as the datums of items. The communication between phases takes the form of data files containing the items, datums of items, and associations between items. At the end of each phase such a file is written, and the next phase reads the file as input.

The static analysis, written entirely in SAIL, is the next phase. It performs the meta-evaluation of the program. It by far is the slowest of all the programs in the system. The next two phases (also written in SAIL) merely partition set and list variables and the associations into the appropriate equivalence classes.

The next phase does preliminary filtering. That is, it notes which representations will not be allowed to represent certain list or set classes because they do not provide essential primitive operations. It also interrogates the user as to the expected sizes of the data structures which are operands to the LEAP operators.

The penultimate phase uses the information gathered to select the representations of the sets and lists of the user's program. It may also be run in a mode in which the user can choose representations for some or all of the classes of sets and lists before the automatic selection.

The last phase is a compiler which uses the selections from the previous phase to decide which library entries to use to implement the primitive operations.

Before analyzing several test cases let us make some general observations about the speed of this demonstration system and some of its limitations. The major limitation is in the size of programs that it can handle. The standard SAIL implementation allows only 4000 items, which is only enough to represent the flow graphs and other data of programs approximately ten pages or so in length. The execution time for the various parts of the system (apart from the static analysis) total about 10 times the time required for merely compiling the program. This does not include the time for writing and reading the LEAP data base between phases. This input-output takes approximately 20 times the other execution time and tends to dominate the whole process. It could be reduced by aggregating several phases into single phases to reduce that cost, or by substantial reprogramming of the input and output primitives. Execution time for the static analysis pass varies dramatically depending on the depth of loop and procedure nesting but in typical programs takes as much execution time as the rest of the phases put together (not including input-output). In extreme cases it has been known to take two or three minutes of execution time to process a two page program that takes just a couple of seconds to compile. Clearly this is the phase which would have to be dramatically improved to make the whole system more cost effective.

Let us now analyze the results of using the system on several test programs. The program texts may be found in the appendices.

6.1 INSERTION SORT

The original insertion sort (Appendix D - INSRT2) was processed using manual selection to choose representations for the set variable *UNSORTED*, and the list variable *SORTED*.

Time to sort 300 integers (read from disk, originally in random order)

AVL TREE for *UNSORTED* , VARIABLE LENGTH ARRAY for *SORTED*
TIME = 6.7 (sec) SPACE = 10K

SORTED LINEAR LIST for *UNSORTED*, LINEAR LIST for *SORTED*
TIME = 18.5 (sec) SPACE = 8K

SORTED LINEAR LIST for *UNSORTED*, VARIABLE LENGTH ARRAY for *SORTED*
TIME = 4.5 (sec) SPACE = 8K

The program was then modified to form INSRT3 (Appendix E). The only difference being that the inner loop which iterated through the *SORTED* list was written as a *FOREACH* instead of an *WHILE* loop with list indexing. This, as expected, dramatically changed the time required for the implementation using a LINEAR LINKED LIST for the list *SORTED* (because with list selection we have to process the header of the list every time, thus the time for traversing the list is proportional to N^2 rather than N).

AVL TREE for *UNSORTED*, VARIABLE LENGTH ARRAY for *SORTED*
TIME = 6.1 (sec) SPACE = 10K

SORTED LINEAR LIST for *UNSORTED*, LINEAR LIST for *SORTED*
TIME = 7.5 (sec) SPACE = 8K

SORTED LINEAR LIST for *UNSORTED*, VARIABLE LENGTH ARRAY for *SORTED*
TIME = 6.0 (sec) SPACE = 8K

We note that the time required for the last representation increased. While it is not definite (because of the inaccuracies of the timing mechanism) that this is significant, it probably is. This would be caused by the fact that using FOREACH'S (in our implementation) always copies the set or list variable being iterated while the FOR with list indexed selection does not.

We also ran this program (INSRT3) with the standard SAIL system (which uses sorted linear linked lists to represent sets and linear linked lists to represent lists) and obtained a running time of 12.9 compared with 7.5 above. This time difference is probably caused by several factors. First, the list and set manipulation routines in our implementation (in particular the FOREACH interpreter) have been carefully optimized. Secondly, we have one less level of indirection in fetching the datums of items. Finally the linear lists used for sets are sorted in ascending order in SAIL, compared to descending order in our system. Since the *NEW* allocator in both systems allocates items in increasing order (in terms of the internal representation of items) a loop consisting of

put NEW(x) in SET

is likely to be much faster in our system since it will always insert the *NEW* at the head of the linked list, rather than having to traverse the entire list and then adding the new element at the end of the linked list. This type of knowledge about the *NEW* allocator seems very hard to include in an time-estimator function for the primitive operation. Perhaps the *NEW* allocation method should be chosen with knowledge of the representations of the data structures. In our system, though, we had fixed the allocation method in advance.

The automatic selection on the program INSRT3 used statistics gathered from executing the program on the same data set of 300 integers (a modified version of INSRT3 was used with explicit statistics gathering statements recording such things as average set size and so forth. These extra statements were inserted manually).

The automatic selection mechanism had to consider only two information structures, the original unsorted set, and the final sorted list. There were initially seven possible set representations to chose from. The applicability filter discarded the bit vector and combination of bit vector and linked list because these representations require knowledge of the maximum number of distinct elements which can be set members. The presence of *NEW*'s precluded the determination of this maximum size (Note in future systems, user assertions may provide information allowing such determination). The attribute representation was also discarded by the applicability filter because of the FOREACH iteration through the set. There is no implementation of iteration through a set represented by attribute bits in our library of implementations. The preliminary prediction phase now had only four of the original seven representations to consider. These were the sorted linked list, the height balanced binary tree, the

hash table, and the variable length array. The preliminary prediction phase discarded the variable length array representation because predictions indicated a small additional space requirement and a large additional time requirement compared with the sorted linked list representation. Similarly, the height balanced tree was discarded because of comparisons with the hash table representation. The final selection phase initially chose the linear linked list representation and did not alter its decision.

There were initially three list representations to choose from: a one way linked list, a two way linked list, and a variable length array. All are complete so the applicability filter did not eliminate any from further consideration. The preliminary prediction phase eliminated the doubly linked list because predictions indicated it would need both more time and space than the one-way linked list. The final selection phase initially chose the variable length array and did not alter its decision.

Thus, the automatic selection picked a hash table representation for the set *UNSORTED*, and a variable length array for the list *SORTED*. We then ran each representation pair 6 times to try and overcome the idiosyncrasies of our timer. The numbers below indicate the average time of the 6 attempts with the ranges of times in parentheses. The storage requirements are the same as before.

HASH TABLE for *UNSORTED*, VARIABLE LENGTH ARRAY for *SORTED*
(these are the representations automatically selected).
5.9 (5.5 , 6.1)

LINEAR LINKED LIST for both *UNSORTED*, and *SORTED*
(these are considered the default representations)
7.3 (6.4, 8.5)

LINEAR LIST for *UNSORTED*, and VARIABLE LENGTH ARRAY for *SORTED*
(the author's own choice)
5.9 (5.3, 6.1)

There is no significant difference between the first and third pairs above. We believe that the system chose the hashed set rather than the linked list because it overestimated the time required for set insertion using the linear linked list because it didn't consider the fact that items are allocated in increasing numerical (internal representation) order. To include this type of knowledge in the automatic selection seems relatively hard.

We then ran the same program over a data set containing 1000 elements with 2 trials per representation. We did not rerun the automatic selector, but just used its choices from the 300 item sample.

HASH TABLE for *UNSORTED*, VARIABLE LENGTH ARRAY for *SORTED*
(these are the representations automatically selected from before).
47.9 (45.0 ,50.1)

LINEAR LINKED LIST for both *UNSORTED*, and *SORTED*
(these are considered the default representations)
1:09.4 (1:08.0, 1:10.9)

Before we get too ecstatic about the improvement (approximately 25%) let us note that another program sorting the same 1000 elements ran in slightly over 6 seconds. This program, however admittedly used a different algorithm (it inserted every integer read in into an AVL tree and then traversed the tree in preorder). Of course the selection of the appropriate sorting algorithm is a separate issue and has been considered elsewhere ([Knuth73]).

6.2 MERGE SORT

We took a merge sorting program (Appendix F, MERGE) and processed it on a sample data set of 300 elements. The automatic selection phase considered three equivalence classes: one containing only the set *UNSORTED*; the second containing the two lists *OLDLISTS*, and *NEWLISTS*; and the third containing the lists *SORTED*, *MERGER*, and all the list datums. In processing the seven possible set representations for *UNSORTED*, the applicability filter eliminated the bit vector and combination (bit vector and linked list) because of the presence of *NEW*s, which make maximum size of the sets indeterminate at compile time. The applicability filter also eliminated the attribute bit representation because a FOREACH search was performed on *SORTED* and our implementation does not provide the primitives for foreach searches on sets represented by attribute bits. Thus, applicability filtering eliminated three of the seven possible set representations. The four remaining were a sorted linked list, a height balanced binary tree, a hash table, and a variable length array. The first prediction phase eliminated the variable length array because predictions indicated both more execution time and more space needed than a linked list. The height balanced binary tree was also eliminated because predictions indicated larger execution time and space requirements using it than using the hash table. Thus, after the preliminary processing stage, we had only two remaining candidates from our original seven: a sorted linked list and a hash table. The predicted time for set insertion dominated the final selection and the hash table was picked.

There were three candidates for representing the lists, *OLDLISTS*, and *NEWLISTS*: a one way linked list, a doubly linked list, and a variable length array. The applicability filter did not discard any of these representations because they are all complete. The preliminary predictor discarded the variable length array because predictions indicated it would take more time and space. The final selection phase initially chose the one way linked list representation and then changed its decision to use the doubly linked list representation. This occurred because the extra space needed was very small (only 300 words) but the predicted time required was about half. The cross-terms of the form time using this representation times the space of other data structures dominated.

There were similarly three candidates for the last equivalence class. Applicability and preliminary prediction did not eliminate any representations. The final selection initially picked a one way linked list and did not alter its decision.

Thus the selection picked linked list representations for the lists (linear one-way for one class and doubly linked for the other), and a hash table for the original unsorted set. This agrees somewhat with my own choice except that I would have chosen a linear linked-list for the set for the same reason we gave before (the *NEW* generator returns items in increasing internal order). I also would not have chosen the doubly linked list since it takes up twice as much space. It seems that it was selected because of some list indexing operations would be expected to take

approximately half the time using a doubly linked list compared with singly linked because we can count either forward from the head of the list of backward from the tail. However, in this particular program the indices used were the constant 1, therefore there was no advantage to using the doubly linked structure.

We ran the resulting program and it took approximately 4.2 seconds and the core size grew to approximately 12K. To demonstrate that this was a good selection we then ran the program using variable length-arrays for the lists and got an average time of about 5 seconds and a similar core size.

We then ran the same programs with a sample data set of 1000 elements The linked list and hash table version took approximately 17.5 seconds and the array version took about 30 seconds. The core sizes were 21K and 37K respectively. Therefore we see again that the automatic selection procedure again made a reasonably good choice. (The best choice turned out to be to use linear linked lists for all of them 12.5 sec, 19K core.)

One interesting observation is that the core size used by the variable length array implementation was 37K as opposed to 19K for the linked list representation. According to our model of storage costs it shouldn't be more than about 2K larger yet it was 18K larger. This is caused by the problem of storage fragmentation or checkerboarding. In the merge sort we are continually allocating larger and larger blocks of storage while at the same time deleting twice as many blocks of half the size. The storage allocation routine we use just forms a free list of the deleted blocks (merging blocks when adjacent blocks become free). Therefore, when we wish to allocate a block of $2\ N$ words it may be true that there are $2\ N$ free cells, but no single contiguous block of $2\ N$ free cells. The storage allocation routine is thus forced to increase the core size even though our model of storage would indicate that this is unnecessary. It is unclear how to include the fragmentation costs in a model of storage in a simple way.

6.3 TRANSITIVE CLOSURE

As our final example we look at the transitive closure procedure we have seen so often before. Here we represented the single binary relation $REL \otimes \cdot A \equiv B$ as $B \in \mathrm{datum}(A)$. This is similar to the field selector (record offset) implementation of the ternary relation. We expressed this explicitly since our demonstration system does not handle triples in its final selection phases.

The relation we created was equivalent to the son relation in a binary tree. That is, every node other than *leaf* nodes had two other nodes related to it. The size of the tree was 1000 nodes. And we asked the procedure to find the descendants of a node in the third row of the tree. The time to perform the transitive closure itself was so small that we called the procedure 40 times to get a meaningful number.

There were two set equivalence classes: the first containing all the set variables of the procedure and the second containing the set datums we used to represent the binary relation. The applicability filter threw out the bit vector, bit vector and linked list, and the attribute bit representations because of the presence of NEW's and iteration as in the other two examples. The preliminary predictor chose the linear linked list representation because predictions indicated it would take both less time and space than any of the other available representations.

Thus, the system selected the default representation (linked list for the sets). Execution time was about (as the average of 5 trials) 28 seconds per program execution with a core size of 17K.

To see how this compares with what we believed to be the next best representation, we then ran the program with a variable length array representation and the program took an average of about 31 seconds with a core size of 17K. Thus, the selection process chose a representation about 10% more efficient than the next most likely representation.

With most of the test cases we attempted there was a marked improvement in the execution time of the program (space did not vary as much). In cases where the system selected a suboptimal representation, specific defects were pinpointed (such as failure to notice constant list indices, storage checkerboarding etc) which may be remedied in future selection systems. All in all, we feel that the ability to automatically choose from various representations for information structures has been shown to be feasible and obviously desirable.

SECTION 7

CONCLUSIONS AND DIRECTIONS FOR FUTURE RESEARCH

We feel that the system has performed quite well. In general it has chosen appropriate data structures for the programs considered. Where it has failed to choose optimum structures, it has led us to specific defects in our models of storage and execution time. Some of these, like the use of constant list indices can be easily handled by simple modifications. Others such as storage fragmentation and using internal properties of other structures (such as the fact that the *NEW* allocator usually returns items in increasing order) are not so easily handled.

The system we have described is far from complete. It works only on a subset of a usable programming language. It is very slow and cannot process large user programs. However it is, we believe, a concrete demonstration of the validity of our original hypothesis. Namely, that it is possible to use high-level data structures such as sets and relations with their conceptual clarity, and have an automatic representation selector select efficient implementations of these abstract structures. Future systems of this type should indeed be usable in a standard production environment.

The concepts we have mentioned here, partitioning of data structures into classes, flow analysis, analysis of algorithms, execution time monitoring, etc., are not new. However, to our knowledge, they have never before been combined to form a coherent system capable of automatically selecting representations of data.

We would be the last to claim that this system solves all the problems of automatic selection of representations. We have barely scratched the surface. Below, in our suggestions for future research we will list some of the hard problems that have occurred to us during our research in this area. Other research is needed in analysis of algorithms, and classical optimization techniques. There are potentially great payoffs in other automatic coding techniques, such as Earley's iterator inversion.

7.1 TOPICS FOR FUTURE RESEARCH

ADDITIONAL OPTIMIZATIONS

1. Computation avoidance - takes advantage of properties of the abstract data structure.

   For example, the boolean expression:

   $$X \in (SET1 \cup SET2)$$

   is logically equivalent to

   $$(X \in SET1) \vee (X \in SET2)$$

   it will in general be faster and take less storage space to evaluate the second expression rather than the first since the union does not have to be computed. See appendix G of transformations.

2. Change of variables

Often we have expressions like *SET1 - SET2*. Sometimes it is beneficial to keep this as an explicit set rather than recompute the expression every time it is used. In this case it would mean that every time we insert an element into *SET1* we would insert it in the difference set if it were not a member of *SET2*. Every time we inserted an element into *SET2* we would remove that element (if present) from the difference set and so forth. This is closely related with the concept of *iterator inversion* of Earley.

3. Copy optimizations - mainly based on dead variable analysis and read-only access to data structures being iterated through.

We often can suppress unnecessary copying operations if we use dead variable analysis. For example:

```
SET1 ← SET2;

SET2 ← phi;

put B in SET1;
```

We would normally compile this as: make a copy of *SET2*, release *SET1*, place descriptor to copy into *SET1*, Release the space used by *SET2*, insert *B* into *SET1*. However, when we realize that *SET2* is dead after the assignment to *SET1* we can eliminate the copy *SET2*, and release *SET2* operations.

We can also use dead variable analysis to determine when we are through with a variable. Thus, we can release the storage it required, much earlier than its explicit release in the program. This technique will not decrease the running time of the program significantly but may decrease the maximum size of the core image as we are able to reuse space sooner.

Similar analysis can tell us when it is necessary to copy a data structure which is being processed by an iterator.

IDENTITY vs. ATOMIC OBJECTS

One question arises when we talk of sets, sequences or relations in a programming language. What are the elements of these data structures? Are they values or variables? The system we have described acts only on variables.

Earley has named these concepts of value and variable by the corresponding terms of *atomic* objects and *identity* objects. An atomic object is essentially a simple value, which can not be altered. That is, it is readonly. To alter a set containing the values 1,2 and 3 to contain the values 1, 2 and 4 we would probably remove the value 3 and then add the value 4 to the set. With sets of atomic objects we often do not have to explicitly construct data structures but can utilize functions and generators to represent them. For example, assume we had knowledge that a given set was simply the set of integers from 1 to 100. To iterate through the elements of that set we

need not explicitly construct a data structure which contains 100 data objects, but could merely use an standard ALGOL FOR statement. Similarly we could replace the set membership test with the simple range check. That is, the integer x is in the set if and only if the value x is greater than or equal to 1 and less than or equal to 100.

An identity object is essentially a variable name. It has its own identity and its value (datum) can be modified at will. Thus if we had a set containing identity objects whose values were 1, 2 and 3 and we wished to modify it to contain the identity objects whose datums were 1, 2 and 4, we could either change the datum of the identity object which was currently 3 or we could remove that object and replace it with another identity object whose datum was 4. If we wished to have the set of integers from 1 to 100, we would be constrained to actually having 100 different objects each of whose datum was some integer in that range.

Both identity objects and atomic objects are valuable concepts for very high-level data structures. We can easily implement either one using the other but when we do so we are playing the same game as the fellow who wrote all his set operations in terms of sequence operations. We have expressed data in terms of an implementation rather than in terms of its high level properties.

In our system we concerned ourselves only with the representation of identity objects. They are more easily handled than atomic objects because it is always clear when a new one is being created, which is not the case with atomic objects. Possible ways in which atomic objects could be handled in future systems include the obvious encoding in terms of read-only identity objects (via such "symbol-table" mechanisms as hash searches etc.). The most interesting problems involve the use of functions and generators to take the place of explicit data structures.

REDUNDANT AND ALTERNATE REPRESENTATIONS

Alternate representations for different phases of program.

Often we can partition a program into several logical phases (e. g. input, processing, output). A representation optimal for one such phase may be suboptimal for another (depending on which access operations are dominant). The problem is to recognize the phases and then decide if it is worth the translation time to get from one storage structure to another. For example we may have a file of employment records read in off of tape in random alphabetical order (INPUT). The program then will update the file according to some other criteria, e.g, employee number (PROCESSING). Finally reports to be generated wanted in alphabetical order so desire sorting on employee name (OUTPUT). We may find it optimal to have different representations for the file in each different phase of the program.

Simultaneous use of multiple representations of an information structure

Often two accessing operations may be performed on the data structure. If no single representation is optimal for both operations it may be advisable to store the data redundantly by using two storage structures each containing the data structure organized in a manner optimal for one of the access operations. Recall that one of our set representations was a bit vector with a

redundant linked list. For example, we might have a set of possibly 72 different elements, with two operations performed on the set: existence test (is x in set), iteration (Foreach x ϵ set). The existence test is done best when representation is a bit string so only indexing is needed. However, iteration is done best when representation is a linked list of the elements.

Note that the merger of two storage structures does not necessarily have the sum of the updating costs of the individual structures. Normally with a list of the set elements we would require that the list be sorted, However now this is not necessary. The operation of insertion can be done by first seeing it the element is already in the set by using the bit string and then adding it to the head of the list if is not. Similarly deletion can be avoided if the existence test fails.

One way of approaching multiple representations is to consider this merger as a separate representation with its own attributes. One problem is that this can lead to a squaring of the size of the representation library if we do this with all pairs of representations.

Data structures as unions (disjoint or not) of storage structures.

We have mentioned this in the context of associations. This is applicable even with sets. Consider a program which is going to test membership in a very large set. For example a spelling checker might check to see if every word in a piece of text were in its dictionary.

The dictionary might be very large and thus would have to be stored in secondary storage. We desire a way to minimize the number of accesses to this storage. One technique might be to keep a large number of words (say the last 1000 encountered) in core. Thús, our search algorithm would first search core for the desired word and only if it did not find it, make the appropriate search in secondary memory. Thus, the set of allowable words is stored as a two data structures (in core, and on disk) with a partial redundancy. Other applications might require no redundancy.

RELAXATION OF CRITERIA FOR THE EQUIVALENCE CLASSES

To reduce the combinatorics of representation selection we insisted that arguments to a single operator be in the same representation. The alternatives are to have either a translation procedure which takes as argument a set in one representation and converts it into the other representation, to write code sequences for each operator which are representation independent, or to represent one or both sets redundantly.

We could thus write a *representation free* union code sequence as follows, where sets A and B are the inputs.

```
set procedure UNION(set A,B);
begin "UNION"
      itemvar LOCAL;
      set RESULT;
      RESULT ← phi;
      foreach LOCAL | LOCAL ϵ A do
            put LOCAL in RESULT;
      foreach LOCAL | LOCAL ϵ B do
            put LOCAL in RESULT;
      return(RESULT);
end "UNION";
```

The arguments A,B and the result may be in entirely different representations. The two *foreach's* and *puts* could check which representation is in use for the construct and use the appropriate routine for that representation.

It is quite easy to write similar *representation free* codes for the other basic operators. However we notice that we lose any efficiency based on representation. For example when compared to the representation dependent model with sorted linear linked lists we find that the representation free routine will take time proportionate to n^2 compared to time proportional to n for the representation dependent union routine.

A basic problem for future research is the evaluation of the tradeoffs between using less efficient representation-free routines, using translations to a common representation, and insisting on common representations.

USE OF PACKING

String representations have traditionally packed more than one character per computer word. Clearly the record type structures which a system like ours generates for items could benefit from the same type of packing. Here we need to evaluate the tradeoffs between sometimes slower access to elements (because of unpacking and packing) and the storage savings.

EXTENSIBLE LANGUAGES.

A user should be able to define his own abstract data structures and supply a library of primitive operations using various representations and still have the system do automatic selection of representation. This is closely related to the work of Earley and the ECL group at Harvard. However, they are not currently working on this particular problem.

7.2 FINAL CONCLUSION

This research has demonstrated the feasibility of automating a significant part of the programming problem: the selection of low level representations for high level information structures. Future work along these lines is likely to allow the techniques to be applied as a matter of course in an optimizing compiler. We have demonstated the desirability of such work.

SECTION 8

APPENDICES

8.1 APPENDIX A - SET PRIMITIVES

In this implementation a set is always be represented by a one-word descriptor. This descriptor usually contains a pointer to some other storage and perhaps additional information. In the following routine descriptions, a value set argument is represented by the one-word descriptor. Similarly, all set-valued primitives return a one-word descriptor

1. **PUT_SET** (itemarg, setarg) - inserts the item argument into the set represented by the set descriptor argument. Returns a descriptor to the resultant set. This routine has the effect of altering its original set argument.

   { item1, item2, item3 } would thus be compiled as:
   ```
   TEMP ← PUT_SET(item1,PHI);
   TEMP ← PUT_SET(item2,TEMP);
   TEMP ← PUT_SET(item3,TEMP);
   ```

   the result would then be in *TEMP*.

   put ITEMARG **in** SETA would be compiled as:
   ```
   SETA ← PUT_SET(ITEMARG, SETA);
   ```

2. **REMOVE_SET** (itemarg, setarg) - removes the item argument from the set represented by the set descriptor argument, RETURNING A DESCRIPTOR TO THE RESULTANT SET. The original set is altered.

   remove *ITEMARG* **from** SETA; would be compiled as:
   ```
   SETA ← REMOVE_SET(ITEMARG, SETA);
   ```

3. **LENGTH_SET** (setarg) - returns the number of elements in the set. It does not alter its argument.

4. **IN_SET** (itemarg,setarg) - boolean returns **TRUE** if the *itemarg* is an element of the set, **FALSE** otherwise. Does not change the set argument.

5. **COPY_SET** (setarg) - returns a copy of its argument. Does not change the argument. With a representation which tried to share storage this routine might just copy the descriptor or increment a reference count [Schwartz74a].

6. RELEASE_SET (setarg) - releases the storage (if any) used by the set argument. Thus, it destroys its argument. With a representation which used garbage collection this routine would probably do nothing at all and all storage reclamation would be done by calls to a garbage collector located elsewhere.

   FOO ← *BAZ* would be compiled as:

```
TEMP ← COPY_SET(BAZ);
RELEASE_SET(FOO);
FOO ← TEMP;
```

7. SET_UNION (seta,setb) - forms the union of its two arguments. It has the side effect of destroying its first argument, but leaves the second unchanged (unless the second argument happens to be the same as the first).

   Thus, *FOO* ← *FOO* ∪ *BAZ* would be compiled as:

```
FOO ← SET_UNION(FOO, BAZ);
```

   FOO ← *BAZ* ∪ *GARP* would be compiled as:

```
TEMP ← COPY_SET(BAZ);
TEMP ← SET_UNION(TEMP, GARP);
RELEASE_SET(FOO);
FOO ← TEMP;
```

8. SET_INTERSECTION (seta, setb) - forms the intersection of its two arguments. destroying its first argument as a side effect. It leaves its second argument unchanged.

   Thus, *FOO* ← *BAZ* ∩ *FOO* would be compiled (using the commutativity of set intersection) as:

```
FOO ← SET_INTERSECTION(FOO,BAZ);
```

9. SET_SUBTRACTION (seta, setb) - does the set subtraction, second argument subtracted from the first, destroying the original first argument as a side effect. Leaves second argument unchanged.

   FOO ← *FOO* - *BAZ* would be compiled as:

```
FOO ← SET_SUBTRACTION(FOO,BAZ);
```

10. SET_EQUALITY (seta, setb) - does the boolean comparison between its two arguments. The two arguments are left unchanged.

    FOO = {*itma*} would be compiled as,

```
TEMP ← PUT_SET(itma, PHI);
TBOOL ← SET_EQUALITY(FOO, TEMP);
RELEASE_SET(TEMP);
```

    the result of the comparison is contained in *TBOOL*.

11. SET_INCLUSION (seta, setb) - does the boolean comparison and returns **FALSE** if there is an item in *seta* which is not in *setb*. Does not change either of its arguments.

12. SET PROPER_INCLUSION (seta,setb)- same as **SET_INCLUSION** except also returns **FALSE** if two set arguments were equal.

13. COP_SET (seta) - returns an *arbitrary* element of the set argument. It does not alter its argument.

14. LOP_SET (reference seta) - takes as argument the address of the set variable (not just the descriptor). Removes a single element from that set which it returns as its value. It alters the set argument.

15. INIT_SET_FOREACH (reference scb; reference locality; seta) - This is called when the foreach is entered. The *scb* is a variable which will contain status information for the *iterator* such as where we are in the set and so forth. The *locality* is the itemvar which is receiving values from the *FOREACH* search. *seta* is a destroyable copy of the set through which we wish to iterate.

16. ITERATE_SET_FOREACH (reference scb) - this is a boolean procedure which takes the scb variable as its parameter. It places the *next* element in the set (if any) into the *locality* which was mentioned in the initialization routine above. If the set has been exhausted it returns the boolean value **FALSE**, otherwise the value **TRUE**. On exhaustion it has the side effect of cleaning up everything, reclaiming space and zeroing out the *scb*. (search control block).

17. END_SET_FOREACH (reference scb) - this procedure forces termination of a *foreach*. It is used to clean up scbs before transfer of control outside a foreach statement such as a done (loop exit) statement or return (procedure exit).

```
foreach X | X ∈ SET1 do
    if X = FOO then remove X from SET1 else done;
```

This is compiled as:

```
INIT_SET_FOREACH(SCB, X, COPY_SET(SET1));
while (ITERATE_SET_FOREACH(scb)) do
    if X = FOO then
        SET1 ← REMOVE_SET(X, SET1)
    else  begin END_SET_FOREACH(SCB);
            done;
          end;
```

We should note that there are other possible ways of choosing the set of primitive operations. For example we can conceive of an assignment primitive, or a primitive for constructing explicit sets. Copying and releasing sets is sometimes very expensive. Therefore we might have many entry points (as many as 4 for binary operations) depending on whether the arguments are dead, and so their storage might be reused immediately. In this system, each routine has but a single entry point.

8.2 APPENDIX B - LIST PRIMITIVES

1. PUT_INDEXED (itemarg, reference list, index) - inserts the item argument into the list specified by the list parameter after the specified index. It has effect of altering the list argument.

   put X in LISTA after 10;

   is compiled into

   PUT_INDEXED(X, LISTA, 10);

   and

   put X in LISTB before 1;

   is compiled into

   PUT_INDEXED(X, LISTB, 0);

2. PUT_BEFORE_ITEM (item1, reference lista, item2) - inserts *item1* into the *list* immediately before first occurence of *item2*. The list argument is altered.
   put X in LISTB before Y;
   is compiled into
   PUT_BEFORE_ITEM(X, LIST, Y);

3. PUT AFTER_ITEM (item1, reference lista, item2) - inserts *item1* into *lista* immediately after first occurrence of *item2*. The list argument is altered.
   put X in LISTA after Y;
   is compiled into:
   PUT AFTER_ITEM(X, LISTA, Y);

4. REMOVE_ITEM (itemarg, reference listarg) - remove the first occurrence of *itemarg* from *list*. The list argument is altered.

5. REMOVE_INDEXED (index, reference listi) - remove the *index* th element of *listi*. The list argument is altered.

6. REMOVE ALL (itemarg, reference lista) - remove all occurences of *itemarg* from *lista*. The list argument is altered.

7. FETCH INDEXED (listexpr, index) - returns the *index* th element of the *listexpr*. The list argument is unchanged.

8. REPLACE_INDEXED (reference listi, index, itemarg) - replaces the *index* th element of *listi* with the *itemarg*. The list argument is altered.

9. LIST_MEMBERSHIP (itema, listb) - boolean TRUE if *itema* an element of *listb*. The list argument is not altered.

10. **LIST_EQUALITY** (list1, list2) - boolean, tests if two lists are equal. Neither argument is altered.

11. **LENGTH_LIST** (lista) - returns the length of *lista*. The list is unaltered.

12. **COPY_LIST** (listo) - given a list descriptor, returns a list descriptor pointing to a copy of the original list. Does not alter its argument.

13. **RELEASE_LIST**(listr) - release the space occupied by a list expression back to free storage. The argument is thus destroyed.

    FOOLIST ← BAZLIST

    is compiled into:

    ```
    temp ← COPY_LIST(BAZLIST);
    RELEASE_LIST(FOOLIST);
    FOOLIST ← temp;
    ```

14. **COP_LIST** (lista) - COP of list (archaic equivalent to list[1]). The list argument is unchanged.

15. **LOP_LIST** (reference lista) - Remove and return first element from list. The list argument is altered.

16. **CONCATENATION** (list1, list2) - form a new list by concatenating two lists together. Both arguments are destroyed.

    FOOLST ← BAZ & FOOLST;

    is compiled into:

    ```
    temp ← COPY_LIST(BAZ);
    FOOLST ← CONCATENATION(temp,FOOLST);
    ```

17. **INIT_LIST ITERATOR**(reference scb, reference localitv, list) - initialize the foreach list element iterator. List argument eventually destroyed. (See set foreach iterators Appendix A).

18. **ITERATE_LIST** (reference SCB) - iterate through a list. Returns **TRUE** if it finds another element in list. **FALSE** otherwise. Side effect of storing item found in the *localitv* mentioned in **INIT_LIST_ITERATOR**.

19. **END_LIST** (reference SCB) - forced termination of a **FOREACH** element iterating through a list.

20. **EXPLICIT LIST** (item1, item2,... itemN, N) - constructs the descriptor for an explicit list. Takes a variable number of parameters.

8.3 APPENDIX C - META EVALUATIONS

Here are some examples of the meta evaluations we use during our static analysis phase.

A. Set operations

1. VALUESET(S1 ∪ S2) = VALUESET(S1) ∪ VALUESET(S2)

2. VALUESET(S1 ∩ S2) = VALUESET(S1) ∩ VALUESET(S2)

3. VALUESET(S1 - S2) = VALUESET(S1)

4. after S1 ← SETEXPR

   a. If S1 is a simple variable (not datum, array element, procedure parameter) new VALUESET(S1) = VALUESET(SETEXPR)
   b. If S1 is not a simple variable, then the

   new VALUESET(S1) = old VALUESET(S1) ∪ VALUESET(SETEXPR)

5. put ITEMEXPR in S1, acts the same as S1 ← S1 ∪ {ITEMEXPR };

6. remove ITEMEXPR from S1, acts like S1 ← S1 - {ITEMEXPR} i. e. no action.

B. Associative operations

1. MAKE iexpr ⊙ iexpr2 ≡ iexpr3.

   Insert every instance of x ⊙ y ≡ z (x ∈ valueset(iexpr1), y ∈ valueset(iexpr2), z ∈ valueset(iexpr3)) into model of the associative store.

2. ERASE iexpr ⊙ iexpr2 ≡ iexpr3.

   No action.

3. SEARCH iexpr1 ⊙ iexpr2 ≡ iexpr3.

   No change to the model of associative store. If this is a foreach element binding some local, do an assign to that local consisting of the corresponding elements from model of associative store.

8.4 APPENDIX D - INSRT2

```
begin "INSRT2"

set UNSORTED; list SORTED;
integer itemvar OBJ1, OBJ2;
integer COUNT, I; string TEMP;

comment FIRST CONSTRUCT AN "UNSORTED SET";

UNSORTED ← phi;
COUNT ← READ INTEGER;
for I ← 1 step 1 until COUNT do
    put new(READ INTEGER) in UNSORTED;

SORTED ← nil;

foreach OBJ1 | OBJ1 ∈ UNSORTED do
    begin "foreach OBJ1"
        COUNT ← 1;
        while COUNT ≤ length(SORTED) do
            begin "INNER"
                OBJ2 ← SORTED[COUNT];
                if datum(OBJ2) ≥ datum(OBJ1) then done "INNER"
                    else COUNT ← COUNT + 1;
            end "INNER";
        put OBJ1 in SORTED before COUNT;
    end "foreach OBJ1";

foreach OBJ2 | OBJ2 ∈ SORTED do
    WRITE_INTEGER(datum(OBJ2));

end "INSRT2"
```

8.5 APPENDIX E - INSRT3

```
begin "INSRT3"

set UNSORTED; list SORTED;
integer itemvar OBJ1,OBJ2;
integer COUNT, I; string TEMP;

comment CONSTRUCT AN "UNSORTED SET";

UNSORTED ← phi;
COUNT ← READ INTEGER;
for I ← 1 step 1 until COUNT do
    put new(READ_INTEGER) in UNSORTED;

SORTED ← nil;

foreach OBJ1 | OBJ1 ∈ UNSORTED do
    begin "foreach OBJ1"
        COUNT ← 1;
        foreach OBJ2 | OBJ2 ∈ SORTED do
            begin "INNER"
                if datum(OBJ2) ≥ datum(OBJ1) then done "INNER"
                    else COUNT ← COUNT + 1;
            end "INNER";
        put OBJ1 in SORTED before COUNT;
    end "foreach OBJ1";

comment PRINT SORTED LIST;

foreach OBJ2 | OBJ2 ∈ SORTED do
    WRITE_INTEGER(datum(OBJ2));

end "INSRT3"
```

8.6 APPENDIX F - MERGE

```
begin "MERGE"

list OLDLISTS, NEWLISTS, SORTED, MERGER;
set UNSORTED;
integer itemvar OBJ1, OBJ2, INFINITY;
integer COUNT, I;
list itemvar LITM1, LITM2;

comment CONSTRUCT AN "UNSORTED SET";

UNSORTED ← phi;
COUNT ← READ INTEGER;
for I ← 1 step 1 until COUNT do
    put new(READ_INTEGER) in UNSORTED;

comment CREATE LIST OF LISTS TO BE MERGED;

OLDLISTS ← nil;

foreach OBJ1 | OBJ1 ∈ UNSORTED do
    put new({{ OBJ1 }}) in OLDLISTS after 0;

NEWLISTS ← nil;
INFINITY ← new(2130);
```

```
while length(OLDLISTS) > 1 do
   begin "OUTER"
        while length(OLDLISTS) > 1 do
            begin "INNER"
                LITM1 ← lop(OLDLISTS);
                LITM2 ← lop(OLDLISTS);
                MERGER ← nil;
                while (datum(LITM1) ≠ nil ∨ datum(LITM2) ≠ nil) do
                    begin "INNERMOST"
                        if datum(LITM1) ≠ nil then
                            OBJ1 ← cop(datum(LITM1))
                          else OBJ1 ← INFINITY;
                        if datum(LITM2) ≠ nil then
                            OBJ2 ← cop(datum(LITM2))
                          else OBJ2 ← INFINITY;
                        if datum(OBJ1) < datum(OBJ2) then
                            begin
                                put OBJ1 in MERGER after length(MERGER);
                                remove 1 from datum(LITM1);
                            end
                          else
                            begin
                                put OBJ2 in MERGER after length(MERGER);
                                remove 1 from datum(LITM2);
                            end;
                    end "INNERMOST";
                put new(MERGER) in NEWLISTS after 0;
                delete(LITM1);
                delete(LITM2);
            end "INNER";
        if OLDLISTS ≠ nil then
            put lop(OLDLISTS) in NEWLISTS after 0;
        OLDLISTS ← NEWLISTS;
        NEWLISTS ← nil;
   end "OUTER";

LITM1 ← lop(OLDLISTS);
SORTED ← datum(LITM1);
delete(LITM1);
delete(INFINITY);

foreach OBJ | OBJ1 ∈ SORTED do
   WRITE_INTEGER(datum(OBJ1));

end "MERGE SORT"
```

8.7 APPENDIX G - TRANSFORMATIONS

The following are examples of transformations may be made to avoid certain computations. Caution must be taken with "side effects". None of these transformations were used in our demonstration system.

1. $x \in (set1 \cup set2) \equiv (x \in set1) \vee (x \in set2)$

2. $x \in (set1 \cap set2) \equiv (x \in set1) \wedge (x \in set2)$

3. $x \in (set1 - set2) \equiv (x \in set1) \wedge \neg(x \in set2)$

4. LENGTH(list1 & list2) ≡ LENGTH(list1) + LENGTH(list2)

5. $x \leftarrow x \cup \{item1, item2\}$ ≡
   put item1 **in** x;
   put item2 **in** x;

   (no need to explicitly create {item1, item2}.

6. $x \leftarrow x - \{item1, item2\}$ ≡
   remove item1 **from** x;
   remove item2 **from** x;

7. $(set1 \cup set2) = phi \equiv (set1 = phi) \wedge (set2 = phi)$

8.8 APPENDIX H - INSTRUCTION WEIGHTS

The table below contains our weightings of the individual machine instructions based on a time unit of approximately .7 microseconds. Data from PDP-10 SYSTEM REFERENCE MANUAL 1969. Digital Equipment Corporation. Note: no difference in execution time is noted if the source or destination of a memory reference is an accumulator. Thus, in our model. loading an accumulator from an accumulator will take as much time as loading an accumulator from the slower memory.

MOVES (MOVE, HRR, HRL, MOVS , HLL , HLR etc.)	
memory to accumulator	3 units
immediate to accumulator	2 units
accumulator to memory	4 units
EXCH	4 units
BLT	
PUSH, POP	5 units
LDB, ILDB	9 units (middle byte(5))
DPB, ILDB	10 units "
IBP	4 units
LSH	6 units
LSHC	7 units
CAI	2 units
CAM	
LOGICALS (OR, XOR, ANDCM, ANDCA, AND etc) (approximate)	
memory with accumulator	3 units
immediate with accumulator	2 units
accumulator with memory	4 units
ADD,SUB	
memory with accumulator	3 units
immediate with accumulator	2 units
accumulator with memory	4 units
AOBJN	2 units
JUMP	2 units
SKIP	3 units
AOJ,SOJ	2 units
AOS,SOS	
TL, TR	3 units
TD	4 units
XCT	
JFFO	5 units
JSP	2 units
JRST	2 units
PUSHJ, POPJ	4 units

8.9 APPENDIX I - EXECUTION TIME COST FUNCTIONS

PUT_SET - insert item in set
π = proportion of time item already in the set
λ = average size of set
M = maximum size of set

REPRESENTATION	set empty	set non-empty
Linked list	100	64 - 42*π + 6*λ
AVL tree	56	180-166π+16.8*LOG2(λ)
Bit - Array	146 + 3*⌈M/32⌉	48
Hash table	521	82-40π + 3λ/16
Bit-string with unsorted linked list	265 + 3*⌈M/32⌉	104-53π
Attribute bit	27	
Sorted variable length array	140	96-80π+5.05*λ + 20.5*LOG2(λ) -.3π*λ

REMOVE SET - remove item from set
π = proportion of time item in the set
λ = size of set

REPRESENTATION	Set empty	Set non-empty	Removal of last
Linked list	14	$23 + 13\lambda/2 + 27\pi$	82
AVL tree	11	$32 + 88\pi + 20*LOG2(\lambda)$	140
Bit - Array	48		
Hash table	11	$42 + 3\lambda/8 + 25\pi$	294
Bit-string with unsorted linked list	15	$51+6\pi\lambda + 50\pi$	149
Attribute bit	27		
Sorted variable length array	11	$17 + 21.5*LOG2(\lambda)+ 3\pi + 3\pi\lambda$	240

IN SET - test if item in the set
π = proportion of time item in the set
λ = size of set

REPRESENTATION	set empty	set non-empty
Linked list (sorted)	14	21 + 5λ
AVL tree	14	19 + 12*LOG2(λ) -5π
Bit - Array	48	
Hash table (32 slots)	13	40 + 5λ/32
Bit-string with unsorted linked list	18	51
Attribute bit	26	
Sorted variable length array	14	20 + 43*LOG2(λ)/2 -5π/2

COP_SET - Choose element of set
λ = size of set
M = maximum size of set

REPRESENTATION	TIME
Linked list (sorted)	15
AVL tree	12 + 12.4*LOG2(λ)
Bit - Array	21 + 12*⌈-(⌈M/32⌉/λ)*LN(1-λ/(⌈M/32⌉*32)⌉
Hash table(32 slots)	27
Bit-string with unsorted linked list	22
Attribute bit	∞
Sorted variable length array	17

LOP. SET - pick item and remove from the set
λ = size of set
M = maximum size of set

	new set empty	new set non-empty
Linked list (sorted)	65	48
AVL tree	42	64 + 18.6*LOG2(λ)
Bit - Array	37 + 12*⌈-(⌈M/32⌉/λ)*LN(1-λ/(⌈M/32⌉*32)⌉	
Hash table (32 slots)	265	52 + 16*(31/32)↑(λ-1)
Bit-string with unsorted linked list	116	89
Attribute bit	∞	
Sorted variable length array	225	21

LENGTH SET - count number of items in set
λ = size of set
M = maximum size of set

	set empty	set non-empty
Linked list (sorted)	9	
AVL tree	9	
Bit - Array	$20 + 12*\lceil M/32 \rceil + 9*\lambda$	
Hash table (32 slots)	9	
Bit-string with unsorted linked list	15	18
Attribute bit	∞	
Sorted variable length array	9	12

SET_UNION - union of two sets
π = proportion of set 2 not in set 1
$\lambda[1]$ = size of set 1
$\lambda[2]$ = size of set 2
M = maximum size of set

	set 2 empty	set 1 empty	sets non-empty
Linked list	14	$73 + 46*\lambda[2]$	$24+14*\lambda[1]+(10+47\pi)\lambda[2]$
AVL tree	14	$67 + 59*\lambda[2]$	$15 + 63\lambda[2] + 166\pi\lambda[2] + 17\pi\lambda[2]*LOG2(\lambda[1])$
Bit - Array	25 $11*\lceil M/32\rceil$	$125 + 14*\lceil M/32\rceil$	$25 + 11*\lceil M/32\rceil$
Hash table	14	$649+33\lambda[2] - 128\pi(31/32)\uparrow\lambda[2]$	$251 + 224(31/32)\uparrow\lambda[1] + 14\lambda[1] + (50\pi+10)\lambda[2]$
Bit-string with unsorted linked list	20	$252 + 46*\lambda[2] + 6*\lceil M/32\rceil$	$40 + 21*\lceil M/32\rceil + 66\pi\lambda[2]$
Attribute bit	∞		
Sorted variable length array	17	$400 + 6\lambda[2]$	$581 + 19\lambda[1] + (8+15\pi)\lambda[2]$

SET_INTERSECTION - intersection of two sets
π = proportion of set 1 not in set 2
λ[1] = size of set 1
λ[2] = size of set 2
M = maximum size of set

	set1 empty	set2 empty	normal	result empty
Linked list	11	43	36+(14+24π)*λ[1] + 12*λ[2]	51 + 38*λ[1] + 12*λ[2]
AVL tree	13	24+32*λ[1]	51 + 30.5λ[2] + 30λ[1] + 209πλ[1] +20πλ[1]LOG(λ[1])	51+30.5λ[2]+ 239λ[1] + 20λ[1]LOG2(λ[1])
Bit - Array	23 + 11*⌈M/32⌉			
Hash table	11	843 - 384*π* (31/32)↑λ[1]	626 - 320(31/32)↑λ[1] + 17λ[2] + (6+31π)λ[1]	833 - 296(31/32)↑λ[1] +17λ[2]+37λ[1]
Bit-string with unsorted linked list	15	123	52+11*⌈M/32⌉ +(40+20π)λ[1]	136+11*⌈M/32⌉ + 60λ[1]
Attribute bit	∞			
Sorted variable length array	11	211	42 +(14+π)λ[1] + 8λ[2]	239 + 15λ[1] + 8λ[2]

SET_SUBTRACTION - difference of two sets
π = proportion of set 2 in set 1
$\lambda[1]$ = size of set 1
$\lambda[2]$ = size of set 2
M = maximum size of set

	set1 empty	set2 empty	normal	result empty
Linked list	11	14	$20+(12+14\pi)*\lambda[2]$ $+ 16*\lambda[1]$	$41 + 30*\lambda[1]$ $+ 12*\lambda[2]$
AVL tree	11	14	$13 + 82.5\lambda[2] +$ $88\pi\lambda[2] +$ $20\pi\lambda[2]LOG(\lambda[1])$	$13 + 82.5\lambda[2]$ $+ 88\lambda[2] +$ $20\lambda[2]LOG(\lambda[1])$
Bit - Array	$23 + 11*\lceil M/32 \rceil$			
Hash table	11	14	$466 + 16\lambda[1] -$ $160(31/32)\uparrow\lambda[1]$ $+(15+7\pi))\lambda[2]$	$665 + 23\lambda[1] -$ $130(31/32)\uparrow\lambda[1]$ $+ 15\lambda[2]$
Bit-string with unsorted linked list	15	18	$52+11*\lceil M/32 \rceil$ $+ 40\lambda[1] +$ $+ 20\pi\lambda[2]$	$136+11*\lceil M/32 \rceil$ $+60\lambda[1]$
Attribute bit	∞			
Sorted variable length array	11	14	$33 + 19\lambda[1] +$ $(7-9\pi)\lambda[2]$	$230 + 10\lambda[1] +$ $7\lambda[2]$

SET_EQUALITY - boolean true if sets equal
π = proportion of time boolean true
$\lambda[1]$ = size of set 1
$\lambda[2]$ = size of set 2
M = maximum size of set

	lengths ≠	both empty	otherwise
Linked list	19	18	$32 - 3\pi + 9*(1 + \pi)*\lambda$
AVL tree	25	24	$41 + 27.25*(1 + \pi)*\lambda + 15\pi$
Bit - Array	$26 - 2\pi + (5 + 5\pi)*\lceil M/32 \rceil$		
Hash table	18	19	$32 + 327\pi + 18\pi\lambda$
Bit-string with unsorted linked list	46	39	$62 - 2\pi + (5 + 5\pi)*\lceil M/32 \rceil$
Attribute bit	∞		
Sorted variable length array	22	19	$20 + (5 + 5\pi)\lambda + 3\pi$

SET_INCLUSION - boolean true if set1 contained in set2
- π = proportion time boolean true
- $\lambda[1]$ = size of set 1
- $\lambda[2]$ = size of set 2
- M = maximum size of set

	lengths bad	set 1 empty	standard case
Linked list	17	21	$32 + 8*(1+\pi)*\lambda[1] + 6*(1+\pi)*\lambda[2]$
AVL tree	25	24	$49 - 3\pi + 16.25\lambda[1](1+\pi) + 15.25\lambda[2](1+\pi)$
Bit - Array	$25 + (6+6\pi)*\lceil M/32 \rceil$		
Hash table	18	21	$30 + 327\pi + 10\pi\lambda[1] + 12\pi\lambda[2]$
Bit-string with unsorted linked list	46	22	$51 + (6+6\pi)*\lceil M/32 \rceil$
Attribute bit	∞		
Sorted variable length array	26	13	$44-3\pi + (9+9\pi)\lambda[1]/2 + (7+7\pi)\lambda[2]/2$

SET_PROPER_INCLUSION - boolean true if set1 contained in set2 but not equal
π = proportion of time boolean true
λ[1] = size of set 1
λ[2] = size of set 2
M = maximum size of set

	lengths bad	set 1 empty	standard case
Linked list	16	31	$42 + 8(1+\pi)\lambda[1] + 6(1+\pi)\lambda[2]$
AVL tree	16	34	$59 - 3\pi + (16.25\lambda[1] + 15.25\lambda[2])(1+\pi)$
Bit - Array	$27 + (9+9*\pi)*\lceil M/32 \rceil$		
Hash table	16	31	$40 + 327\pi + 10\pi\lambda[1] + 12\pi\lambda[2]$
Bit-string with unsorted linked list	46	39	$51 + (6+6\pi)*\lceil M/32 \rceil$
Attribute bit	∞		
Sorted variable length array	22	29	$60 - 3\pi + (9+9\pi)\lambda[1]/2 + (7+7\pi)\lambda[2]/2$

COPY_SET - form a copy of a set
λ = size of set
M = maximum size of set

	set empty	set non-empty
Linked list (sorted)	11	$49 + 46*\lambda$
AVL tree	15	$48 + 59*\lambda$
Bit - Array	19	$118 + 6*\lceil M/32 \rceil$
Hash table (32 slots)	15	$633-128(31/32)\uparrow\lambda + 33\lambda$
Bit-string with unsorted linked list	19	$227 + 46\lambda + 6*\lceil M/32 \rceil$
Attribute bit	∞	
Sorted variable length array	15	$384 + 6\lambda$

RELEASE SET - release the storage occupied by a set
λ = size of set
M = maximum size of set

	set empty	set non-empty
Linked list (sorted)	11	22
AVL tree	11	$1+32\lambda$
Bit - Array	12	50
Hash table (32 slots)	11	$818 - 384*(31/32)\uparrow\lambda$
Bit-string with unsorted linked list	15	92
Attribute bit	∞	
Sorted variable length array	9	191

FOREACH LOOP (including initialization) through a set
λ = size of set
M = maximum size of set

	set empty	set non-empty
Linked list (sorted)	108	$135 + 34*\lambda$
AVL tree	87	$101 + 66.5\lambda$
Bit - Array	$141 + 65*\lambda + 13*\lceil M/32 \rceil$	
Hash table (32 slots)	117	$1227+34\lambda-448(31/32)\uparrow\lambda$
Bit-string with unsorted linked list	175	$214+40\lambda$
attribute bit	∞	
Sorted variable length array	95	$304 + 38\lambda$

PUT_INDEXED - insert into list
λ = size of list

	list empty	list non-empty
One-way linked list	93	$71 + 7\lambda/2$
Two-way linked list	108	$92 + 5\lambda/4$
Variable length array	151	$95 + 3.3\lambda$

PUT_AFTER - insert into list after specific item
λ = size of list

	list non-empty
One-way linked list	$71 + 6\lambda$
Two-way linked list	$81 + 4\lambda$
Variable length array	$97 + 5.8\lambda$

PUT_BEFORE - insert into list before specific item
λ = size of list

	list non-empty
One-way linked list	63 + 6λ
Two-way linked list	81 + 4λ
Variable length array	100 + 5.8λ

REMOVE_INDEXED - remove the n th element of a list
λ = size of list

	result list empty	result list non-empty
One-way linked list	89	55 + 7λ/2
Two-way linked list	93	68 + 5λ/4
Variable length array	221	24 + 1.5λ

REMOVE_ITEM - remove first occurence of specified item from list
λ = size of list
π = proportion of time item in list

	list empty	not only item in list	only item in list
One-way linked list	12	$13 + 12\lambda - 6\pi\lambda + 34\pi$	88
Two-way linked list	12	$12 + 8\lambda - 4\pi\lambda + 39\pi$	79
Variable length array	12	$26 + 4.5\pi + (5-\pi)\lambda$	231

REMOVE_ALL_ITEM - remove all occurences of specified item from list
λ = size of list
π = proportion of list that is item

	list empty	not only item in list	only item in list
One-way linked list	12	$20 + 12\lambda + 40\pi\lambda$	$49 + 52\lambda$
Two-way linked lis	12	$18 + 8\lambda + 41\pi\lambda$	$34 + 49\lambda$
Variable length array	10	$28 + 16\lambda - 4\pi\lambda - 6\pi$	$222 + 12\lambda$

COPY_LIST - make a copy of a list
λ = size of list

	list empty	list non-empty
One-way linked list	13	$51 + 46*\lambda$
Two-way linked list	15	$58 + 54\lambda$
Variable length array	17	$386 + 6\lambda$

RELEASE_LIST - release space occupied by a list
λ = size of list

	list empty	list non-empty
One-way linked list	13	24
Two-way linked list	11	23
Variable length array	9	191

COP_LIST - return first element of list
λ = size of list

One-way linked list	17
Two-way linked list	15
Variable length array	12

LOP LIST - return and remove first element of list
λ = size of list

	result list non empty	result list empty
One-way linked list	50	67
Two-way linked list	46	76
Variable length array	22 + 3λ	219

CONCATENATION - concatenate two lists together
λ = size of list

	list 1 empty	list 2 empty	neither empty
One-way linked list	16	14	52
Two-way linked list	16	14	59
Variable length array	14	11	$731 + 6\lambda[1] + 6\lambda[2]$

LIST_EQUALITY - boolean true if lists equal
π = proportion of time boolean true
$\lambda[1]$ = size of list 1
$\lambda[2]$ = size of list 2

	lengths ≠	both empty	otherwise
One-way linked list	21	20	$34 - 3\pi + (9+9\pi)\lambda$
Two-way linked list	20	18	$24 + (4+4\pi)\lambda - 3\pi$
Variable length array	24	21	$22 + (5+5\pi)\lambda + 3\pi$

LIST_MEMBERSHIP - boolean true if item an element of the list
π = proportion of time item in non-empty list
λ = size of list

	list empty	list non-empty
One-way linked list	15	$19 + 11\lambda - 5.5\pi\lambda$
Two-way linked list	15	$19 + 3\pi + 8\lambda - 4\pi\lambda$
Variable length array	15	$23 + 3\pi/2 + (5-5\pi/2)\lambda$

FOREACH LIST - foreach item in list, initialization and iteration
λ = size of list

	list empty	list non-empty
One-way linked list	112	$139 + 38\lambda$
Two-way linked list	92	$123 + 49\lambda$
Variable length array	99	$308 + 40\lambda$

FETCH_INDEXED - fetch the n th element of the list
λ = size of list

	list non-empty
One-way linked list	$17 + 5\lambda/2$
Two-way linked list	$35 + 5\lambda/4$
Variable length array	20

REPLACE_INDEXED - replace the n th element of the list
λ = size of list

	list empty	extend list	replace	replace last
One-way linked list	97	79	$23 + 5\lambda/2$	34
Two-way linked list	113	93	$42 + 5\lambda/4$	45
Variable length array	153	$97 + .6\lambda$	25	25

LENGTH_LIST - returns number of elements in list
λ = size of list

	list empty	list non-empty
One-way linked list	9	9
Two-way linked list	9	12
Variable length array	11	14

EXPLICIT_LIST - make an explicit list
λ = size of list

	list non-empty
One-way linked list	$63 + 43\lambda$
Two-way linked list	$73 + 51\lambda$
Variable length array	$379 + 3\lambda$

SECTION 9

REFERENCES

[Allen69] F Allen. *Program Optimization*
Annual Review in Automatic Programming Vol 5 p 239-307. 1969

[Anderson72] B Anderson. *Programming Languages For Artificial Intelligence: The role of nondeterminism.*
School of Artificial Inteligence, Univ. of Edinburgh Experimental Programming Reports No 25 March 1972.

[Baumgart72] B Baumgart. *Micro Planner Alternate Reference Manual*
Stanford Artificial Intelligence Laboratory, Operating Note 67 Apr 1972.

[Balzer67] R Balzer. *Dataless Programming.*
AFIPS Proceedings of FJCC 1967 p 535-544.

[Balzer72] R Balzer. *Automatic Programming.*
Institute Technical Memo, University of Southern California, Information Sciences Institute. Sep 1972.

[Bobrow73a] D Bobrow and B Raphael Bertram. *New Programming Languages for AI Research.*
Tutorial presented at Third International Joint Conference on Artificial Intelligence. Stanford Aug 1973.

[Bobrow73b] D Bobrow and B Wegbreit. *A Model and Stack Implementation of Multiple Environments.*
CACM vol 16, no 10 (Oct 73).

[Brent73] R Brent. *Reducing the Retrieval Time of Scatter Storage Techniques.*
CACM vol 16, no 2. (Feb 1973).

[Cocke70] J Cocke and J Schwartz. *Programming Languages and Their Compilers.*
NYU Courant Institute. April 1970

[Codd70] E Codd. *A Relational Model of Data for Large Shared Data Banks.*
CACM vol 13, no 6. (June 1970).

[Crane72] C Crane. *Linear Lists and Priority Queues As Balanced Binary Trees.*
PH. D. Thesis Stanford Computer Science Department Technical Report CS 259. February 1972.

[Crick70] M Crick and A Symonds. *A Software Associative Memory For Complex Data Structures.*

IBM Cambridge Scientific Center. Report No. G320-2060. Aug 1970.

[DEC69] PDP-10 SYSTEM REFERENCE MANUAL. Digital Equipment Corporation. 1969.

[Delobel73] C Delobel and R Casey. *Decomposition of a Data Base and the Theory of Boolean Switching Functions.*
IBM Journal of Research and Development. Sep 1973.

[Derksen72] J Derksen. *The QA4 Primer.*
Stanford Research Institute, June 1972.

[Earley71a] J Earley. *Comments on SETL (Symmetric Use Of Relations).*
SETL Newsletter 52. Courant Institute NYU. Sept 1971

[Earley71b] J Earley. *Toward an Understanding of Data Structures.*
CACM vol 14, 10 (Oct 1971)

[Earley73a] J Earley. *Relational Level Data Structures For Programming Languages.*
Computer Science Department, University of California, Berkeley. March 1973.

[Earley73b] J Earley. *An Overview of the VERS2 Project.*
Electronics Research Laboratory, College of Engineering, University of Calif, Berkeley Memorandum ERL-M416 Dec. 1973.

[Earley74a] J Earley. *High Level Iterators and a Method of Automatically Designing Data Structure Representation*
Electronic Research Laboratory, College of Engineering Memorandum ERL-M416, Feb 1974. University of Calif, Berkeley.

[Earley74b] J Earley. *High Level Operations In Automatic Programming.*
SIGPLAN Notices, Vol 9, No 4. (April 1974).

[Elias] P Elias. *Efficient Storage And Retrieval By Content and Address of Simple Files.*
MIT Department of EE & Research Laboratory of Electronics. No Date.

[Feldman69] J Feldman and P Rovner. *An Algol-Based Associative Language.*
CACM vol 12 no. 8. August 1969.

[Feldman72a] J Feldman. Automatic Programming
Technical Report C. S. 255 Stanford Computer Science Dept. Stanford University. Feb. 1972.

[Feldman72b] J Feldman, J Low, D Swinehart, and R Taylor. *Recent Developments in SAIL - An Algol Based Language For Artificial Intelligence.*
Proceedings of FJCC 1972. p 1193-1202.

[Feldman73] J Feldman and J Low. *Comment on Brent'S Scatter Storage Algorithm.*
CACM vol 16, no 11 (November 1973).

[Foster65] C Foster. *Information Storage and Retrieval Using AVL Trees.*
Proceedings ACM National Conference. 1965. p 192-205

[Foster73] C Foster. *A Generalization of AVL Trees.*
CACM vol 16 no 8 (Aug. 1973).

[Geschke72] C Geschke. *Global Program Optimizations.*
PH.D thesis Department of Computer Science, Carnegie-Mellon University. Oct 1972.

[Harrison72] M Harrison. *Data Structures And Programming.*
Revised Version. Courant Institute of Mathematical Studies. New York University. February 1972.

[IBM69] IBM, *System/360 Operating System: Assembler Language.*
Systems Refererence Library C28-6514-6. June 1969

[Ingalls71] D Ingalls. *FETE - A FORTRAN Execution Time Estimator*
Stanford Computer Science Department Report C. S. 204 Feb 1971.

[Johnson] T Johnson. *A Mass Storage Relational Data Structure For Computer Graphics and other Arbitrary Data Stores.*
MIT Department of Architecture and Department of Civil Engineering NSF contract GK-365, MIT Project No. DSR 74684. No Date.

[Kildall 72] G Kildall. *Global Expression Optimization During Compilation.*
PH.D thesis, Department of Computer Science, University of Washington. TR 72-06-02. June 1972

[Knuth68] D Knuth. *FUNDAMENTAL ALGORITHMS: The Art of Computer Programming Vol I.*
Addison-Wesley 1968.

[Knuth73] D Knuth. *SORTING AND SEARCHING: The Art of Computer Programming Vol III.*
Addison-Wesley 1968.

[Knuth71] D Knuth. *An Empirical Study Of FORTRAN Programs.*
SOFTWARE - PRACTICE AND EXPERIENCE, VOL I. p 105-133. Wiley-Interscience 1971.

[Knuth74] D Knuth. *Structured Programming With* Go To *Statements.*
Stanford Computer Science Department. Report STAN-CS-74-416. May 1974.

[Madnick67] S Madnick. *String Processing Techniques.*
CACM July 1967.

[Maurer68] W Maurer. *An Improved Hash Code for Scatter Storage.*
CACM vol 11, no 1 (Jan 68).

[McDermott72] D McDermott and G Sussman. *The CONNIVER Refererence Manual.*
AI Memo No. 259. MIT May 1972.

[Minter72] J Minter. *Associative Memories and Processors: A descritive appraisal.*
TR 195 Univ of Maryland, Computer Science Center, College Park, Maryland. July 1972.

[Minsky72] N Minsky. *Rotating Storage Devices As Partially Associative Memories.*
Technical Report 72-4. Computer Information and Control Sciences. Univerity of Minnesota. April 28, 1972.

[Morris73] J Morris. *A Comparison of MADCAP and SETL.*
University of California, Los Alamos Scientific Laboratory. 1973

[Morris68] R Morris. *Scatter Storage Techniques.*
CACM vol 11, no 1(Jan 68)

[Parhami72] B Parhami. *RAPID: A Rotating Associative Processor For Information Dissemination.*
UCLA - ENG 7213 Feb. 1972. Comp. Sci Dept.

[Randall71] S Randall. *A Relational Model of Data for the Determination of Optimum Computer Storage Structures.*
Department of Electrical Engineering, Systems Engineering Laboratory Tech Report 54. University of Michigan, Ann Arbor Sept 1971

[Rivest74] R Rivest. *Analysis of Associative Retrieval Algorithms.*
Ph. D. Thesis Stanford Computer Science Department 1974.

[Schwartz71] J Schwartz. *More Detailed Suggestions Concerning "Data Strategy" Elaboration For SETL.*
SETL Newsletter 39. NYU Courant Institute. May 1971.

[Schwartz74a] J Schwartz. *Automatic And Semiautomatic Optimization in SETL.*
SIGPLAN Notices, vol 9, no 4. (April 1974).

[Schwartz74b] J Schwartz. *Automatic Data Structure Choice in a Language of Very High Level.*
Courant Institute, NYU. 1974.

[Smith73] D Smith and H Enea. *Backtracking in MLISP2.*
Proceedings of the Third IJCAI. 1973.

[Sussman70] G Sussman, T Winograd, and E Charniak *MICRO-PLANNER Reference Manual.*
AI MEMO 203, Project MAC, MIT July, 1970.

[Sussman72] G Sussman. *Why Conniving Is Better Than Planning*
A. I. Lab. MIT. A. I. Memo 255, FEB, 1972.

[Tesler73] L Tesler, H Enea, and D Smith. *The LISP70 Pattern Matching System.*
Proceedings of the Third IJCAI. 1973.

[Tompa73] F Tompa and C Gotlieb. *Choosing A Storage Schema.*
Technical Report No. 54. May 1973. Department of Computer Science, University of Toronto. Toronto Canada.

[VanLehn73] K VanLehn. *SAIL User Manual.*
Stanford Computer Science Technical Report STAN-CS-73-373. July 1973.

[Wegbreit71] B Wegbreit. *The Treatment of Data Types in EL1.*
Harvard University. 1971

[Wegbreit73] B Wegbreit. *Procedure Closure in EL1.*
TR 13-73 Center for Research in Computing Technology Harvard University May 1973.

[Wegbreit74] B Wegbreit. *Mechanical Program Analysis.*
Xerox Palo Alto Research Center July 1974.

[Wichman72] B Wichman. *Estimating the execution speed of an ALGOL program.*
SIGPLAN Notices. vol 7, no 8. Aug 1972.

[Wulf73] W Wulf, R Johnsson, C Weinstock, and S Hobbs. *The Design of An Optimizing Compiler.*
Computer Science Department. Carnegie-Mellon University. Pittsburgh, Pennsylvania Dec 1973.