Programming Languages
J. J. Horning Editor

Automatic Data Structure Selection: An Example and Overview

James R. Low
The University of Rochester

The use of several levels of abstraction has proved to be very helpful in constructing and maintaining programs. When programs are designed with abstract data types such as sets and lists, programmer time can be saved by automating the process of filling in low-level implementation details. In the past, programming systems have provided only a single general purpose implementation for an abstract type. Thus the programs produced using abstract types were often inefficient in space or time. In this paper a system for automatically choosing efficient implementations for abstract types from a library of implementations is discussed. This process is discussed in detail for an example program. General issues in data structure selection are also reviewed.

Key Words and Phrases: abstract data types, automatic programming, data structures, optimizing compilers, sets, lists

CR Categories: 4.12, 4.22, 4.6

This research was supported in part by a grant from the Alfred Sloan Foundation and also by a grant from the National Science Foundation, grant number MCS76-10825.

Author's address: Department of Computer Science, The University of Rochester, Rochester, NY 14627

1. Introduction

In the past several years computer scientists have realized that the use of several levels of abstraction in programming often gives improved results in terms of clarity, correctness, and ease of maintenance of the resulting programs. One important aspect of abstraction is the use of data types such as stacks, queues, sets, and so forth. The properties of abstract types can often be described axiomatically [20, 21]. This aids in constructing formal or informal proofs of program correctness. Several systems (CLU [13], SIMULA-67 [3]) have gone further and shown the benefits of "hiding" the low-level implementations of the abstract types. Thus a programmer might deal with a stack only in terms of a small number of primitives upon it such as PUSHing an element, POPing an element off the stack, testing if the stack were EMPTY, and so forth. The programmer would not have to concern himself with whether the underlying representation were a linked list, an array, or some hybrid.

When a program is written using only primitives on an abstract type without using any knowledge of the underlying representation, the underlying representation may be changed without affecting the operation of the program except possibly for resource requirements.

Given a programming system which provides the abstract type *stack,* a natural thing to do would be to have a number of alternative implementations of stacks and choose the implementation which is most efficient (by some criteria) for any individual program. An "intelligent" compiler would pick the "best" set of implementations for the abstract types of an individual program from a fixed library of implementations for the types.

For the purpose of this paper, we define "abstract data types" to mean SIMULA-like classes which the user sees only as a number of well-defined primitives such as PUSH and POP mentioned above. The term "abstract data structure" will mean a particular instantiation of an abstract type (e.g. a particular stack). The programmer will not assume any particular representation for the abstract data structure and will manipulate it using only the given primitives.

A prerequisite for selecting a good representation for an abstract data structure is a rich library of implementations for the various abstract types. Cost formulas for these primitive operations performed on the data structure must also be available. The costs will typically be functions of the size of the data structure and other properties of the data. This library of implementations and cost formulas need only be constructed once for the programming system, but the implementation specifications will be used in the compilation of each program. There are several steps in the process of selecting good representations of the data for a given program. The compiler must determine how the data

structures are used within the program, their domain, their size, what operations are performed upon them, and so forth. In the prototype system described below, static flow analysis, monitoring executions of sample runs, and user interrogation provide this information. Given the above information, the system uses an implementation selection algorithm which attempts to minimize the cost (according to some fixed criteria) of executing the program. An iterative technique is used for selecting representations in the prototype system described below.

In the remainder of this paper we discuss a prototype system which has been used successfully for selecting representations for sets and sequences. The selections made by this prototype system compare favorably with those made by a human programmer. The selections have differed from those that would have been made by the author in only a few cases. Those cases were typically of two types: the system picked a more complex representation (such as AVL tree or hash table) when its utility was only marginally better than that of a simpler representation (such as bit vector or linked list); the human programmer used information not available to the automatic selection system. This prototype system could easily be extended to select representations for other reservoir-like abstract types such as queues and stacks. Later we shall give an overview of the problems which must be considered in making a practical data structure selection system.

2. Prototype System

A prototype system for semiautomatic selection of representations was constructed for a subset of the SAIL programming language [8, 16]. SAIL is an Algol-60 based language with extensions including varying length strings, processes, event mechanisms, and an associative data language called LEAP [7].

We were interested in choosing representations for the abstract data structures of LEAP. The basic entities manipulated by LEAP are called "items." An item is a symbolic object much like a LISP atom. A variable, called a datum, may optionally be attached to the item. Datums can be either simple variables or arrays. Items are often categorized by the data type of the attached datum. Thus we have integer items (items with an integer variable as datum), set items (items with a set variable as datum), untyped items (items with no datum), and so forth. Items are allocated either at compile time via declarations or dynamically from a heap at execution time (via the built-in function called NEW). The lifetime of items and their datums does not follow Algol block structure. An item and its datum exist until explicitly deleted by the use of the DELETE procedure.

LEAP also contains "sets" of items, "lists" of items, and a ternary relation among items. Set and list values have the same lifetime as arithmetic quantities. There are set expressions, set variables local to a block, and set datums. The ternary relation is global to the entire program.

The subset of SAIL which was used in the prototype system included integers, reals, strings, procedures, items, sets, and lists. Explicitly excluded from the subset were arrays, recursive procedures, go-to statements, multiprocessing, events, and the ternary relation of LEAP. A recent extension to the system [17] now performs selection of representations for the ternary relation.

In the prototype system we chose representations only for sets of items and lists of items. A set is a conceptually unordered collection of items in which any given item appears at most once. Operations built into SAIL to manipulate sets include: union, intersection, difference, element insertion, element deletion, test for membership, and iterating through the members of a set expression. A list is a linear sequence of items. It behaves much as a flexible length array or string of items except that there are primitives for inserting or deleting items at arbitrary positions within the list. Operations on lists include: fetching or storing an item by ordinal position within the list; concatenating two lists; removing items from a list (either by ordinal number within the list or by giving the name of the item to be removed); inserting items into the LIST (before or after an ordinal position or a particular item); and iterating through the elements of a list.

Let us now look at an example of a program written with sets and lists and see how the prototype system chose representations for it.

Example

The program (see Figure 1) computes a spanning tree for a graph. The graph consists of a set of nodes (NODESET) and a set of undirected edges between pairs of nodes (EDGES). The program assumes that there is a path (through zero or more other nodes) between every pair of nodes. A spanning tree for the graph consists of a subgraph containing all the original nodes and a subset of the edges of the original graph such that:

(1) For any pair of distinct nodes there exists a path between the nodes.

(2) There is no path from a node to itself (subgraph is cycle-free).

The basic algorithm is the EQUIVALENCE algorithm given by Knuth [11, Algorithm 2.3.3E] modified to record the edges used. The technique used is to partition the nodes into groups. Each group will have the property that there exists a path from each node to every other node in the group. The program starts by placing each node in a separate group. It then looks at each edge. If the edge connects two nodes in the same group, then the program ignores that edge. Otherwise it records the edge in the set of edges which will

Fig. 1.

```
1.     (1) begin "SPANNING TREE"
2.           set EDGES, NODESET, SETOFGROUPS,
                     TREESET;
3.           list itemvar FIRSTGROUP,SECONDGROUP,
                                EDGETEMP,
                     FIRST,SECOND,X;
4.           list itemvar procedure GROUPOF
                     (list itemvar NODE);
5.   (144)   begin "GROUPOF"
6.   (557)     while (FATHER(NODE) neq NODE) do
7.   (413)       NODE := FATHER(NODE);
8.   (144)     return(NODE);
9.     (0)   end "GROUPOF";
10.          procedure MERGEGROUP
                     (list itemvar GROUP1,GROUP2);
11.   (24)   begin "MERGEGROUP"
12.   (24)     remove GROUP2 from SETOFGROUPS;
13.   (24)     FATHER(GROUP2) := GROUP1;
14.   (24)   end "MERGEGROUP";
15.        comment AT THIS POINT WOULD HAVE CODE
                 TO EITHER BUILD THE GRAPH INLINE
                 OR READ THE GRAPH FROM AN INPUT
                 DATA SET
                 ...
                 THE RESULT IS THE SET OF NODES OF
                 THE GRAPH (NODESET), AND THE SET
                 OF EDGES (EDGES);
           comment EACH NODE INITIALLY ITS OWN
                 GROUP;
16.    (1)   foreach X such that X in NODESET do
17.   (25)     FATHER(X) := X;
18.    (1)   SETOFGROUPS := NODESET;
19.    (1)   foreach EDGETEMP such that
                     EDGETEMP in EDGES do
20.   (72)   begin
21.   (72)     FIRST := END1(EDGETEMP); SECOND :=
                     END2(EDGETEMP);
22.   (72)     FIRSTGROUP := GROUPOF(FIRST);
23.   (72)     SECONDGROUP := GROUPOF(SECOND);
24.   (72)     if FIRSTGROUP neq SECONDGROUP then
25.   (24)       begin
26.   (24)         put EDGETEMP in TREESET;
27.   (24)         MERGEGROUP(FIRSTGROUP,
                     SECONDGROUP);
28.   (24)       end;
29.   (72)   end;
30.    (1)   print ("EDGES OF SPANNING TREE");
31.    (1)   foreach EDGETEMP such that
                     EDGETEMP in TREESET do
32.   (24)     begin
33.              string itemvar NODENAME1, NODENAME2;
34.   (24)       FIRST := END1(EDGETEMP);
                     SECOND := END2(EDGETEMP);
35.   (24)       NODENAME1 := NAME(FIRST);
                     NODE-
                     NAME2 := NAME(SECOND);
36.   (24)       print(datum(NODENAME1),"
                     datum(NODENAME2));
37.   (24)     end;
38.    (1) end "SPANNING TREE"
```

constitute the spanning tree (TREESET) and merges the two groups into one.

The groups are represented by trees consisting of the nodes within the group. Given any node in such a tree, the root of that tree may be found by following the chain of FATHER links. The FATHER link from the root node of a tree will be to itself. The root of the tree is used to name the group. Two nodes of the graph are in the same group if the roots of their group trees are the same. To merge two groups the program simply finds both root nodes and changes the FATHER link of one root node to point to the other root node.

The program has the following abstractions:

1. Nodes
2. Edges
3. A map from an edge to the two nodes connected by the edge.
4. A map from a node to its father in the tree representing a group of connected nodes.
5. A set of all the nodes.
6. A set of all the edges.
7. A set of the edges making up the spanning tree.

We represent the NODES by items, the EDGES by items, the maps by specific positions within list datums of NODE and EDGE items, and the sets by SAIL sets.

For the maps we have the following:

DATUM(NODE) (1) is an item with a string datum for the name of the node.

DATUM(NODE) (2) is the node which is the FATHER of this NODE in its group tree.

DATUM(EDGE) (1) is one node which is an endpoint of the edge.

DATUM(EDGE) (2) is the other endpoint of the edge.

For notational simplicity we assume that there are macros: NAME(X), which expands to DATUM(X)[1]; FATHER(X), which expands to DATUM(X)[2]; END1(X), which expands to DATUM(X)[1]; and END2(X), which expands to DATUM(X)[2].

Note that in full SAIL (not the subset implemented by the prototype system) we would probably implement the maps by using the ternary relation.

The reader is advised to look at the Appendix for a description of the syntax of SAIL used in the following example. In Figure 1 the numbers to the left are for reference only and would not appear in the source text.

The prototype system consists of several phases. The first phase involves monitoring of the program (using default implementations of the abstract data structures) to determine the frequency of the various operations. The second phase involves a static flow analysis of the program to determine the range of each set or list expression and also to determine which operations are performed on the individual sets and list. The third phase is user interrogation in which the programmer is asked questions about attributes of the set and list operations which may alter choice of representation. The fourth phase is the selection of representations using a method similar to hill climbing. The final phase consists of compiling and executing the program again, this time using the representations selected by the fourth phase.

Monitoring

The monitoring phase consists of having the user compile his program with a special version of the SAIL compiler. The special compiler inserts counters into

the generated code that will enable the system to determine how many times each statement in the program was executed. This compiler uses the standard SAIL representations for sets and lists (linked lists). The user then runs the program with an input data set he considers typical. The counts of the statements for a typical run of the example are included in parentheses to the left of the program in Figure 1. For this particular execution of the program a graph with 25 nodes and 72 edges was used. Monitoring in the prototype system consists only of keeping statement counters.

Flow Graph

The program source text and the file containing the values of the counters above are then processed by a program which builds a flow graph of the user program. Each node of the graph represents either some control information (such as a procedure call or a beginning of a loop) or a LEAP operation. The flow graph does not include expressions or statements dealing with non-LEAP entities unless the statement also uses a LEAP construct such as DATUM. The costs of non-LEAP constructs are considered separately during the selection phase. Associated with each node is the value of the counter indicating how many times the corresponding statement was executed in the sample run. The system uses this flow graph to analyze the user's program to determine how the SETS and LISTS are used.

Meta-Evaluation

The next phase of the system involves traversing the flow graph in a form of flow analysis we call "meta-evaluation." There is a data structure (in our system, which is written in SAIL, this takes the form of the ternary relation of LEAP) which contains entries for each possible set variable, list variable, declared item, the class of items allocated by a particular call to the NEW item allocator, possible datums for each item, and so forth. The result of meta-evaluation will be to change this data structure so that it contains information about which of the classes of items (either individual declared items or all the items allocated by a specific NEW) are possible members of each set and list at each point in the program. Information concerning which primitives are applied to each set and list variable or temporary is also derived. The process of meta-evaluation starts by looking at the node of the flow graph representing the entry point to the main program. At each node the meta-evaluator usually performs some operation to update its data structure. Thus at an assignment to a set variable node it determines the possible set of items (declared or NEWs) which can be elements of the set expression on the right-hand side of the assignment and then updates the model of the possible contents of the set variable being assigned to. After updating the data structure, it then meta-evaluates all nodes which are immediate successors to the current node. Normally each node has a single successor, but control nodes such as those representing IF-statements or CASE-statements have more than a single successor. When we reach a node which JOINs two paths in the flow graph (such as following an IF-THEN-ELSE), we compute the sets of potential elements of the variables by merging the sets of potential elements from each path. Loops are repeatedly meta-evaluated until a single meta-evaluation causes no new information to be added to the data structure. The process of meta-evaluation is conservative. It always errs on the side of assuming that the set of potential elements of variables is larger than might actually be true in execution of the program. The meta-evaluator will never say that an item is not an element of the variable when it actually might be.

The meta-evaluation of the example derives information such as: the domain of NODESET and SETOFGROUPS is only the set of items which represent NODES in the graph; the domain of the list datums of the node items is the set of items used for naming nodes and the set of node items; the only operations performed on TREESET are insertion and the foreach iteration; and so forth.

After the meta-evaluation phase the system has a model of the possible domain of each set and list variable, the primitives applied to each variable or expression, and which variables and expressions are operands to particular instances of the binary set and list operators. Note that there are techniques other than meta-evaluation for obtaining the same information. Classical summary flow analysis techniques [2, 10], particularly use-define chaining [1], could be modified to provide the same information.

Partitioning into Equivalence Classes

In the next part of the static analysis phase the system makes classes of variables and expressions for which it will choose the same low-level implementation. Consider a binary set operation such as union. The most general library would need to have implementations taking sets represented in two different ways as arguments to the union and produce a set in a third representation. Thus we might have a union routine taking one set represented by a hash table and uniting it with another set represented by a binary tree, producing a set represented as a bit map. The cost of coding such a library would be exorbitant; the number of concrete implementations of the binary operators varies with the cube of the number of representations. Alternatively, if you allow representation conversions, the number of conversion routines varies with the square of the number of representations. For the prototype system, we made a simplifying assumption that for each implementation of a binary operator the input and output sets (or lists) are restricted to the same representation. So, the size of our library need

only expand linearly with the number of low-level representations. This assumption induces an equivalence relation upon the set and list variables and expressions. Two variables or expressions will be in the same equivalence class if they are operands to the same instance of an operator. Thus, the part of the static analysis phase following meta-evaluation computes these equivalence classes using information derived from meta-evaluation. For the example in Figure 1 this produces five equivalence classes: the class consisting of the list datums of the items representing nodes; the class of the list datums of the items representing edges; the class consisting of the variables SETOFGROUPS and NODESET; the class consisting of the variable EDGES; and the class consisting of the variable TREESET. The variables SETOFGROUPS and NODESET were placed in the same class because of the assignment statement at line 18.

One result of partitioning into equivalence classes is that the number of different selections that the system has to make has been reduced. In sample programs we have noticed that classes typically have five or more variables. The number of decisions is thus reduced by a factor of five. We could, of course, merge all the classes into a single class and then make only one decision about how to represent all sets in a program, but this is too gross an action as it forces sets with completely different properties to be represented in the same manner. Our partitioning scheme allows the most individual choice of representation with the constraint that concrete implementations of the binary operators take as operands and produce as results a single representation.

The system next merges all the attributes (domains, primitives used, etc.) from the individual variables and expressions within a class to obtain the attributes of the class itself. For the example in Figure 1 the system derives that the operations on the class containing the variable EDGES consist only of insertion and iteration (foreach). The same operations are performed on TREESET. Operations on the list datums of the node and edge items are selection and replacement based on ordinal position in the list.

User Interaction Phase

After partitioning we enter a user interaction phase. The user is asked about the parameters of the cost functions of the representations. In the example, the user is asked questions about the individual LEAP statements of his program such as the set insertion in line 26. The user is asked the average size of the set, the probability that the set is empty, and the probability that the item being inserted is already a member of the set. Similar questions are asked about the set iterations of lines 16 and 31 and so forth. Given the values of the parameters along with the frequency counts (obtained through monitoring), the selection phase will be able to compute estimated cost (in cpu time) for the operation with the various representations. Many of the questions currently asked by the system could be answered by additional monitoring. However, there are also questions (such as expected lifetime of new items) which are extremely difficult to determine without very detailed (and expensive) monitoring. Other questions, such as whether the size of a set is bounded by some constant independent of input data, are generally impossible to answer if only monitoring and inference based on monitoring are used. We feel that some form of user interrogation or assertions within the user's program will always be necessary.

Table I. Remove Set – Remove Item from Set.
π = Proportion of Time Item is in the Set
λ = Size of Set.

Representation	Set empty	Set nonempty	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	48	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	27	27
Sorted variable length array	11	$17 + 21.5*LOG2(\lambda) + 3\pi + 3\pi\lambda$	240

Representation Library

The prototype system contains a fixed library of representations for sets and lists. The representations for sets are: sorted linked lists; AVL (height balanced binary) trees; a bucket hash table (with 32 buckets); fixed length boolean arrays (bit arrays); variable length (in multiples of 10 words) sorted array; a boolean field within records for items; and a combination of fixed length boolean array and unsorted linked list. Representations of lists are: one-way linked lists; two-way linked lists; and variable length arrays. The system contains a library of cost functions alluded to earlier. These functions provide estimated costs of each implementation of each primitive operation as a function of such things as set size, probability of booleans being true, and so forth. A typical set of cost functions is given in Table I. In addition to cpu-time cost functions, the system also contains functions which will predict the amount of memory needed for a representation as a function of the maximum and average sizes of the abstract data structure during execution. The questions asked during the user interaction phase were to provide the parameters to these time and space cost functions.

Selection

The final phase is selection. This phase starts with the system determining which representations are applicable for each equivalence class. A representation is not applicable if the class uses primitives which are not coded for the representation or if the representation needs information at compile time which is not availa-

ble. In our example the system could not use fixed length bit arrays for the sets EDGES and TREESET because the maximum size of the sets depended on input data. After applicability is determined, the system asks the user if he wants to pick representations for some of the equivalence classes himself. The user is presented with applicable representations and may either pick one for the class or let the system later automatically pick one.

Our criterion for automatic selection is to minimize the expected space-time product for the execution of the resulting program. To optimize the total space-time product the system *cannot* simply minimize the space-time product for individual classes because the terms involving the space used in storing one class are multiplied by the time used in execution of primitives on another. The selection process uses an iterative technique similar to hill climbing. It first computes the space and time costs for each class in each program segment (the main program and the individual procedures). If any representation uses both less time and less space than any other representation for the class then it is picked immediately. In our example, the list classes corresponding to the datums of node and edge items are thus best represented by one-way linked lists. The class containing the variables SETOFGROUPS and NODESET is similarly best represented by a sorted linked list. Though there is not a single best representation, according to this criterion for EDGES the domination test (better time and space considered individually) narrows the choices for EDGES to using either a linked list or hash table representation. The choices for TREESET are similarly reduced to either using a linked list or an AVL tree. If there is not a single choice the system computes the total space-time product for each remaining representation for each class considered by itself (ignoring cross-terms with other classes). For the initial guess of the best representation for each class the system picks the representation which minimizes this space-time product. In the example this causes the classes for both TREESET and EDGES to be represented initially by sorted linked lists.

Now the system attempts to improve upon this choice by considering the cross-terms between classes. The system tries changing the representation for a single class to see if it can get an improvement in the total expected cost. It continues to attempt this for each class in turn until it has tried to change the representations of every class and has failed to get an improved cost. Thus, if any representation is changed, we will then cycle through all the other classes and attempt to find better representations for them as well. The process terminates when after having picked a new representation (best given the representations of the other classes) we cannot find a better representation (in the sense of decreasing total cost of the program) for any other class. Note that this technique has the problem normally associated with hill climbing, that of finding local minima. A good though not necessarily optimal assignment of representations to classes is thus obtained. In our example the representation for TREESET is changed to become the AVL tree and the representation for EDGES is changed to become a hash table. It is interesting to note that EDGES and TREESET are identical as far as the program is concerned except for size and the number of times that they are operated upon. They both have the same operations (insertion and iteration) performed on them. The size of EDGES on the average was 72 (after initialization staying constant in size) while the size of TREESET was 12 (starting as empty and growing to size 24 over the life of the program). The number of insertions and iterations involving EDGES was several times larger than for TREESET. Thus we see that the system was able to choose different data representations for quite similar sets based upon the size and frequency of operations performed upon them.

Lastly, a compiler takes the choices made during selection and compiles the appropriate code for every set and list operation based upon those choices.

3. Overview of Issues in Automatic Data Structure Selection

The construction and use of the prototype system has demonstrated to us many of the components necessary for a practical automatic data structure selection system. We have also seen many problems which will have to be solved in order to construct a very good system. In this section we will address some of the issues of language design and system organization that we feel should be considered by the implementor.

Language Constructs

Typical abstract types that we would expect to have built into a high-level programming language would include various kinds of reservoirs of data objects such as stacks, sequences, first-in-first-out queues, sets, and priority queues. Other built-in types should include n-ary relations and general mappings from n-tuples into k-tuples. Special purpose languages would have other built-in abstract types peculiar to their problem domains. For example, a language dealing with linear equations would likely have matrices. A good general purpose high-level language should include extension mechanisms to allow users to implement their own abstract types.

Criteria for Selections

In order to make decisions about which of many implementations to use for a given abstract data structure, we need to have objective criteria for evaluating tradeoffs. Normally we can fix some notion of expected

"cost" of executing a program. Our criterion for choosing among data representations will be to minimize that expected cost.

The cost of running a program is usually a function of the total real time needed to execute the program, the primary memory requirements, secondary memory requirements, I/O channel usage, CPU usage, costs associated with various peripherals, and so forth. The particular function will vary from installation to installation and often will depend on such factors as priority associated with the job or time of day. Cost functions normally contain too many parameters for us to fully optimize. Our difficulties are compounded by the fact that the choice of representations for the different abstract data structures within a program cannot be made independently. We are rarely able to optimize the choice of representation to minimize the given cost function. The payoffs from making good though perhaps sub-optimal choices are, however, dramatic and well worth the cost of our analyses.

For the remainder of the paper, the reader should assume that the cost function is a simple nondecreasing function of the primary memory space occupied by a program and the CPU time needed by the program. One such function (that which was used in the prototype system) is the integral of the primary memory as a function of time (space-time product). The reader should remember that actual cost functions are usually more complicated.

Representation Alternatives

Given any abstract data type, it is rarely the case that a single representation is optimal for all programs. In general, we have to do a thorough analysis to determine the tradeoffs of time and space as a function of the size of the data structure and the frequencies of the various primitives applied to it.

The first major question is whether to explicitly represent the abstract structure at all. For example, often it is better to represent mappings as programs. The trigonometric functions are usually represented by program segments rather than as explicit tables. Sets can often be thought of as mappings from elements to the Booleans TRUE or FALSE. If the mapping is a simple one, such as for the set of all odd integers, the set may best be represented by a predicate. Representation of an abstract data structure by a program segment, however, is not always optimal and depends upon frequencies of the various primitives on the abstract data structure. We must evaluate the costs of keeping the tables versus the costs of executing programs to evaluate the mappings. We must also consider the alternative of computing and storing only those values which the program uses. Thus we can imagine a system (cf. POP-2 memo functions [5]) which computes a value for a function only when it has not previously computed the function for the same arguments. The general technique involves first looking in the table. If the function has previously been computed for the particular argument, the value will be in the table and we may simply return that value. Otherwise we compute the value by executing the program segment, store the argument-value pair in the table, and then return the value.

The considerations mentioned above indicate that the representation problem for abstract types is extremely difficult independent of questions about how data might be stored. Finding general techniques for deciding when to store results and when to recompute them is a research area in itself though of course related to the problems of data structure selection.

For the remainder of this paper we assume that the abstract data structures will be explicitly stored. We consider systems which try to choose which storage structures and associated algorithms are best for a particular program.

Multiple Representations

Once we have decided to have explicit data structures as representations of abstract types, we must decide which explicit data structure to use. In addition to individual data structures, we must also consider the possibility of using more than one representation. There are several ways in which an abstract data structure may be represented beneficially by more than one concrete implementation. One technique is to represent the structure redundantly by storing the same data in two or more ways. The motivation for this strategy is that often there is no one representation which is most efficient or even applicable for all the primitive operations applied to the given abstract structure [17]. Consider a small finite set of integers. A bit map is not efficient for iterating through a nondense set. An efficient representation for iteration is a linked list, but this is not nearly as efficient for testing membership as the bit map. Therefore, for a particular program, it might be optimal to represent such a set both ways, by using the bit map for membership queries and the linked list for iteration. Of course we must consider the extra costs for storage and for inserting or removing elements from the set. However, if the operations of membership testing and iteration dominate the program's cost, then the redundant representation is justified.

In addition to total redundancy, there is also the possibility of partial redundancy. An example where partial redundancy is often used is in the mapping between virtual page numbers and physical page numbers in a paged memory system. In many such systems, there is a table for each user of the mappings between the user's logical pages and the physical pages in either primary or secondary memory. Part of that table is also kept in a high speed associative memory of relatively small capacity. On each address calculation, the memory mechanism first looks in the associative memory for the mapping information, and only if it is not there

does it look in the user's complete page map. Since the associative memory is at least an order of magnitude faster than primary memory, a net speedup for page mapping will occur. Similar uses of partial redundancy are seen in keeping directories of open disk files in primary memory, using buffer memories (caches), and so forth.

Two or more distinct implementations may beneficially be used to represent an abstract data structure when the frequency of the various primitives using that data structure changes greatly over time within the program. We often see that there are distinct phases within the lifetime of an abstract structure, for example, a dynamic construction phase when the structure is changing rapidly in size and a processing phase when the structure is relatively static. If we can recognize these phases, we can use different representations for each phase, with a translation of representation between the phases.

One more way in which multiple representations are valuable occurs when a user's abstract structure can be decomposed into two or more disjoint abstract data structures. For example, if the user program uses a ternary relation in which all makes, erases, and searches for instances of the relation have the first component specified as a constant, then we can act as if the ternary relation were a number of binary relations. Consider a ternary relation which has entries like (FATHEROF, JOHN, TOM), (SALARYOF, JOHN, 10000), (FATHEROF, ALICE, TOM), and (SALARYOF, TOM, 20000). If the first component is always specified, we can treat the relation as the two disjoint binary relations FATHEROF and SALARYOF and possibly use different representations for each binary relation.

Information Gathering: Library

In order to make the decision of which member of a library of implementations to use, we must have knowledge about how the abstract data structure is used within the program (size, primitives used, and their frequencies), and we must also have knowledge about the costs of the various implementations. Let us first consider the library of implementations.

The cost of using an implementation is a function of the resources which it uses. With our simple cost formula of the space-time product, the cost of a given implementation consists of terms involving the storage occupied by the data structure and manipulation algorithms and the processing time associated with manipulations on the data structure.

The space used by a representation is often easily computed. For example, if we are representing a set by a linked list of nodes, the space needed is merely the size of each node times the length of the list plus some small constant for header information. The length of the list would normally be the number of elements in the set. If, however, the manipulation algorithms may insert the same element of the set into the concrete structure more than once (it may be cheaper to always insert rather than check if the element is already there), then the computation of space needed is not so simple but would depend on the number of times each element is likely to be inserted in the set. Similar problems occur if we have schemes where we do not always recover space when the abstract structure contracts in size. For example, we often store information on disk with each item being placed on a particular track or in an overflow area if its track is filled. When a previously filled track has elements removed, we are very unlikely to search the overflow area to find elements which can be moved to their home track. Thus the amount and kind of storage needed by a representation is not always simple to determine. Fortunately we can often get simple functions of expected storage cost which give us good approximations for the storage costs.

Processor time requirements for the various manipulation routines are also not always straightforward to determine. Many routines cannot be mathematically analyzed to give us closed form cost functions. When the routines cannot be analyzed, we may monitor executions of them and statistically infer cost functions. In the prototype system described earlier, the various algorithms used could be mathematically analyzed by using the techniques of counting machine instructions executed described by Knuth [7]. Occasionally, a few simplifying approximations were used in order to obtain closed form cost functions.

In creating a selection system, we thus must compute formulas for the storage and processor costs using the individual representations. These functions should be placed into a library of functions which can be used in a selection phase. This implies that to add a new representation to the library we need only provide its manipulation routines and its cost functions. We do not need to alter the main structure of our selection system.

Information Gathering: User Program

When choosing a representation for an abstract data structure, it is crucial to know how the data structure is used within the user program (which primitives are used, how often, the size of the data structure, and so forth). Once we have determined such attributes, we can evaluate the cost functions to predict how costly a particular representation would be. We can then choose that representation which will tend to minimize the total cost of running the program.

There are three major techniques for determining such information. The first consists of statically analyzing the program. This can give us information about which primitives are used. However, the sizes of data structures and the frequency of execution of the various primitives will normally depend on input data which is not considered during the static analysis. This suggests

the second alternative of requiring the user to provide the information. This can be done either through interactive conversations between the representation selection program and the user or by requiring the user to make assertions or special declarations within the source program. The major difficulty with this approach is that often the user does not know the answers to the questions. As a program develops the attributes may change, and thus the user must be very careful to keep the information up to date. The third alternative is to monitor executions of the program, gathering statistics of the number of times primitives are executed, the size of data structures, and so forth. This has the disadvantage that the behavior of a program will often change dramatically as a function of the input data. The temptation of monitoring the program only once with a single input data set must be avoided. We also realize that the sample data sets used in program preparation and debugging are normally not typical of production runs. Therefore we should periodically include monitoring in production runs and investigate whether it is best to change the representation of the abstract structures. In making the decision to change the representation we must, of course, evaluate the cost of restructuring an existing database.

A combination of all three techniques will be necessary to get sufficient information to intelligently choose data representations.

Representation Selection

A selection program which has a library of cost functions for various representations and the parameters from the user's program for those cost functions can compute approximate expected costs for the program using particular assignments of representations to the individual abstract data structures of the program.

The selection process is essentially an assignment problem. In general we may have K different abstract data structures (DS(1), DS(2), DS(3), $\cdots$, DS(K)). For each data structure DS(I), we have $N(I)$ different choices of representation. As we stated earlier, depending on the cost function, the optimal assignment of representations might not be found by optimizing individual choices. In such a situation (as when we are using the space-time product as our criterion), to find the optimal assignment we must potentially consider all possible combinations of representations for the individual structures. If, for example, we have 20 different abstract data structures, each having 10 possible representations, a brute force technique would have to look at perhaps 20^{10} different assignments to determine which is the best. Clearly we cannot consider all these alternatives. We must apply structuring and heuristic techniques which will cut down the range of possibilities. We realize that by employing these techniques we may pick a suboptimal assignment, but we hope that our heuristics will give us a reasonably good one. The particular heuristics used will depend on the cost function.

4. Conclusion

The use of abstract data structures contributes to the construction of better programs. In the past, programmers have not used them because they were not part of the programming language. Recent programming languages have provided abstract data structures, but they normally have only a single general-purpose implementation for each abstract type, which cannot be efficient for all purposes. In this paper we have proposed a system (and demonstrated its feasibility in the prototype implementation described) which is a tool for producing efficient programs using abstract data types.

The techniques presented here are not only valuable for selecting data representations but also would be highly useful in selecting algorithms. Problem domain specific systems would be very useful in the fields of numerical analysis, statistics and many other areas.

Appendix. SAIL Notation

The notation used in the example (Figure 1) is a modified version of the SAIL syntax. SAIL has notation which is highly similar to Algol-60. The differences (which appear in the example) are:

1. Block names—Following any **begin** the programmer may insert a string constant. The programmer may also insert a string constant after any **end**. If there is a string constant following an **end**, the compiler checks to make sure that is the same constant as followed the opening **begin**.
2. Itemvars—An itemvar is simply a variable which holds items. It is often preceded by a data type. A **list itemvar** is thus a variable which will hold items whose datums are lists. An **integer itemvar** would be a variable which holds items whose datums are integers. Just as an **integer procedure** returns values (whole numbers) which can be stored in an **integer** variable, a **list itemvar procedure** returns values (items with **list** datums) which may be stored in a **list itemvar** variable.
3. Datum—To reference the datum connected with an item we use the pseudo-function **datum**. The **datum** construct looks at the type of item (or itemvar) to decide which data type the datum is. Thus **datum**(CLASS1) where CLASS1 is a list itemvar will behave just as a list variable. When **datum**(CLASS1) appears on the left side of an assignment statement, the datum of the item is replaced. Otherwise the datum of the item is fetched.
4. List selectors—Lists may be indexed just as if they were a one dimensional array of items with the lower bound of the dimension equal to 1. If we had a list variable LVAR, LVAR[2] corresponds to the second item in the list.

   LVAR[2] := itemexpression;

   will remove the second element of the list and replace it with the value of the itemexpression. In the example we were using a list to represent mappings. These lists were datums of items. Thus the construct

   datum(CLASSVAR)[2]

   means to take the second element of the list which is the datum

of the item stored in the itemvar CLASSVAR.

5. Foreach statements—Foreach statements are similar to FOR-loops which use **itemvar** instead of **integer** loop-control variables.

   foreach *X* **such that** *X* **in** SETEXPR **do**

   will mean to execute the statement following the **do** once for each element in the set with the itemvar "*X*," receiving in turn each element of the set. Since sets are conceptually unordered, the implementation does not define the order in which the elements of the set will be placed in "*X*."

6. PUT (REMOVE). These statements insert (remove) an item into (from) a set variable.

Received August 1976; revised May 1977

References

1. Allen, F.E., and Cocke, J. A program data flow analysis procedure. *Comm. ACM 19*, 3 (March 1976), 137–146.
2. Allen, F.E. Bibliography on program optimization. Rep. RC5767, IBM T.J. Watson Res. Ctr., Dec. 1975.
3. Birtwistle, G. et al. DECSYSTEM-10 SIMULA Language Handbook. Rep. C8398, C8399, Swedish Nat. Defense Res. Inst., Stockholm, Sweden, 1974.
4. Bobrow, D.G., and Raphael, B. New programming languages for artificial intelligence research. *Computing Surveys 6*, 3 (Sept. 1974), 155–174.
5. Burstall, R.M., Collins, J.S., and Popplestone, R.J. *Programming in POP.2*. Edinburgh U. Press, Edinburgh, Scotland, 1971.
6. ECL Programmer's Manual. Ctr. Res. Comptng. Tech., Harvard U., Cambridge, Mass., Dec. 1974.
7. Feldman, J., and Rovner, P. An Algol-based associative language. *Comm. ACM 12*, 8 (Aug. 1969), 439–449.
8. Feldman, J., et al. Recent developments in SAIL—an ALGOL-based language for artificial intelligence. Proc. AFIPS 1972 FJCC, AFIPS Press, Montvale, N.J., pp. 1193–1202.
9. Geschke, C.M., and Mitchell, J.G. On the problem of uniform references to data structures. *IEEE Trans. Software Eng. SE-1* (June 1975), 207–219.
10. Kildall, G. Global expression optimization during compilation. TR-72-06-02, Ph.D. Diss., Dept. Compt. Sci., U. of Washington, Seattle, Wash., June 1972.
11. Knuth, D.E. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms*. Addison-Wesley, Reading, Mass., 1968.
12. Knuth, D.E. Structured programming with goto statements. *Computing Surveys 6*, 4, (Dec. 1974), 261–301.
13. Liskov, B., and Zilles, S. Programming with abstract data types. Proc. Symp. on Very High Level Languages. Sigplan Notices (ACM) *9*, 4 (April 1974), 50–59.
14. Low, J. *Automatic Coding: Choice of Data Structures*. Interdisciplinary Syst. Res. Rep. *16*, Birkhaüser Verlag, Basel, 1976.
15. Low, J., and Rovner, P., Techniques for the automatic selection of data structures. Conf. Rec. Third ACM Symp. on Principles of Programming Languages, Atlanta, Ga., Jan. 1976, 58–67.
16. Reiser, J. SAIL USER MANUAL. Tech. Rep. STAN-CS-76-574, Compt. Sci. Dept., Stanford U., Stanford, Calif., Aug. 1976.
17. Rovner, P. Automatic representation selection for associative data structures. Ph.D. Th., Harvard U., Cambridge, Mass; Tech. Rep. TR10, U. of Rochester, Rochester, N.Y., Sept. 1976.
18. Schwartz, J. Optimization of very high level languages—I: Value transmission and its corollaries. In *Computer Languages, Vol. I*, Pergamon Press, Elmsford, N.Y., 1975, 161–194.
19. Tennenbaum, A. Type determination for very high level languages. Ph.D. Th., Courant Inst. Math. Sci., New York U., New York, Oct. 1974.
20. Wegbreit, B., and Spitzen, J.M. Proving properties of complex data structures. *J. ACM 23*, 2 (April 1976), 389–396.
21. Wulf, W.A., London, R.L., and Shaw, M. An introduction to the construction and verification of ALPHARD programs. *IEEE Trans. Software Eng. SE-2*, 4, (Dec. 1976), 253–265.

Programming Languages

J. J. Horning
Editor

Incorporation of Units into Programming Languages

Michael Karr and David B. Loveman III
Massachusetts Computer Associates

The issues of how a programming language might aid in keeping track of physical units (feet, sec, etc.) are discussed. A method is given for the introduction of relationships among units (a watt is volts*amps, a yard is three feet) and subsequent automatic conversion based upon these relationships. Various proposals for syntax are considered.

Key Words and Phrases: units, language design, compiler construction, language syntax
CR Categories: 4.12, 4.22

This work was supported by the U.S. Army, Frankford Arsenal, under Service Contract No. DAAA-74-c0530 and Battelle/Army Research Office Scientific Service Program.

Authors' address: Massachusetts Computer Associates, Inc., Wakesfield, MA 01880.