AN ADVISORY SYSTEM FOR DEVELOPING DATA REPRESENTATIONS

Shmuel Katz

IBM Israel Scientific Outer
Technion City, Haifa, Israel

Ruth Zimmerman

Computer Science Department
Technion - Israel Institute of Technology
Haifa, Israel

Abstract

A knowledge-based interactive system for choosing abstract data structures and concrete represent.* tions for them is described. The system acts as an "expert" on data structures and is to be consulted by programmers during the design stage of their work A specification method is described which assumes a restricted vocabulary of data structure terminology common to the user and the system. The selection process uses a base of knowledge about known abstract data structures and representations, and provides new combinations of known structures which satisfy complex requirements not anticipated in advance

Key words and phrases: data structure representations, expert system, choice of data structures.

C.R. categories, 3.64, 4.34.

1. Introduction

This paper describes a knowledge-based inter active system which chooses abstract data structures and concrete representations for them. Both the principles behind the system, and features of the presently implemented version are presented.

The system acts as an expert on data structures and representations, giving advice on data structures to a programmer using a high-level "popular" language- such as PL/1, Fortran or Pascal. The system should be consulted during the development of the user's program, before the actual code is written. The user directly describes his data requirements, mainly in terms of the operations he wishes to perform, and is interrogated to fill in information which is crucial to the selection. We have employed the techniques of expert systems on a problem taken from the program ming world itself, and we try to emulate the process followed by a human expert in data structures, producing combinations of known structures (and representations).

The main justification for a selection system is that finding good representations has proven to be quite difficult: many programs in wide spread use suffer from unwise realizations of data structures. This is due to a lack of familiarity with the more recently developed data structures and to a lack of awareness of the importance of an appropriate choice for data structures.

The use *of* very high level (VHL) languages such as SETL, SAIL or ESI. might be expected to alleviate the problem of bad selection by leaving such matters to an optimizer as seen in [L74], [SSS81] or [T79]. There are two reasons why this view is not fully satisfactory. First, the general programming community simply does not use these languages and has no intention of doing so in the near future. Secondly, in VHl languages the programmer may mistakenly overcommit himself to an inefficient abstract structure. This is because the user must choose among a fixed set of abstract data structures, while the optimization is done after this choice has been made and cannot affect it.

Synthesizing a program which efficiently implements a data abstraction using general programming knowledge and methodological heuristics has made some progress [B79], but is difficult to treat in the wider context of program synthesis. Our system attacks this specific problem in depth by giving detailed advice based on the same knowledge available to a human expert in data structures, e.g., from a standard text in data structures. This detailed advice may be expanded in the future to create automatically programmed procedures for the user's operations and specific declarations of the representations in the needed programming language.

The system includes a base of knowledge about data structures as well as algorithms that should work on the base. The base is a given set of abstract data structures (e.g. set, sequence, graph, etc.) each with one or more representations. The elements of the base structures (e.g. members of the set, nodes of the tree, etc.) are regarded as atomic elements for the purposes of the representations in the base. We assume that the programmer has in mind operations on a data structure whose properties he understands but the collection of operations he requires does not necessarily define a single abstract structure in the base, and does not necessarily have one word elements. The system searches for an efficient representation by viewing the programmer's data structure as a combination of base structures.

The "wisdom" of the system (as opposed to its 'rote knowledge') lies in its ability to reach a combination containing some of the structures of the base, thus enabling realization of non-standard, not

previously identified abstract data structures.

Certain assumptions have been made in order to clarify and somewhat simplify the types of problems to be presented to the advisory system.
(a) Internal storage. The system should be consulted in case of problems requiring internal memory, (users with problems which require extensive external storage should better 'consult' a data base expert).
(b) Distinct structures. The user should come with a set of conceptually distinct structures, consulting the system separately for each distinct structure. If the user's operations can be divided into two groups, A and B, such that each operation in A has no effect on the result of an operation in B, and vice versa, A and B are distinct structures.
(c) Strong interaction. A structure is viewed as a collection of elements upon which some operations may occur, and a strong interaction is assumed among these operations. Strong interaction means that all subsequent operations take into account the changes in the structure made by all the previous operations.

In fact various degrees of interaction could be defined among the operations. Total non-interaction, for example, means that no operation affects any subsequent operations, and is clearly useless for defining a dynamically changing data structure. Semi-interaction means that some of the subsequent operations ignore changes made by some of the previous operations. This requires a detailed specification about the degrees of influence among the operations. Again, we have chosen to work with the most common situation, strongly interacting operations.
(d) Efficiency. Within the limits of the contents of the base given to the system we claim that the system reaches the best possible combination of representations for a given specification (when expressed in terms familiar to the system) under two criteria:

The time criterion: The representations chosen give "efficient time complexity" for each of the operations the user has defined. For each operation, a desired level of efficiency has been fixed. Thus operations involving a single element of a structure (e.g. DELETE, INSERT, etc.) should not take more than 0(log n) time, and if possible 0(1) time is preferred by the system. Simple operations involving all the elements in the structure, such as ITERATE, are required to take 0(n). For a SORT operation its representations would have to be in 0(n log n). A SIZE operation, on the other hand, must be performed in 0(1). The system will generally consider worst-case time efficiency, but asks questions which allow using the average case when appropriate.

The reduction criterion; Under the above time constraints the system suggests a representation containing the fewest possible distinct representations of structures in the base. Thus, in general, time is preferred to space.

In Section 2 a "natural" specification method, used both as a communication tool with the user and as a representation of part of the system knowledge, is presented. Section 3 describes the base of data structure knowledge, while Section 4 presents the selection process. Two examples are given in Section 5 and conclusions are in Section 6. Some of the information in the present base may be found in the appendix.

The advice given, i.e., the output from the system,includes naming the representation structures chosen, with the interconnections among them. The structure in which each user operation should be done is identified, and the way to update the other representations is briefly described. A trace of the considerations which led to the choice is optional. In short, the system provides advice on the same level as a human expert in data structures advising a colleague.

Some of the ideas behind the system described in this paper were first presented in [RK77]. In addition to the works already mentioned, Kant [K77] described an optimizer which was part of a large program synthesizing system. Rowe and Tonge [RT78] present a specification language and algorithms for selecting low level implementation structures, but did not implement a system.

2. Expressing Requirements - The Specification Method

In order to build an interactive advisory system a medium must be developed in which the user can express his requirements to the system. (Systems which are oriented toward program analysis, on the other hand, ([L74], [SSS81]) ordinarily do not need such a medium.)

Unfortunately, the standard formalisms, such as algebraic specification [ADJ] or state transformations, are often unintuitive and difficult to reason about. Any specification method involves expressing the unknown, the object to be specified, in terms of something known. This is done using a known metaformalism. Programmers in fact communicate (orally) among themselves about data structures using a common linguistic base of known vocabulary about data structures.

On the level of expressing abstract requirements, this vocabulary consists of generic operations and additional attributes which together are sufficient to express what is to be done with the structure and how to access its elements. With the above motivation, we will now define how we use these terms. A linguistic base is any collection of generic operations, criteria and specific attributes (all explained below) which is assumed known for the purposes of specification. A sample linguistic base is shown in Figure 1.

A generic operation is the common denominator of meaning among a recognized family of operations on data structures, and is assumed understood by both the user and the system, based on the meaning of the words in English. For example, one generic operation is INSERT. This means that an item is to be adjoined to a structure in such a way that subsequent operations will react as if the item is in the structure. Similarly, a generic DELETE operation will make some element no longer available in the structure. The precise manner in which the user's needed operation, say X, acts on the structure is not specified merely by identifying X as a generic delete, and will depend on the attributes required. Only with this information

Generic Operations

insert	size
delete	iterate
fetch	exist
replace	

Criteria and Attributes

element		undetermined:	any
contents:	value	connection:	(between elements)
	min		undirected
	max		directed
	n-th largest		directed-from
	n-th smallest		directed-to
place:	first	order:	increasing
	last		decreasing
	n-th index	evaluating	
	n-th place	function:	i
time:	new	quantity:	one
	old		n (number)
			all

Figure 1 - A sample linguistic base

will it be possible to determine whether X is, e.g., a "pop" from a stack or "remove the 7th element" from a sequence.

The criteria are additional types of properties which allow distinguishing the possible instantiations of a generic operation. They define the access method to be applied when performing the operation. Each criterion may have an arbitrary set of specific attributes, which are in effect the possible "values" which the criterion may assume. While the underlying meaning of the generic operations is self-evident, no standard terminology exists for the criteria and attributes, even though the concepts are widespread. Thus an informal explanation of their intended meanings is given by the system.

The main descriptors in the present base are the criteria of element contents, time, place, order, connections between elements, or undetermined (a "don't-care" option). Additional modifiers, used with 'element contents', are the quantity, or the evaluating functions to be used. As an example of their use, a 'find by element contents' means an operation which is to be given a value, or a function based on the value such as min, and locates an element whose contents has that value or fulfills the appropriate function. On the other hand, 'find by place' is given an index or pointer as argument, while 'find by time' would need to locate, say, the newest or oldest element in the structure.

When the elements involved are not atomic, an evaluating function is needed to define the element's 'value'. For the purpose of the system, it is sufficient to indicate which operations have the same evaluating function, using arbitrary identifying' numbers for the functions.

liven the trivial basis given in Figure 1 is sufficient to describe many data structures which are difficult to understand using other methods. Consider an abstract data structure with four operations. A summary of a specification using the method we suggest might be:

generic operation	*-attribute*	(using criterion)
insert		
delete	min-one	(using element contents, quantity)
fetch	old	(using time)
size		

In words, we have defined a structure into which we would like to insert elements so that we can always determine how large the structure is (by applying the size operation), fetch (in order to print or examine) the element which is the structure the longest time (applying the fetch-old operation), and delete one copy of the element(s) whose value is minimal in the structure (applying the delete - min-one operation).

The same specification method is used to describe the abstract structure 'library' of the system base.

As mentioned in the Introduction, the output will also be expressed in terms of a common vocabulary between the system and the user. For example, the advice includes naming chosen representations such as "a 2-3 tree" or "a bit vector", assuming these known to the user. Some examples of output are given in Section 5.

3. The Base

One of the main guidelines for any Expert System should be the separation between the knowledge the system has and the algorithms that work on that knowledge. This concept enables easy additions or changes in the knowledge without many (or any) changes in the algorithms.

The base of the system is the extendable part of its knowledge. it includes information about abstract data structures (e.g., set, graph, sequence, counter, etc.), each of which has a number of representations associated with it. An abstract structure is a collection of operations for which there exists at least one representation with efficient time complexity for each operation. The organization of the base is shown schematically in Figure 2, and is explained in more detail below.

The abstract data structure part of the base includes definitions of abstract structures 'known' to the system according to our specification method (described in the previous section), and information about their relative simplicity.

During our study it became evident that for most abstract structures the representation choice is completely independent of the choice for the other abstract structures in the required combination of structures. However, in some specific situations, in order to reach the best possible representation, the choice should consider the properties of the representations already chosen. For example, we have identified a subgroup of structures which have one 'designated' operation (providing information about the structure), for which they are useful, and which justifies combining the structure with other abstract structures not immediately permitting this operation.

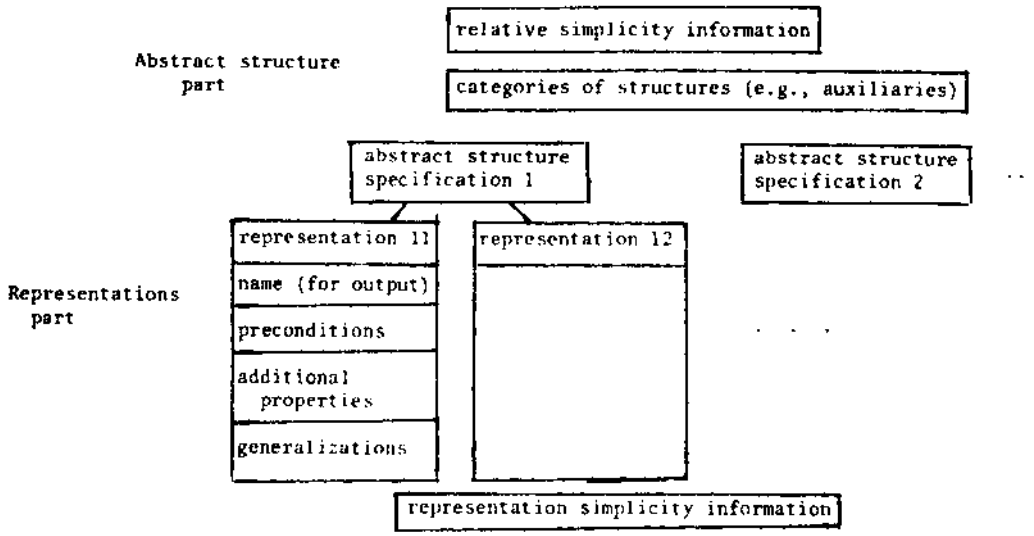

Figure 2 - Organization of the base

These structures are termed auxiliaries. Upon further analysis, they often become extraneous and can be eliminated whenever the designated operation of the auxiliary can be performed efficiently by the representations of the other structures.

Examples of such structures are the collection which allows iterating through its elements, or the counter, which allows determining the size of a structure efficiently. A representation for an auxiliary will be chosen only after it becomes clear that it is not extraneous.

The user's needs may sometimes be satisfied in more than one way. The information about relative simplicity helps the system to choose the simplest structures that fulfill the user's needs. The simplicity level is actually a hint on the complexity of the representations for a structure, and is measured according to the most complicated representations a structure has. Thus a stack is considered simpler than a set, by comparing the linked list with the 2-3 tree (the stack's and the set's most complex representations, respectively).

The representations part of the base includes a list of acceptable representations (i.e. those with all operations efficient according to the time criterion), for each abstract structure and additional simplicity information. Each of the representations has associated with it 'preconditions' for its applicability, generalizations, and additional properties.

The preconditions are properties of the elements (expected size, range of values), of the operations (relative frequencies), or of the high-level structure product. For example the binary tree representation of a set requires some ordering relation among the elements. Additional properties which are useful in the selection process may be found in the Appendix.

The generalizations are alternative representations which implement the same operations as the included representation (within the limits of efficient time complexity), but may permit additional operations to be done efficiently and may be more space consuming. For example a threaded tree is a generalized representation of a binary tree: it enables an easier inorder pass through the tree but requires more space. The generalization list is used in the reduction stage of the selection (see Sec. 4).

Additional information about the simplicity of the representations is needed because some operation - attribute combinations can be executed in several representations within the required orders of magnitude of time. Of course, the 'simplest' representation is desired and when the other properties of the representations are not sufficient for choosing, we add special information which to the system is arbitrary, but which the designer of the base can provide, according to his intuitive understanding, or by considering the number of primitive operations more precisely.

4. The Choice Process

After absorbing the description of the user's abstract data structure the system must choose a realization. This choice process can be divided into stages: (a) The operations are grouped into those which refer to the same abstract user structure, (b) for each user structure, a collection of known abstract structures from the base are chosen to 'cover' the needed operations, (c) tentative representations are chosen for each structure in the base, (d) an attempt is made to generalize tentative representations in order to reduce the number of representation structures, (e) representations based on links and pointers are combined into a single 'plex' (for reasons explained below), and (f) various slight generalizations and interconnections among representations are added in order to ensure adequate time complexity for all operations on all structures.

The stages are briefly described below. The

examples in the next section illustrate the difficulties and their solutions.

(a) Identifying the user'e structure: The description given according to the specification method described in Section 2 includes naming the entities which are in the user's data structure requirements. When defining each of the operations he wants, the user is also required to indicate which entity serves as the aggregate and which as the item (element), for that operation. A level is simply a collection of operations with the same aggregate. When the name associated with an aggregate entity for some operations is used in other operations of the user's data structure as an element, the multi-level situation exists. It is clearly not difficult to identify the levels by partitioning the operations appropriately.

Recall that the base structures made no assumptions about the nature of 'elements'. In the multi-level situation an aggregate on an upper level will have elements which are entire structures. Thus the system must decide, using the preconditions in the base, which of the representations are still applicable, for such an aggregate. The representations ultimately chosen will generally contain pointers or vectors of indices to the 'elements'.

Note that the levels need not form a simple hierarchy, and that a circular situation is possible (e.g., the elements of a structure A are themselves structures of type B, whose elements are of type A).

At this stage the system also attempts to eliminate operations whose access method was left unspecified (using the attribute any), when they can be subsumed in other, more specific operations already specified.

When evaluating functions are involved, the operations of each level are further divided into classes. Each class is a collection of operations which use the same evaluating function, or do not require such a function. When the 'designated' operations of the auxiliary structures are present, they form a separate class.

(b) Choosing a cover: Crucial to the process of choosing within a single class is the idea of finding a set of abstract structures whose union of operations is a superset of the operations required for that class. Such a set is called a cover of the user's class. When the cover includes more than one abstract structure, it is called a structure product. Whenever a level includes several classes, the union of the covers of the classes forms the level cover.

The existence of a cover guarantees that each operation required by the user can be efficiently implemented in a representation for one of the structures in the cover.

The choice of covers is done independently for each level, keeping the number of structures needed to a minimum, and choosing structures which are 'simplest' according to the guidelines found in the base.

(o) Tentative representation ohoioe: The best representation according to the time and the space criteria is chosen for each abstract structure in the cover of the level. This choice is done class by class. In trying to keep the principle of an expandable base system the system (except for the auxiliary case) chooses the representation of each abstract structure regardless of the previous representations already chosen, but will take into account the user's global needs as given in his specification. This allows the choice to be done in any order among the classes and among the abstract structures in them.

(d) Generalizations: A reduction in the number of representations in the structure product is extremely beneficial both to space and time efficiency, because then, less updating may be needed to keep all the representations containing the same information. Thus once representations have been chosen for each abstract structure in the structure product of the class, an attempt is made to 'generalize' the representations (using "generalization lists") in order to find a 'common ancestor' which could provide all of the needed operations at reasonable cost. For example, consider a structure product which includes a set and a priority queue, for which a binary tree and a linked heap have been chosen, respectively. This stage will generalize the linked heap into a binary tree and thus the final representation choice will be reduced to a binary tree (perhaps with an additional pointer to the extremum value).

The reduction is only possible among representations which use the same evaluating function and is thus done within each class, but not among the classes.

The strong interaction assumption discussed in the Introduction implies that operations which change the user's abstract structure require updating in each of the structure product representations. The updating should be done within the efficient time defined for the operation. The choice of a cover guarantees that these operations are performed 'naturally' (along the access paths) in one representation - the natural representation, but are not natural for the others. The updating means that the same generic operation should be performed on each of the other representations with the element identified in the natural representation, and leads to two problems:

(1) A direct access (usually not the natural one along the usual access paths) should be provided in order to reach the element identified in the original operation,

(2) Using an unnatural access requires additional connections (pointers) among the elements of the representation in order to actually perform the generic operation.

The need for such a direct access can be avoided either if we can use the regular access paths to the other structure (e.g., elements are accessed efficiently by their values) or if we can do

(e) Combination to a 'plex'. When some representations share the 'linked' property a "node merging combination" may be built. The corresponding nodes of the structure in the product are merged, constructing one plex where each node includes one copy of the information (or a pointer to the information, when the information is not atomic) and pointers (access paths) according to each of the original representations. Thus the same node can be reached using

either access path. No direct access is needed in the plex for any operation, but some operations still create the second (processing) problem.

if) When there are structures not allowing efficient access by value to elements which must be updated, direct accesses using pointers or indices to these elements are added to the structure where the access is natural. Additional links may also be added to allow needed rearrangements caused by the external change.

5. Examples

The following exchange involves a problem from a car rental agency, which will not be described in detail, but which the user understands and for which he has developed some data structure requirements. The user writes in capitals, and responses of the system are preceeded by a *. Our comments are in parentheses.

* Define your entities please, when you finish, hit 'enter' twice
  BCAR (a structure of the available cars)
  CAR
  KM (the number of kilometers so far)
  NO (the number of the car)
  (that is all, for now)
* Please define your operation_attribute pairs, (The exact form of the user-system communication protocol is more detailed than given below, it includes optional prompting and a detailed inquiry about which is the aggregate and which is the element in each operation).
  FETCH_MAX (BCAR, CAR) (to find the available car with the most kilometers).
  DELETE_MAX (BCAR, CAR) (the most used car is no longer available).
  INSERT (BCAR, CAR) (a car has become available).
  DELETE_MIN (BCAR, CAR) (least used is rented).
  DELETE ANY (BCAR, CAR) (in case we do not care which car is rented).
  SIZE (BCAR) (number of available cars).
  INSERT_INDEX (CAR, NO) (to insert the car number into the 'car file').
  INSERT_INDEX (CAR, NAME) (to insert the brand of car).
* Is NAME a new entity?
  YES
  REPLACE_INDEX (CAR, KM) (to update the car kilometrage).
  REPLACE_INDEX (CAR, DATE) (to update the date of last maintenance).
* Is DATE a new entity?
  YES

(The system analyzes two levels to be chosen independently)

* Choice for (BCAR, CAR) level:
* The delete-any will be subsumed by the delete-max or the delete-min.
* An evaluating function must be used in the implementation for the operations: delete-max, delete-min, fetch-min. If you use the same evaluating function for each operation answer yes.
  YES
* From here on, whenever the term 'value' is mentioned, refer to value calculated by your function .
* Can the elements appear more than once in the structure?
  NO
* The abstract choice for the (BCAR, CAR) level is max priority queue, min priority queue, counter.
* Do the elements come in random order (not ordered by their value)?
  YES
* The representation choice is: (the two priority queues are recombined into one structure - the binary tree)
  binary tree
  cell
* The nodes of the binary tree will include pointers to the element (CAR) and will be ordered according to the evaluating function.
  Use a pointer to the maximal element in the tree.
* Execution of the operations:
  INSERT, DELETE_MAX, DELETE_MIN - in the binary tree and update the cell.
  FETCH_MAX - in the binary tree.
  SIZE - in the cell.
* Choice for (CAR, (NAME, NO, KM, DATE)) level:
* Can the elements appear more than once in the structure?
  YES
* The abstract choice for (CAR, (NAME, NO, KM, DATE)) is a VECTOR.
* Can the maximal number of different elements in the structure be estimated?
  YES
* Are the values of the indices bounded and contiguous?
  YES
* Does each index always point to an element having a fixed meaning?
  YES
* The representation choice is a record.
* All the operations - in the record.

(Note that if the user needs different 'value functions' for the delete-max and delete-min then their representations cannot be recombined into one representation, and the abstract choice will be represented by 3 different representations with all the problems arising from that. The counter then may be eliminated if one of the priority queues is represented by a 'heap in an array', since an index to the last element is used in the heap representation, and its value is the needed size.)

Example 2. Here we consider a single level, with aggregate SI and elements S2. The requirement description is shortened for space reasons, and is

insert-index (S1,S2)	(insert and fetch values according to their fixed place in the aggregate).
fetch-index (S1,S2)	
delete-min (S1.S2)	(delete element with smallest value)
size (SI)	(find number of elements in SI)

After absorbing the description the system responds.

* The abstract choice for the (S1,S2) level is vector, min-priority queue, counter.
* Are the values of the indices bounded and contiguous?
  NO (record or simple array are eliminated for the abstract vector).

* Are successive insertions random, and not by successive index or place numbers?
  YES (leads to tentative binary tree ordered by index for abstract vector)
* Can a value of an element appear more than once?
  NO (eliminating multi-valued representations for priority queue).
* Can maximal number of different values be given?
  YES (leads to tentative heap in an array making counter extraneous)
* The representation choice is
  binary tree ordered by index
  heap in an array with index of last element of heap
* Required interconnections are
  each node of binary tree contains also pointer to its father and index of that value in the heap in an array
  each node of heap contains also pointer to node with that value in the binary tree.
* Execution of the operations
  insert-index - insert in binary tree by index, delete old value from heap using index to location, insert in heap by value
  fetch-index - in binary tree by index
  delete-min - in heap and using pointer to binary tree index to delete there
  size - in largest index value of heap in array.

(Note that a negative answer to the last question would lead to a linked heap to implement the priority queue. In that case we could have formed a plex. Each of its nodes would contain one copy of the information (the element S2), two pointers to the sons and one to the father according to the heap access paths, and two pointers to the sons and one to the father according to the access paths of the binary tree ordered by index a).

6. Conclusions

The system has been implemented in APL on an IBM 370/168. Although it is a practical tool which can solve many realistic problems, the present version is nevertheless a prototype. The most natural extensions, of course, would simply involve adding additional abstract structures, operations, attributes, or new representations to the base of the system without affecting the basic algorithms of the system.

The user of this advisory system must have a clear idea of his requirements, hopefully obtained at a fairly early stage in a top-down development of his program. The transformation to the notation we have suggested can still sometimes be difficult, depending on what is known to the system (i.e., the nature of the base), and further work on the communication medium would be beneficial.

The strongest claims we can make are of internal completeness between the specification and the base:
- Every structure combination which can be suggested by the system is describable in our specification method, and
- Any specification containing only generic operation attribute combinations permitted in the base, can be realized by the system, i.e., a combination of representations of base structures can be found which allows efficiently performing all the operations of the specification.

References

[ADJ] J.A.Goguen, J.W.Thatcher, E.G.Wagner: "An Initial Algebra Approach..." in Current Trends Programming Methodology (ed.R.Yeh), 1978.

[B79] D.R. Barstow: "An Experiment in Knowledge-Based Automatic Programming", Artificial Intelligence, Vol.12, 1979, pp. 73-119.

[K77] E. Kant: "The Selection of Efficient Implementations for a High Level Language", SIGPLAN Not.,12, 8, Aug. 1977, pp. 140-146.

[L74] J.R. Low: "Automatic Coding: Choice of Data Structures", Ph.D. Thesis, Stanford Univ., Computer Science Memo, STAN-CS-74-452, 1974.

[RK77] S.J. Rosenshein, S.M. Katz: "Selection of Representations for Data Structures, SIGPLAN Not., 12, 8, August 1977, pp. 147-154.

[RT78] L.A. Rowe and F.M.Tonge: "Automating the Selection of Implementation Structures", IEEE Transactions on Software Engineering, SE-4, No.6, November 1978, pp. 494-506.

[SSS81] E. Schonberg, J.T. Schwartz, M. Sharir: "An Automatic Technique of Selection of Data Representations in SETL Programs". ACM - TOPLAS, 3, 2, April 1981, pp. 126-143.

[T79] A. Teperman: "Optimization of Very High Level Language", Ph.D. Thesis, Technion, Haifa, 1979.

Appendix

a. Partial List of Representations

BIT VECTOR	CELL
HASH TABLE	RECORD
SPARSE TABLE	ADJACENCY MATRIX
BINARY TREE	MULTILIST
2-3 TREE	ADJACENCY LIST
THREADED TREE	INVERSE ADJ. LIST
ARRAY ORDERED BY TIME	DOUBLY LINKED LIST
UNORDERED ARRAY SIZE 20	
2-3 TREE DOUBLY LINKED	
LINKED LIST ORDERED FROM NEW TO OLD	
HEAP (ARRAY) MAXIMUM IN ROOT	
LINKED HEAP MAXIMUM IN ROOT	
BINARY TREE ORDERED BY PLACE	
BINARY TREE ORDERED BY INDEX	

b. Abstract Structure List

SET (BAG)	SEQUENCE
ORDERED_SET (BAG)	VECTOR
STACK	GRAPH
QUEUE	DIRECTED-GRAPH
MAX_PRIORITY QUEUE	COUNTER
MIN_PRIORITY QUEUE	COLLECTION

c. Properties of the Representations:

(1) A 'linked' representation.
(2) 'Tree like' representation.
(3) Representation with access by value.
(4) Array with contiguous information.
(5) Rearrangement after deletion is needed.
(6) Number of elements differs from representation size.
(7) Represents connections among elements.