Research Directions in Abstract Data Structures

Mary Shaw
Computer Science Department
Carnegie-Mellon University
Pittsburgh, Pa. 15213

INTRODUCTION

A number of interesting research problems arise from current attempts to incorporate abstraction mechanisms in programming languages. Some of them are central issues in current research projects and others are direct extensions of current work. Several problems in each of these areas are outlined below. A third section presents a series of speculations about how this work might evolve in the future.

Since this is supposed to be a session to spark research ideas, I have written brief descriptions of a number of topics and not delved very deeply into any one. Most of these problems can probably be solved in more than one way. Hence the fact that a problem is being worked on should not deter anyone from looking at it from another viewpoint.

Many of the problems I am excited about and feel are feasible in the (relatively) short term are clustered around the Alphard language research. Alphard is a member of the new generation of programming languages whose central concern is the construction of "well-structured" programs and which are exploring the notions of abstraction and data types to that end. In order to be concrete, the descriptions below refer to Alphard, but no prior knowledge of Alphard should be necessary to appreciate the research problems being described.

CURRENT ACTIVE TOPICS IN ALPHARD

The main thrust of the Alphard research addresses four problems related to the abstract definition of data structures:

- adding new abstract data structures to the base language,
- verifying the intended properties of these structures,
- adapting the basic language semantics to the added structures, and
- providing the programmer with control over how each data structure may be used.

Extension of Language to Incorporate Abstract Data Structures

In Alphard, the form is introduced as a syntactic device to encapsulate the information related to the abstract definition of a data type. Each form contains a specification, which describes the behavior of the data structure, and a separate implementation, where the declarations and procedures which implement that behavior are written. This encapsulation serves two purposes: it makes it clear which definitions are associated with the data type, and it provides a means for keeping some of the information about a type private to the definition of the type. This work is in many ways an extension of the Simula *class* concept [Dahl72]; similar ideas are also being explored in CLU [Liskov74]. Some reasons for localizing information in a type definition are discussed in [Parnas71] and [Wulf73].

Verification Relative to Data Abstractions

Two of the major outstanding needs of program verification are ways to segment verifications of large programs into chunks of manageable size and to reuse proofs instead of recreating them. Alphard addresses these problems by structuring the verification the same way the program is organized. The verification of a form consists of showing that the implementation faithfully preserves the behavior described in the specification. Programs (other forms) which use a verified form may then treat the abstract description of the latter behavior as primitive in their own proofs. This is expected to reduce the difficulty of verification by segmenting the labor and permitting reuse of groups of theorems.

Specialization of Control Constructs to Defined Types

Many properties of an abstract data type can be localized by defining functions which share data. (In Alphard, scope restrictions to do this are provided by the form.) Conventional languages make it hard to avoid distributing some information about the internal organization of a data type throughout the program. One of the most common ways such information is strewn around a program is by using control constructs "natural" to the implementation, rather than the specification, of the abstract type.

For example, the choice of a looping construct is often intimately tied to the implementation of a data structure. Thus, if S is a sequence, you often write

for i := 1 step 1 until n do <statement involving S[i]>

which strongly suggests S is implemented as a vector, or

while S := S.next ≠ null do <statement involving S.value>

which makes it clear that a linked list is involved. Alphard addresses this problem by providing control constructs whose basic semantics is fixed, but whose details can be provided in a form with the rest of the information about the implementation of an abstraction. As a result, you may write

for x in sequence do <statement involving x>

which does not suggest a particular implementation of sequences. This will be particularly helpful if the implementation is ever changed; fewer (ideally zero) uses of the structure will have to be located and modified.

Protection Applied to Data Types

One of the objectives of block structure is to restrict the the number of sections of a program that have access to any given variable. Block structure is inadequate because it can implement only purely hierarchical organizations. The form provides a way for a set of procedures to share data which is hidden (and therefore protected) from other users. However, even this is not enough. We would like to be able to control not only whether or not an object is accessible, but also what kinds of access (or operations) can be performed at various points. Alphard is adapting protection techniques from operating systems to address this problem [Jones74].

Coping With Reference Variables

Uncontrolled proliferation of reference variables leads to the same kinds of problems as the uncontrolled use of gotos or nonlocal variables. The solution in the latter cases comes from providing constructs that express clearly all the "well-structured" uses of the "undesirable" construct. In Alphard, the protection mechanism and the ability of a form to encapsulate information have partially solved the problem. We now allow one-level (but not multi-level) references. It remains to be seen whether this is a complete solution.

Parameterization of Abstractions

It is not sufficient to be able to describe a particular abstraction; you must be able to write classes of related ones. For example, you should be able to write a description of a stack with a parameter indicating the type of element to be stacked -- you certainly do not want to have to write separate abstractions for stack-of-integers, stack-of-reals, etc.

THE NEXT SET OF DATA STRUCTURE PROBLEMS FOR ALPHARD

Although a number of proposals exist, the problems in this next group are not quite as close to solutions as the previous ones. Many came up during the initial design of Alphard but were deferred until the second round.

Specification Languages

If a user is to rely entirely on the specification of a form and not on the description of its implementation, the specification language must be capable of expressing all the properties the designer of the form intends to provide.

Literals and Other Input/Output Issues

One of the goals of the abstraction mechanism is to make it possible for a programmer to extend the language with new types that have all the status and privileges of the primitive or built-in types. If this is to be the case, it must be possible for the author of a new abstract type to define the conversion rules for literals and for input and output of values of the type.

Duality of Data and Control

A user often wishes to view certain information as stored data when the implementor wishes to view it as a set of values to be calculated as needed. For example, the current length of a queue might be computed from indices of the first and last elements instead of stored (and maintained) as a variable. The converse can also be true; an implementor may choose to store tabulated values of a function if it is known that most of the accesses will hit the tabulated points.

The syntax of the language should allow the decision between implementation as a variable and implementation as a function to be changed without requiring massive changes to the code that uses the structure. This is often referred to as the "uniform referent" problem. If data may only be accessed, the solution is reasonably simple. However, the situation is substantially complicated when you wish to store into something specified as a variable and implemented as a function. One approach to this problem is presented in [Geschke75].

PROBLEMS FOR THE FUTURE

We can now imagine a series of increasingly complex problems concerned with the use of abstract data type definitions.

Several Implementations of a Single Specification

It is currently possible to write different implementations of a single specification. However, if the implementations are to be truly interchangeable, we must be able to verify that both versions meet the specification. We must also be confident that the specification captures all the things a user might depend on. In addition, it would be very nice if the data type definition described the non-functional properties (such as speed or space consumption) that differ among implementations; a user needs this to decide which of several implementations to use.

Library of Representations and Assistance with Selection

When it is possible to create several implementations for the same functional specifications, libraries of them will be created. Users will then want guidance about how to select implementations for the specifications they have chosen. Many things might influence the decision; space and time requirements are obvious candidates. Programming support tools should be developed to combine data on the performance of available implementations with projections on the way the abstraction will be used. This could then be used to supply advice on the selection of an implementation. These tools should also collect performance data on running programs to assist in decisions about changing implementations in order to increase overall performance.

Automatic Conversion of Representations

When the library-and-advice problem is under control, the next issue will be to update the data structures that

already exist when a representation is changed. This issue can arise any time an enduring data base is created; it is currently dealt with most often by requiring the change to be upward compatible or performing a massive (incompatible) update when the change is made. When several implementations are known to meet the same specifications, we can find ways to automate the updating process. An initial solution will be able to find and update all instances of an altered data structure. A complete solution will be able to update the representation gradually, so a particular structure will be updated only when it is used.

Mixed Representations in a Running System

Another problem that will be interesting when several implementations can meet the same specifications is that of allowing different instantiations of an abstract structure to have different implementations in a single program. We would like a programmer to be able to think of all his arrays as arrays and, in separate decisions, decide that some of them should be represented in row-major order and others in column-major order. The initial solution to this problem may impose the constraint that multi-operand operators can accept only one representation per call. Subsequent solutions should relax even that constraint.

Automatic Representation Conversion in a Running System

Once the previous two tasks are solved, we will want to combine the results to allow the representation of a single abstract data object to change while a program is running. There are already known cases where this is sufficiently desirable that it is done manually; for example, a compiler may use one symbol table organization while it is adding new names and convert it to another form before using it for lookup only.

The initial task is to simply allow a representation to be changed upon explicit request. However, if the facility is to be truly useful, we will need criteria that indicate when a change is in order. Developing such criteria is the second stage. The final step is automatic data collection and application of the criteria so that representation conversion is done automatically.

ACKNOWLEDGEMENTS

Many of the ideas described here have appeared elsewhere; I have tried to organize them so that they give a picture of currently interesting research topics. The Alphard research was supported by the National Science Foundation (Grant DCR74-04187) and by the Advanced Research Projects Agency of the Office of the Secretary of Defense (Contract F44620-73-C-0074), monitored by the Air Force Office of Scientific Research.

REFERENCES

[Dahl72] Dahl, O.-J. and Hoare, C. A. R., "Hierarchical Program Structures", (In Dahl, Dijkstra, and Hoare, *Structured Programming*, Academic Press, 1972).

[Geschke75] Geschke, C. M. and Mitchell, J. G., "On the Problem of Uniform References to Data Structures", *International Conference on Reliable Software*, April 1975.

[Jones74] Jones, Anita, and Wulf, William, "Towards the Design of Secure Systems", *International Workshop on Protection in Operating Systems*, IRIA, August 1974.

[Parnas1971] Parnas, D. L., "Information Distribution Aspects of Design Methodology", *IFIP Congress 1971*.

[Wulf73] Wulf, W. A. and Shaw, M., "Global Variable Considered Harmful", *SIGPLAN Notices*, February 1973.

[Liskov74] Liskov, B. and Zilles, S., "Programming With Abstract Data Types", *Proceedings of a Symposium on Very High Level Languages, SIGPLAN Notices*, April 1974.