Abstraction, Data Types, and Models for Software

Mary Shaw
Carnegie-Mellon University
Pittsburgh, Pa. 15213

In the area of software development and maintenance, a major issue is managing the complexity of the systems. Programming methodologies and languages to support them have grown in response to new ideas about how to cope with this complexity. A dominant theme in the growth of methodologies and languages is the development of tools dealing with abstractions. An *abstraction* is a simplified description, or *specification*, of a system that emphasizes some of its details or properties while suppressing others. A *good* abstraction is one in which information that is significant to the reader (i.e., the user) is emphasized while details that are immaterial, at least for the moment, are suppressed.

What we call "abstraction" in programming systems corresponds closely to what is called "modelling" in many other fields. It shares many of the same problems: deciding which characteristics of the system are important, what variability (i.e., parameters) should be included, which descriptive formalism to use, how the model can be validated, and so on

Many important techniques for program and programming language organization have been based on the principle of abstraction. Current abstraction techniques have evolved in step not only with our understanding of programming issues, but also with our ability to use the abstractions as *formal specifications* of the systems they describe. In the 1960's, for example, the important developments in methodology and languages centered around functions and procedures, which summarize a program segment in terms of a name and a parameter list. At that time, we only knew how to perform syntactic validity checks, and specification techniques reflected this: "specification" meant little more than "procedure header" until late in the decade. In the 1970's we recognized the importance of organizing programs into modules that localized information; this led to language support for data types and for generic definitions. The corresponding specification techniques include strong typing and verification of assertions about functional correctness. Since we can rely on formal specifications only to the extent that we are certain they match their implementations, the development of abstraction techniques, specification techniques, and methods of verifying the consistency of a specification and an implementation must surely proceed hand in hand [1].

Over the past five years, most research activity in abstraction techniques for programming languages has been focussed on one particular topic, *data types*. In a data type abstraction, we specify the functional properties of a data structure and its operations, then implement them in terms of existing language constructs (and other data types). To use the abstraction, we show that the specification is accurate, and we subsequently deal with the new type solely in terms of its specification. This philosophy was developed in CMU's work on Alphard as well as in several other language research projects, including CLU and Euclid. In addition, research directed specifically at specification issues, such as the algebraic formulation of data type specifications, has contributed to our understanding of the issues.

A particularly rich kind of abstract data type definition allows one abstraction to take another abstraction (e.g., a data type) as a parameter. These *generic* definitions provide a dimension of modelling flexibility that conventionally-parameterized definitions lack. The flexibility provided by generic definitions is demonstrated by the algorithmic transformation of [2], which takes any solution to one class of problems as a parameter and automatically provides a solution to the corresponding problem in a somewhat richer class. This generic definition is notable for the richness of the specified assumptions about the generic parameter. Definitions such as this one offer one way to systematically transform program definitions to meet specific needs; the potential and limitations of this kind of definiton provide a number of worthwhile research questions.

Over the past year I have grown dissatisfied with abstract data types as the primary focus of the effort on abstraction techniques in programming languages. Abstract data types fall short in two ways. First, the model they provide for organizing programs isn't suitable for all programs and components. Second, the research on abstract data types has concentrated exclusively on functional correctness, ignoring other important properties of programs. I have begun to address these two deficiencies in work that is a natural extension of my earlier work on data types.

Language investigations involving abstract data types,

together with other research projects, have addressed questions of functional specification in considerable detail. That is, they provide formal notations such as input-output predicates and algebraic axioms for making assertions about the effects that operators have on program values. In many situations, however, a programmer is concerned with properties other than pure functional correctness. Such properties include time and space requirements, memory access patterns, reliability, synchronization, and process independence; they have not been addressed by the data type research. A specification methodology that addresses these properties must have two important characteristics. First, it must be possible for the programmer to make and verify assertions about the properties rather than simply analyzing the program text to derive exact values or complete specifications. This is analogous to our approach to functional specifications: we don't attempt to formally derive the mathematical function defined by a program; rather, we specify certain properties of the computation that are important and must be preserved. Second, it is important to avoid adding a new conceptual framework for each new class of properties. In [3], I describe a result that allows specifications of execution time requirements to be made and verified within the framework that already exists for functional specifications.

A certain amount of work on formal specifications of extra-functional properties has already been done. Most of it is directed at specific problems rather than at techniques that can be applied to a variety of problems; the results are, nonetheless, interesting. The need to address a variety of requirements in practical real-time systems was vividly demonstrated at the conference on Specifications of Reliable Software. Other work includes specifications of security properties, reliability, and communication protocols. Some of the results, particularly the work on security and reliability, are concerned with reducing global properties of the program or distributed effects to theorems to be proved about individual operations. I have considered the use of a similar approach for dealing with some problems in language design in [4].

Efforts to use abstract data types have also revealed some limitations of the technique. In some cases, problems are not comfortably cast as data types or the necessary functionality is not readily expressed via algebraic axioms. In response, I am beginning to look into alternatives to the abstract data type model for program organization. In other cases, the problem requires a set of definitions that are clearly very similar but cannot be expressed by systematic instantiation or invocation of a data type definition, even using rich generic definitions. In this context, I would like to explore alternatives to the only operations currently well-understood for program abstractions, composition and instantiation. Introducing new kinds of models, of course, also raises questions about how to use more than one kind of model effectively in a single program.

A somewhat independent modelling effort is my interest in models for quantitative evaluation of software, especially the problem of developing and validating models that allow high-level properties of interest (e.g., maintainability, understandability) to be predicted in terms of low-level measures (e.g., statement profiles) and the problem of finding suitable criteria for comparing and evaluating models. Some problems with current criteria for evaluating metrics and models are discussed in [5], and some principles for the use of modelling techniques for quantitative software measurement are given in [6].

This workshop dealt with the mutual modelling interests of programming language, artificial intelligence, and data base researchers. These groups draw on different problems and motivations; in order for members of the various groups to share ideas effectively, it is important to understand how their modelling interests and techniques differ. At the workshop, four such distinctions emerged:

- Data (and data sharing) receives more attention in artificial intelligence and data bases than in programming languages.
- Implicit computations are currently of more concern and practical interest in artificial intelligence and data bases than in programming languages.
- Formal techniques, especially for specification, play a larger role in programming languages than in artificial intelligence or data bases.
- Large irregular heterogeneous data structures appear most often in artificial intelligence, least often in data bases.

Communication among these three areas will be simplified if these differences of emphasis are borne in mind. Mutual interactions will also be enhanced by moderation and open-mindedness in debate. In particular, the concepts *type*, *specification*, and *constraint* all appear to be quite soft. The same rule or computational effect may often be classified under more than one of these concepts, and it is important for any dialog on the subject to use the terms carefully. In addition, I often hear concepts, systems, or features described as globally "good" or "bad". This is unfortunate, for goodness depends on context -- on the setting in which a concept, feature, or whatever will be used -- and criticism that fails to take the context into account is often misguided.

The workshop raised several sets of issues as research topics in the three areas of concern. To programming language researchers, it raised the challenge of finding ways to describe and control

- implicit computation
- inconsistent, incomplete, or partially correct systems or information
- data sharing
- richer (i.e., non-hierarchic) type systems

without adding excess cost or complexity to languages. The workshop reinforced two of my own opinions about language research:

- that formal methods should receive more attention, and that sets of special-purpose transforms should be replaced by systematic calculi,

- that proliferation of special-purpose languages and sublanguages is a bad idea, but that providing a common base language with extension facilities is often a good idea

My recent work in abstraction techniques includes:

1. Mary Shaw, "The Impact of Abstraction Concerns on Modern Programming Languages." *Proc. of the IEEE*, to appear September 1980.

2. Jon Louis Bentley and Mary Shaw. "An Alphard Specification of a Correct and Efficient Transformation on Data Structures." *Proc. of IEEE Conference on Specifications of Reliable Software*, April 1979, pp.222-237. (To appear in *IEEE Transactions of Software Engineering*.)

3. Mary Shaw. "A Formal System for Specifying and Verifying Program Performance." Carnegie-Mellon University Technical Report CMU-CS-79-129, June 1979.

4. Mary Shaw and Wm. A. Wulf. "Toward Relaxing Assumptions in Languages and Their Implementations." *ACM SIGPLAN Notices* 15, 3, (March 1980).

5. Mary Shaw. "When is 'Good' Enough?." In *Draft Software Metrics Panel Final Report*, *Yale University Research Report* 182/80, June 1980. Final version to be published by MIT Press, Fall 1980.

6. J. C Browne and Mary Shaw. "Toward a Scientific Basis for Software Evaluation." In *Draft Software Metrics Panel Final Report*, *Yale University Research Report* 182/80, June 1980. Final version to be published by MIT Press, Fall 1980.

Research support

Several research projects are mentioned here. They have been supported by the National Science Foundation under Grant MCS77-03883, by the Office of Naval Research under Contract N00014-79-C-0672, and by Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under contract F33615-78-C-1551. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the US Government.