Data types and program correctness*

by BARBARA H. LISKOV
Massachusetts Institute of Technology
Cambridge, Massachusetts

One of the most important current software issues is reliability, and accordingly, a major criterion of programming language design must be that the language contribute to the production of reliable programs. Although there are other important aspects of software reliability (e.g., fault tolerance), the most fundamental is program correctness: does the program do what it is supposed to do? A language can contribute to this goal by enhancing the provability of its programs. This paper discusses the impact of user-defined data types on program provability.

The principal motivation for having a language support user-defined data types is that they contribute to software reliability by enhancing the programmer's ability to use abstraction in writing programs. Abstraction plays an extremely important role in programming because it is the main tool available for controlling the complexity of programs. Thus the process of structured programming is based primarily on recognition of useful abstractions. The abstractions provide a basis for problem decomposition: the original problem is solved by a program which uses the abstractions, and each abstraction becomes a new problem to be solved.

There are (at least) two kinds of abstractions used in programming: functional abstractions and data abstractions. User-defined data types support the latter; in the absence of a type-definition facility, the use of data abstractions is at least difficult, if not impossible. However, many languages which permit new data types to be defined view new types as a name for some selected storage representation (e.g., Pascal, Algol 68). This is inadequate because it ignores the fundamental connection between a data type and a set of operations which are meaningful for its representation. This means that the new type is not being used abstractly. Of course, procedures may be written to provide the meaningful operations, but these are defined separately from the type definition, so the relationship between them is not apparent.

The connection between data types and operations has been noted before, and recommended as a technique for treating types abstractly. However, most work in this area has been concerned with identifying a distinguished set of operations which must be defined for every new type. For example, Balzer proposed a set consisting of four operations to access, update, insert, and destroy abstract data collections: each new type definition must specify how each of these operations is to be implemented.*

The problem with this approach is that there is no guarantee that the set of distinguished operations corresponds to the meaningful set of operations of the type. In fact, followers of this approach have noted that there are types which cannot be defined in terms of the distinguished set of operations.* As before, the additional operations could be defined by procedures, requiring the use of two separate mechanisms.

An approach which can handle any abstract data type by a single definition mechanism is to permit the programmer who identifies the abstract data to define his own set of operations as part of the type definition. This requires that the language provide the proper type-definition mechanism: a type definition will consist of a description of how objects of the new type are to be represented in storage, and algorithms, written in terms of the representation, for all the operations.

One advantage of this approach is that all information about a type-definition is gathered in a single place and supported by a single construct (for example, the class construct of Simula 67). This leads to simpler, more understandable programs. A more important advantage is that it is possible to encapsulate the type definition, so that users of the type can manipulate objects of that type only via the defined operations. In particular, the representation of objects is not visible outside the type definition. The type checking mechanism of the language can be used to enforce this. Languages containing such encapsulated mechanisms are under development.*

Encapsulating a type definition enhances the provability of programs because it permits proofs to be decomposed around the type definitions.* A single proof is given that the type definition implements the type correctly; this is the only proof in which implementation details must be considered. The proof of a program using the abstract type depends only on the abstract behavior of the type.

Consider the proof that a type definition correctly implements a type. This proof depends on the fact that en-

---

* This work was done with support of IBM funds for Research in Computer Science.

* Hoare makes this point and gives a sample of proof in Reference 11.
capsulation ensures that assumptions made by the operations about the meaning of the representation cannot be invalidated by actions outside the type definition. For example, the abstract data type, stack, with operations create, push, pop and empty, might be represented (in Pascal³) by

```pascal
record
  top: integer,
  data: array[1..100] of integer
end
```

Note that not every possible configuration of this structure is a legitimate stack; in legitimate stack representations, for example, \(0 \leq \text{top} \leq 100\), and data \([1], \ldots, \text{data[top]}\) contain the pushed values, in the order in which they were pushed. The proof of the stack definition will consist of proofs that each operation behaves correctly, and each such proof assumes the operation is passed a legitimate stack object, and shows the operation returns a legitimate stack object. In the absence of encapsulation, proofs about the legitimacy of stack objects must be given in all programs using stacks, which is much more work.

The interface between the type definition and the users of the type must be precisely specified if this proof technique is to be successful. The encapsulation of the type definition ensures that the behavior of a type can be specified without describing how objects of the type are represented. For example, statements about the legitimacy of stack objects need not be included in a specification of the abstract type, stack. Thus encapsulation leads to simpler and more understandable specifications by reducing the information which must be expressed.¹²

The emphasis in the preceding paragraphs has been on program provability. Although this is interesting in its own right, it is also important because of its close relationship to program understandability. Understanding a program is the basis of an informal proof technique in which a programmer reasons about the meaning of a program (either his own or someone else's) in order to convince himself that it behaves correctly. The arguments advanced above concerning the relationship of data types to provability apply equally well to understandability.

REFERENCES