A Next Step in Data Structuring for Programming Languages

Jim Mitchell
Ben Wegbreit
Xerox Palo Alto Research Center
Palo Alto, California

Many current languages, such as Pascal, associate a **type** with each object in the language. Recent research has attempted to generalize the notion of a simple type to include **parametrized types**, types which can take other types and values as parameters. The idea here is to be able to define a data structuring mechanism, such as a stack, which is parametrized with the type of the values to be on the stack. Clu [1] and Alphard [2] are the two best known examples of this approach. We propose a framework for this research, in the form of two programming language constructs: abstractions and schemes.

An **abstraction** provides a statement of the properties of a data type without any commitment to a specific implementation. Programs which use abstractions can be developed and verified without regard to how the abstraction will be realized. Any type which fulfills the requirements of the abstraction can be bound in its place without affecting the program's correctness or its proof.

In [4], the notion of a **scheme** as a model for a set of types is developed. A scheme is written as a module which takes normal values plus types as formal parameters. Instantiating a scheme with actual parameters yields a *scheme instance* which is a data type. For example, one can write a scheme for AVL trees [5] which takes the type of the keys in the tree as a parameter. An instance of this scheme, for example, AVL trees of strings, is a normal data type.

Abstractions and schemes are independent concepts for facilitating data structuring. They do, however, have three points of contact.

(1) An instance of a scheme, being a data type, may be a faithful realization of an abstraction.

(2) An abstraction and a scheme may be written with the same type parametrization so that each instantiation of the abstraction is realized by an analogous instantiation of the scheme.

(3) It is useful to be able to impose constraints on the parameters to a scheme. This occurs when a scheme definition needs some argument to possess certain properties. Abstractions provide a convenient means for expressing constraints on type parameters.

A type can realize an abstraction, and an instance of a scheme, being a type, can realize an abstraction: a scheme will be said to realize an abstraction if it can be shown that all possible instantiations of it realize the abstraction. By analogy with first-order logic, we might call such a scheme a *model* for the abstraction.

For example, assume that we had an abstraction for the algebraic definition of *field* which captured its properties, vis., additive and multiplicative group with associativity and distributivity. We could define a scheme, *Complex*, which implements the notion of complex numbers over some field, using rectangular coordinates, say. We might wish to prove that all instances of this scheme were themselves, but that proof will certainly require that its single type parameter also be a realization of a field: cf. point (3).

It should be noted, in passing, that we could also define an abstraction for a *complex field*; then, the scheme *Complex* could be a realization of it as well as of the *field* abstraction. Another *Complex* scheme using polar coordinates, say, might provide yet another realization.

The advantage of such an approach, of course, is that one could build a library of completely general data structuring mechanisms, each of which would *never* have to be programmed again. Given such a facility, even relatively large expenditures of time and money to verify those mechanisms become worthwhile because they can be amortized over many uses. We also believe that this approach will result in more reliable software because of the reliability of these very important components. Likewise, one will be able to improve the efficiency of a system by replacing one scheme by another, abstractly equivalent scheme which is better suited to the particular usage patterns in that system.

In order for such a framework to be satisfactory, however, there are some other considerations which must be attended to. The following, somewhat cryptic list covers these and gives simple examples intended to motivate rather than logically justify each requirement (details are discussed in [4]):

(1) Schemes should be able to manipulate objects directly, and not only indirectly via pointers (although they clearly must be able to do that, too). If this is not so, then simple notions such as complex numbers will either be inefficient or will not be done within the scheme framework.

(2) One should be able to manufacture new, compound schemes by naming other schemes rather than reprogramming them. For instance, if one has a *list*

scheme, and a *reference counting* allocation scheme, it should be trivial to manufacture a new scheme, *rclist*, to provide lists with nodes that are allocated using a reference counting scheme. A more difficult and perhaps more useful combination scheme would use the *list* scheme twice in the definition of a new data type so as to define nodes which reside simultaneously on two lists; this is a commonly occurring phenomenon in systems programming.

(3) It must be possible to control the access to a data structure in order that the verification of the scheme from which it came can be valid.

(4) Separate schemes which implement the same abstraction should have the same name; i.e., schemes should be *generic* (for example, two *Complex* schemes, one using rectangular, and one using polar coordinates).

(5) It must be possible to impose *constraints* on the parameters to a scheme. In order to specify or select one scheme from a set of schemes all realizing the same abstraction, for example, some properties of the parameters will need to be examined. For instance, if one needed a hash table, which scheme to pick might well depend on the type of the keys (fixed length or varying), or the expected number of entries in the hash table (small numbers suggest a linear-probe hash table, enormous numbers might require something like a disk-resident data base regime such as B-trees [3]).

REFERENCES

[1] Liskov, B. and Zilles, S., Programming with Abstract Data Types, in Proceedings of Symposium on Very High Level Languages, *SIGPLAN Notices* 9, 4 (April 1974), 50-59

[2] Wulf, W.A., Alphard: Toward a Language to Support Structured Programs, Department of Computer Science Internal Report, Carnegie-Mellon University, Pittsburgh, Penn., April 1974

[3] Bayer, R. and McCreight, E., Organization and Maintenance of Large Ordered Indexes, *Acta Informatica* 1, pp. 290-306 (1972)

[4] Mitchell, J., and Wegbreit, B., Schemes: A High Level Data Structuring Concept, to appear in *Current Trends in Programming Methodologies*, R. Yeh, ed., Prentice-Hall.

[5] Adel'son-Vel'skii, G.M., and Landis, E.M., AVL Trees, in *Soviet Math.* 3, pp. 1259-1263 (1962).