M.I.T. Laboratory for Computer Science

Request for Comments No. 245
March 24, 1983

Efficient, Multi-Language Access to Persistent, External Data

by James W. Stamos

NOTE: This note was hastily written for another purpose, but will suffice as an introduction to my preliminary thoughts on a thesis topic.

1. Introduction

Conventional programming systems suffer from two important limitations when viewed as general information systems. First, changes to the programming environment are transient. Unless the user or application explicitly saves state information in a file that is external to the programming language and its memory management mechanisms, the results of one session are not visible in later sessions. Programming language constructs do not permit the saving of relevant portions of the environment with the guarantee of type safety across time and space. One of the few examples of a *persistent* programming environment is Smalltalk-80 [4]. The entire programming environment resides in a conventional disk file and exists in a quiescent state between user sessions. Since Smalltalk-80 objects do not leave the virtual memory, the user and the application programmer never need to consider translation issues, type checks, version mismatches, or storage management considerations. However, one shortcoming of Smalltalk-80 is the assumption of a single user. Smalltalk-80 provides no support when two or more users wish to share a collection of objects.

The second relevant limitation on most programming languages is the lack of support for sharing instances of user-defined types. Arbitrary sharing of information with type safety requires the sharing of type information and an agreement on type equivalence/conversion within a language and between two or more languages. Since different users typically have different address spaces, inter-user sharing requires the notion of transmitting an abstract value from one environment to another. Although some optimizations may be made when sharing occurs within one time-shared host, the general case involves separate hosts and communication through a computer network.

Removing the transience of data restriction and permitting sharing of abstract values closes the gap between programming languages and databases. Recent overlap in the research activities of both communities [1, 9, 10] indicates a potential for success with an integrated system. For example, there are many plans for installing database access constructs in new or existing languages [2, 3, 8]. On the

other hand, database users implicitly write a program when constructing a query. Generalized notions of queries that permit arbitrary computations will probably arise from the addition of ordinary programming language constructs to query languages.

In order for a repository of external data to achieve its full potential, the data representation, the notion of types, and the communication prototols need to be as general as possible. The plethora of different programming languages, operating systems, and machine architectures, coupled with the decreasing cost of network attachment, argue strongly for a single standard or a small set of such standards. On the other hand, the frequent manipulation of external data will demand an efficient implementation. The classic performance-generality tradeoff must be made.

2. Research Issues

Within the framework of persistent, external data, one particularly interesting area is the efficient translation of instances of user-defined types to and from an external representation [5, 6]. The amount of work that may be done at compile time, link time, and run time is language dependent. Early binding and optimization may play an important role in determining the translation speed and hence overall usability.

A second question is the degree of programmer involvement and loss of modularity required for acceptable performance. Herlihy [5] describes a *template* scheme in which the implementor of an abstract data type must write only two procedures, *encode* and *decode*, in order to make that type transmissible. While simplifying program development, this technique may incur excessive amounts of overhead even when used in conjunction with a sophisticated compiler. A slight reduction in program modularity could increase performance without adversely affecting the overall structure. For example, in many cases a programmer employs a small set of hidden abstract data types when implementing a higher-level data type. Efficiency constraints may require that the implementor/compiler of the encode and decode operations for the visible type make use of the representations of the lower level types. This restricted breach in modularity is analogous to the controlled exporting of nested interfaces by MESA configurations [7]. Configurations permit the sharing of detailed knowledge between participating modules and subconfigurations but keep this information from users of the configuration.

A third issue is the tradeoff between generality and efficiency. If only applications written in the same programming language accessed the external data, the transmission, translation, and representation could all be optimized for this special case. Introducing other languages adds further constraints and requires a more general solution. At some level, however, most programming languages can view information as a structured collection of objects and a set of references between objects. For example, except for at most a few minor details, a memory management scheme developed for one language with a heap is usually applicable to a wide variety of other heap-based languages. This degree of commonality may permit the existence of efficient translation and transmission schemes that accommodate a majority of existing programming languages.

Optimizing transmission for identical implementations of an abstract data type and/or identical languages invites attention because it provides a number of benefits. Such special cases may act as benchmarks and indicate the relative efficiency of general translation schemes. Restricted use of such optimizations for frequently transmitted data types may substantially increase performance with only a minimum reduction in modularity and portability. Particular optimization attempts may also lead to a better understanding of the general transmission technique and suggest methods for improving the latter's efficiency.

3. Applications

The efficient translation and transmission of abstract values will play a key role in numerous applications. Sharing may be done across time, space, or both. For example, two users connected by an internet can easily exchange data regardless of their programming languages or host machines.

A general translation mechanism also simplifies making dynamic changes to data type representations, programming languages, and operating systems. The current system would first convert all existing data to its external representation. After changing to the new implementation, programming language, or operating system, the external data would be retrieved and automatically converted to the new internal format.

Remote databases that interact with users during a query or transaction will also send and receive abstract values. Copying, caching, movement, and replication techniques for both centralized and distributed database systems can also benefit from the efficient translation and transmission of structured data.

References

[1] ACM. Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modelling, Pingree Park, Colorado (June 23-26, 1980). *SIGPLAN Notices* 16, 1 (January, 1981).

[2] Allman, E. Stonebraker, M., Held, G. Embedding a Relational Data Sublanguage in a General Purpose Programming Language. *SIGPLAN Notices* 8, 2 (1976), pp. 25-35.

[3] Earley, J. Relational Level Data Structures for Programming Languages. *Acta Informatica* 2, 4 (1973), pp. 293-309.

[4] Goldberg, A., and Robson, D. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[5] Herlihy, M. *Transmitting Abstract Values in Messages.* MIT Laboratory for Computer Science MIT/LCS/TR-234 (April, 1980).

[6] Herlihy, M. and Liskov, B. A Value Transmission Method for Abstract Data Types. *ACM Transactions on Programming Languages and Systems* 4, 4 (October, 1982), pp. 527-551.

[7] Mitchell, J. G., et al. *Mesa Language Manual.* Xerox PARC CSL-79-3 (April, 1979).

[8] Schmidt, J. W. Some High Level Language Constructs for Data of Type Relation. *ACM Transactions on Database Systems* 2, 3 (September, 1977), pp. 247-261.

[9] Smith, J. M. and Smith, D. C. P. Database Abstractions: Aggregation. *Communications of the ACM* 20, 6 (June, 1977), pp. 405-413.

[10] Smith, J. M. and Smith, D. C. P. Database Abstractions: Aggregation and Generalization. *ACM Transactions on Database Systems* 2, 2 (June, 1977), pp. 105-133.