Strategies for Data Abstraction in LISP

Barbara K. Steele

Massachusetts Institute of Technology
Artificial Intelligence Laboratory
545 Technology Square
Cambridge, Massachusetts 02139

Abstract

The benefits of abstract data types are many and are generally agreed upon [Liskov and Zilles 1974, Linden 1976]. New languages are being constructed which provide for and enforce the use of data abstractions [Liskov et al 1977, Wulf et al 1976]. However, many of us are not in a position to use these new languages, but must stick to our installation's compiler. How then can we obtain the benefits of data abstraction? We discuss the implementation of data abstraction in a LISP program and the subtleties involved in doing so: specifically, how it is possible to enforce proper data abstraction in a language which does not provide for abstract data types.

I. INTRODUCTION

Programmers typically write the actual code for a program after the problem has been defined and analyzed, and after an appropriate data representation has been selected. This allows assumptions about the structure of the data to be shared by all parts of the code. The knowledge of where to obtain a certain piece of data or how to construct a given output form is incorporated in code throughout the program. Unfortunately, any necessary changes or optimizations in the structure of the data will therefore require major revisions in the program in order to correct all references to that data, as well as to any data dependent on it [Linden 1976]. Data abstraction is an alternative to this conventional method of programming which largely eliminates its unfortunate effects.

This report describes research done at both Argonne National Laboratory in Argonne, Illinois, and at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. The work was supported in part by the National Science Foundation and the Advanced Research Projects Agency.

Data abstraction is a programming strategy which calls for the separation of data definition from program definition. All the information about the representation of the data used in the program is coded in modules independent from the functions which use that data in performing the actual computations of the program. The program is thus decomposed into program functions (those which perform the actual computations of the program), and data modules (modules that contain information about the organization of various data structures). We wish to keep the assumptions each makes about the other to a minimum. The program functions need know nothing of the representation of the data structures. Instead, the representation of each data object is hidden within a data module. Each module contains the selection and construction functions which operate on the object. Regardless of the language we use, we may implement data abstraction simply by writing functions for selecting and constructing data, and using these data module functions within the program functions every time a piece of data is referenced. In this way we maintain the separation of program definitiona from data definition, and accrue all the advantages of data abstraction even though our language is not specifically set up to enforce that strategy.

In LISP, for example, instead of writing `(CADR DIRECTORY)` to obtain a telephone number which we know is stored in the second position of the list, we would write a separate function called `GET-TELNO` which alone knew how to obtain the telephone number from `DIRECTORY`. Then if the representation for `DIRECTORY` ever changed, only the function `GET-TELNO` would have to be modified. In place of `(CADR DIRECTORY)`, we would write `(GET-TELNO DIRECTORY)`. `GET-TELNO` then becomes a member of the data module associated with the `DIRECTORY` data object. Every time we wish to select or construct data which is part of the `DIRECTORY` structure, we call the appropriate pre-defined function. All such functions together form the data module which define the `DIRECTORY` structure. To change the representation of the directory, we need only change the definitions of the functions in this data module. Programs which reference functions in the data module should remain unchanged.

Thus, a data module is simply the group of definitions of the operations which can be performed on a given data object. Each time data is to be manipulated by the program, a call is made to the desired function within the appropriate data module. In this way, if the representation of some data object is to be changed, only the functions in that object's corresponding data module must be modified. If data abstraction had not been practiced, all those places in the program where that data object was referred to would have to be located and changed appropriately. In significant programs, such a task would discourage many a programmer, and might inhibit the change to a perhaps more efficient data representation.

When it actually comes down to writing the data modules, many subtle situations arise in which deciding what the "right thing" to do is is difficult. To aid the programmer in this process, we have compiled some guidelines for implementing data abstraction in a program. We point out some issues that may arise and discuss various methods of dealing with them. We have developed these suggestions from our own use of data abstraction in a LISP program of significant size [Kerns 1977]. They are not to be taken as hard-and-fast rules, but items to consider when deciding which is the proper method of abstraction.

II. DATA ABSTRACTION GUIDELINES

II.1 Sharing Data Modules within a Program

Q: If a function FOO is written to select some piece of a data object, say, the name field of an entry in the file STUDENTS, then later it is observed that some piece of another data object (perhaps the name field of an entry in the file PROFS) is accessed in the same way, may FOO be used to access either piece of data wherever one or the other is needed in the program?

A: Although the data objects STUDENTS and PROFS may share the same representation, each object should have its own set of operations since their representations may change independently. If the representation of PROFS changed, then a new data module full of operations would have to be written, and modifications would have be made to the program to point references to data in the PROFS file to operations in the new data module, thus violating the whole purpose of data modules.

It is important for the programmer to distinguish between instances of the same data type, and two completely different types of data that might happen to currently use the same representation. Entries in the Quuxton telephone book, for instance, constitute instances of the QUUXENTRY data type. They may of course use the same accessing and constructing operations defined for them. But just because the Quuxton telephone book and the Quuxton Electric Company use the same format for their data does not mean they should use the same data module full of operations. Either of their representations may change independently of the other. If that happened, a program meant to correlate information from the two of them would be in trouble.

In order to guarantee that two different kinds of data do not use any of the same selector or constructor functions, all those functions which select or construct pieces of the same data object should be grouped together into one data module. Although some languages (like PL/1) provide a way to enforce such groupings by implementing them as a procedure with entry points for each of the functions in the module, in LISP we must be content with an informal "logical" grouping of the functions, as discussed above. Functions belonging to one data module should be independent and distinct from functions in another data module. The distinction should be such that any change in representation made to a given data object should require changes to be made within only one data module. The program may then remain unchanged and oblivious to the fact that any data representation has been altered.

II.2 Sharing Functions within a Data Module

Q: It is sometimes difficult to know when not to reuse a function within a data module. Consider the following example: we are given a list which represents a function call (here we consider the representation of the function call as data, as it would be to a compiler, for example). The first element of the list is the function itself, and the remaining elements are the arguments to be passed. We wish to obtain the list of arguments from the function call, so a selector function is written to do this:

```
(DEFUN GETARGS (DATA)
       (CDR DATA))
```

The list of arguments is passed to program function which requested it. The program function processes the first argument, then calls itself recursively on the remainder of the list. That means we need two more selector functions; one to obtain the first argument in a list of arguments, and one to select the list of all but the first argument. Can we use the selector function GETARGS to obtain the list of all but the first argument from the list of arguments?

A: If it has been decided to represent the function call as a list before the program has been coded, it may be a temptation to use the same function to obtain both the entire argument list from the function call, and all but the first argument from the argument list. If two separate functions are written, one notices that both functions return all but the first element in the list they are passed, and both return a list of arguments. But they are passed different kinds of data, and furthermore, one returns the entire argument list, while the other returns only part of it. Consider the situation if this data had been represented as a tree, with the name of the function as the head node and each of its arguments as subtrees. It is clear that distinct functions would be required; a function to produce the entire argument list would return a list of the subtrees of a tree, and the function to produce all but the first argument from an argument list would return all but the first tree in a list of trees (or "forest").

Note that one might simply define the first function to return a list of arguments, while making no commitment as to whether that list is part of the internal representation of a function call; i.e., the list may or may not be freshly consed. This allows one to legitimately use CAR, CDR, MAPCAR, etc. on the list. Or does it? We address this question in section II.4.

In any case, in order to avoid the mistake of reusing a function when a different one should be written, it is necessary for the programmer to completely free his mind of the current representation of the data. If possible, the data representation should not be chosen until after the program is coded. Whether the data representation has been chosen or not, keeping in mind an alternative data representation while writing the program is helpful. Hierarchial naming of data module functions can be very helpful also. Consider renaming GETARGS to FCALL!ARGS, meaning "here is a function call, return the arguments." To strengthen the intended purpose of the function in the
programmer's mind, consider this definition of FCALL!ARGS:

```
(DEFUN FCALL!ARGS (FUNCTION-CALL)
       (CDR FUNCTION-CALL))
```

That is, instead of the obscure name "DATA", we use "FUNCTION-CALL" as the dummy argument. Unfortunately, LISP has no built-in provision for user-defined object types, but it certainly isn't difficult to cons them up and write functions to do type checking if one is so inclined. In many cases it will severely damage programming bugs.

II.3 Hierarchy of Functions in the Data Modules

Q: When a function is written to select a piece of data, how much data should be passed to it from which to obtain the desired piece? That is, should the selector function be passed an entire file, or just a record, or what?

A: We don't need to worry about the volume of data passed between functions since LISP just passes pointers. However, there are two disadvantages of passing entire data structures. If one requires that the entire data object be passed to a function regardless of the piece of data to be obtained, then: (1) functions become less modular. Consider the file/record example. If a program function PROCESS-RECORD deals only with a record and requests pieces of information from it occasionally, but we have written operations that can access that information only when handed the entire file it is found in, then PROCESS-RECORD can only deal with with records in the context of a file (which is pretty silly). (2) there is a loss of efficiency, since functions may have to perform the same work over and over. At one point in a selector function, we may refer to a certain record, in which case work is done to obtain the record from the file. Then later we may refer to a particular field in that record, in which case the field selector function must find the record all over again and then find the field.

Instead, we may choose to pass only the relavent piece of data; for example, the desired record for PROCESS-RECORD instead of the whole file. This leaves us with the problem of telling a selector function exactly how much data it is receiving from which to obtain the desired piece. Suppose we wish to obtain the last name of the person whose record we are now processing. Although we could easily write a function to accept the record and return the last name from the name field of the record, it may be that in another program function we have just the name field and wish to obtain the last name from that. Rather than write as many functions to obtain the last name as there are types of data to obtain it from, we restrict ourselves to functions which obtain a piece of data "x" from the smallest piece of data "y" which contains x. That is, to obtain x, we pass to the function the data y, where x is a subset of y and there are no subsets of y containing x other than x. A "subset" here is defined to be an accessible piece of data, where the degree to which data is broken down into "accessible" pieces depends on the requirements of the program.

Similarly for those data module functions which construct data, we pass a first name and last name to a function which returns the name field. This field is then passed with the other fields to a function to construct the record, and so on, gradually building the file up section by section. The alternative would be to require every individual piece of data to be ready at the same time, and then pass them to a function which would construct the resultant file all at once.

Just one comment on our comments: the strategy suggested above may in some cases "give the representation away." It can at least reveal the hierarchial structure of the data; if this is undesirable, the programmer must decide what canonical pieces of data to pass from which to obtain smaller pieces. In any case, we strongly recommend that the selector and constructor functions be named in such a way as to remind the programmer what type of data the function expects to receive (such as "ARGLIST!FIRST-ARG" to select the first argument from a list of arguments, "CONDITIONAL!CLAUSES" to obtain the list of clauses from a condtional expression, etc).

> Q: What about intersecting subsets of data, for example, rows and columns of a matrix?

A: As mentioned before, the degree to which data is broken down depends on the requirements of the program. For this example, one would of course want a function to obtain some row from the matrix, and another to obtain some column. Then naturally, to obtain certain values in the matrix, one could write functions to obtain these values from either a row or a column, or both, thus making the path for data acquisition non-unique. If it is necessary in a program to be able to access data in more than one way, functions can be written for each of them. Remember, though, that this means that if the representation changes, more than one function definition within the data module will have to be changed. The alternative would be to write only one function which could be passed a flag in addition to the data, telling it what kind of data was being received. For example, one could write a function to obtain a value from either a row or a column of a matrix, and then pass it both the row or column and a flag identifying which it was that you were giving it. The function would then do the right thing to obtain the desired value. This method is somewhat awkward, and undesirable in that the flag spoken of is just the kind of thing which makes connections between modules strong, a property we wish to minimize. Furthermore, unless there is some reason to compute the flag at run time, it is really just part of the name of the selector:

```
(FETCH 'ROW data)  <==>  (FETCH-FROM-ROW data).
```

II.4 Inherent Data Structures

> Q: LISP operates only on atoms and lists, and all user defined structures must be designed using these inherent structures. Furthermore, there exist built-in functions to construct and select pieces of lists. Must functions be written to select pieces of data which are represented as lists, even though CAR, CDR, CONS, MAPCAR, and others are pre-defined and available?

A: Data module functions are still relevant in this environment, since the data which is currently represented using a built-in structure may change and use some other structure, perhaps user defined. Often, however, the program works with a list of data in which each element is the same data type; for example, a list of arguments as discussed in section II.2. There really is no better way to represent such a list of homogeneous data than as a list, and since LISP contains built-in functions for selecting and constructing lists, it seems pointless to hide the obvious representation by writing special data module functions. But perhaps we can argue the point by suggesting that, in this case, the access within or construction of such "real" lists can be hidden in a <u>program</u> function. Consider the case in which a change must be made to the program (motivated perhaps by some data representation modification, or an actual modification to the logic in the program) where each time the selection of some element of a "real" list occurs, a certain procedure is to be performed. If access to the elements were not hidden, the programmer would have to search diligently through the program for every occurence of that event. However, it is a simple task to add a procedure call (or even to embed the actual procedure) in the program function which performs the selection or construction of the "real" list.

II.5 Additional Data

> Q: What if a program is altered to accept data types in addition to data types it already processes? Not just more data, but a new kind of data containing different information. For example, say that a record in the DIRECTORY file is currently represented as: (<name> <house number> <street> <city> <phone no.>) and the data module for the DIRECTORY structure contains the appropriate selector functions. Now suppose that the program which builds the DIRECTORY file begins handing <u>our</u> program file records of the form: (<index> <name> <house number> <street> <city> <phone no.>) where <index> is some obscure piece of information that our program doesn't care about.

A: Although the program can be made immune from changes in the representation of the data via data modules, it can not always be protected from additional data, when this new data is a different type than what had been processed in the past. If it is merely a matter of accessing original data in a new representation containing additional data, such a change can be compensated for in the data modules. But if the program is expected to process the additional data, then this constitutes a change in the program, and the data modules can not be counted on to handle it. Processing additional data constitutes a change in the specification of the program, not in the internal implementation. Analogous statements can be made about the removal of data.

II.6 Internal Data

> Q: What about data which is internal to the program? That is, data which is selected and constructed from values determined internally according to execution conditions and passed within or between functions. For example, a program function may wish to return two values, and so it conses them together. Should the functions receiving this pair simply take CAR and CDR of it, or must a data module be created with selector functions for obtaining pieces of it and a constructor function for forming it in the first place?

A: One might argue that a change to this data is a change to the program, and so hiding it in a data module is pointless. This is true to some degree; however, abstracted data is always easier to deal with. It makes the program easier to understand and verify: (CONS FOO BAR) stuck in the middle of a program is probably more confusing than (RETURN-MULTIPLE-VALUES FOO BAR). It makes the program easier to modify: suppose A creates internal data, and B and C use it. Then C needs additional information. If the representation of this internal data is hidden, only A and C must be modified. Or take the example given in the question. If three values must be passed, the CONS must be changed to LIST or some other more complex structure. The necessary changes are simplified if the internal data has been abstracted from the program. The point here is that it is a mistake to think of "the program" as monolithic. Abstraction of all data, regardless of where it comes from, always eases modification.

III. BENEFITS

At least four of the benefits of implementing data abstraction in a program are: (1) it allows greater flexibility in choice of data representation, (2) it enhances self-documentation, (3) it encourages one to consider the data at an abstract level, apart from program specification, and (4) it facilitates proving program correctness.

III.1 Flexibility

Data representations are easily changed when all that needs to be done to compensate for the change is to rewrite a few functions. One might decide against converting to a more optimal data representation if it means completely revising an existing program, but if data abstraction has been practiced, the necessary changes are simple and quickly made. In addition, because of the ease of conversion from one data representation to another, it is possible to confirm experimentally which is best for the program.

III.2 Self-Documentation

A program is self-documenting if the functions used are given appropriate names that aid a reader in identifying the piece of data being selected or constructed. This increased readability helps to reduce programming errors. Coding the program is facilitated as well, since each time a piece of data is needed, the programmer need not take time to recall the procedure needed to obtain it, but simply codes the appropriate aptly-named function call.

III.3 Abstract Data

In distinguishing data modules from program functions, data abstraction allows the programmer to disregard the details of data representation during program specification and to handle them, instead, when programming has been completed. Thus, not only is he encouraged to think of the data at an abstract level, but once programming is done, a more rational decision concerning optimal data representation can be made [Balzer 1967].

III.4 Proving Program Correctness

The task of proving program correctness is simplified when a program has been decomposed into program functions and data modules functions. That all of the functions it uses are correct can be taken as a set of axioms in proving a given function correct, thus breaking down the work that must be done to prove the entire program correct [Parnas 1971]. Of course, modularization other than that done in abstracting the data should be practiced in order to obtain the full benefits, but data abstraction will facilitate proofs of correct data selection and construction.

IV. PROBLEMS AND PARTING THOUGHTS

Run time for the decomposed program will undoubtedly prove to be much greater than for the original program due to the frequency of calls made to the data module functions. To avoid this overhead, a compile-time pre-processor which replaces each call by the appropriate in-line code is desirable [Parnas 1972, G. Steele 1977]. We have developed a set of optimizing transformations to be used with a source-to-source transformation system, discussed in [B. Steele 1980].

Although ideally one should not be concerned with efficiency while implementing data abstraction in a program, more pragmatically, it behooves one to take into consideration the capabilities of automatic optimizations when coding the program. For example, perhaps the same piece of data is selected more than once within a program function. Depending on how complicated the selection function is, the returned value would perhaps be best bound in a lambda expression to avoid having to select it again. The overhead of assignment should be avoided if possible, however. If the optimizer to be used has the power to note multiple uses of functions and perform a binding if the complexity of the function definition recommends it, the programmer need not be concerned with the issue of common sub-expression [Geschke 1972, Wulf et al 1975]. But if the optimizer is only capable of procedure integration for the numerous calls to data module functions, then the programmer might wish to give some thought to lambda binding the results of a call himself. There is program readability on the one hand, and speed of execution on the other. Clearly, program optimization and data abstraction go hand in hand; together they ease the programming task and bring us closer to the ultimate goal of automatic program synthesis.

References

Balzer, R. M.
"Dataless Programming." Proc. AFIPS 1967 FJCC, Vol. 31, AFIPS Press (Montvale, NJ, 1967), 535-544.

Geschke, C. M.
Global Program Optimizations. Ph.D. thesis. Carnegie-Mellon University (Pittsburgh, October 1972).

Kerns, B.
An Experiment in Information Hiding. B.A. Thesis, Greenville College, Greenville, IL, (May 1977)

Linden, T. A.
"The Use of Abstract Data Types to Simplify Program Modifications." Proc. of Conference on Data, ACM Sigplan Notices, 8:2 (1976), 12-23

Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C.
"Abstraction Mechanisms in CLU." CACM, Vol. 20, No. 8 (1977), 564-576

Liskov, B., and Zilles, S.
"Programming with Abstract Data Types." Proc. Symp. on Very High Level Languages. SIGPLAN Notices (April 1974).

Myers, G. J.
"Composite Design Facilities of Six Programming Languages." IBM Systems Journal, Vol. 15, No. 3, (1976), 212-224.

Parnas, D. L.
Information Distribution Aspects of Design Methodology. Tech. Report, Dept. of Computer Science, Carnegie-Mellon University (Pittsburgh, 1971)

Parnas, D. L.
"On the Criteria to be Used in Decomposing Systems into Modules." CACM 15 (December 1972), 1053-1058.

Steele, B. K.
An Accountable Source-to-Source Transformation System. S.M. Thesis, Massachusetts Institute of Technology, in preparation.

Steele, G. L. Jr.
"Debunking the 'Expensive Procedure Call' Myth." Proc. ACM National Conference (Seattle, October 1977), 153-162. Revised as MIT AI Memo 443 (Cambridge, October 1977).

Wulf, W. A., et al.
The Design of an Optimizing Compiler. American Elsevier (New York, 1975).

Wulf, W., London, R. and Shaw, M.
Abstraction and Verification in ALPHARD: Introduction to Language and Methodology. Tech. Report, Dept. of Computer Science, Carnegie-Mellon University (Pittsburgh, 1976)