Report 83-15
Stanford -- KSL

Scientific DataLink

Knowledge-Based Programming Using Abstract Data Types.
Gordon S. Novak Jr, May 1983

card 1 of 1

Heuristic Programming Project
Report No. HPP-83-15 (Revised June 1983)

May 1983

Knowledge-based Programming Using Abstract Data Types

Gordon S. Novak, Jr.

Heuristic Programming Project
Computer Science Department
Stanford University

To appear in *Proceedings of the National Conference on Artificial Intelligence*, Washington, DC, August 1983

Knowledge-based Programming Using Abstract Data Types[1]

Gordon S. Novak Jr.[2]
Heuristic Programming Project
Computer Science Department
Stanford University
Stanford, CA 94305

1. Abstract

Features of the GLISP programming system that support knowledge-based programming are described. These include compile-time expansion of object-centered programs, interpretation of messages and operations relative to data type, inheritance of properties and behavior from multiple superclasses, type inference and propagation, conditional compilation, symbolic optimization of compiled code, instantiation of generic programs for particular data types, combination of partial algorithms from separate sources, knowledge-based inspection and editing of data, menu-driven interactive programming, and transportability between Lisp dialects and machines. GLISP is fully implemented for the major dialects of Lisp and is available over the ARPANET.

2. Introduction

A compiler can be viewed as a program that, using knowledge embodied as data and procedures, converts a specification of a program into an executable program. Compilers for traditional programming languages have embodied a relatively small amount of knowledge and have not been easily extensible by the user. Restrictions imposed by traditional languages, for example, that a calling program and subroutine must be written in terms of identical data types, have inhibited the accumulation of programming knowledge in the form of reusable programs.

The power of a programming system can be measured by the leverage it provides, that is, by its ability to convert abbreviated specifications into sizable programs. To increase the power of compilers, it is necessary to increase the knowledge they contain and to make user-specified knowledge effective during the compilation process. This paper describes the knowledge used by the GLISP compiler and its associated programming systems, focusing on features that permit reusability of programs and accumulation of programming knowledge. The GLISP compiler provides a high degree of leverage in converting GLISP programs into efficient Lisp code.

3. GLISP

GLISP [5] [6] [7] is a high-level language that includes Lisp as a sublanguage and is compiled into Lisp. It provides a powerful abstract data-type mechanism that allows the structure and computed properties of objects to be described. Properties, predicate adjectives, and messages can be inherited from multiple superclasses. Compilation of properties is recursive at compile time and is performed relative to the types of the objects in question; this allows the same properties and behavior to be inherited by objects that are represented differently.

GLISP provides an object-centered programming system that allows messages to be interpreted at run time. A major advantage of GLISP compared to other object-centered programming systems is that when the type of an object is known, the GLISP compiler can determine the appropriate response function for a message to the object at compile time and can compile a direct call to that function or macro-expand it in-line. This provides the representational power of object-centered programming with no penalty in execution speed compared to ordinary Lisp.

4. Related Work

The use of messages and computed properties in GLISP is related to the use of messages in object-centered programming (OCP) systems [1] [3] [4]; GLISP contains a system for interpretation of run-time messages to objects and thus supports OCP. However, OCP has several inherent problems.

The first problem is that object-centered programs tend to be slow compared to programs written in the underlying language (typically 20 to 50 times slower). One reason for this slowness is that messages must be interpreted at run time; when the response to a message is a small amount of code (e.g., for data access), this overhead becomes a large fraction of execution time. Even with special hardware support, the overhead of message lookup is a significant cost.

A potentially more serious performance problem is caused by the referential opacity of OCP. Most program optimizations require some global knowledge about the program; that is, most optimizations are of the form "If both operations A and B are to be performed, there is a way to do A and B together that is cheaper than doing each separately." For example, if one wishes to make a list of the female "A" students in a class, it may be cheaper to compute the set of students who are both female and "A" students than to compute the two sets separately and intersect them. In an OCP system in which the female students and "A" students were found by sending messages, however, it would not be possible to perform this optimization because it would not be known how the two sets were computed.

---

[1] This research was supported in part by NSF Grant SED-7912803 in the Joint National Science Foundation - National Institute of Education Program of Research on Cognitive Processes and the Structure of Knowledge in Science and Mathematics and in part by the Defense Advanced Research Projects Agency under Contract MDA-903-80-c-007.

[2] Author's present address: Computer Science Department, University of Texas at Austin, Austin, TX 78712. Phone (512) 471-4353. Net address CS.NOVAK@UTEXAS-20.

Another problem caused by referential opacity is that error checking must be deferred to run time. If the response to a message is not looked up until run time, it will not be known until then whether the object can in fact respond to that message, whether the result objects it returns can respond to the messages that will be sent to them, and so forth; this complicates debugging. Borning and Ingalls [2] have reported an experimental compile-time type-checking system for Smalltalk.

By looking up message responses at compile time, GLISP eliminates the overhead of run-time lookup for most messages. Because messages can be expanded as in-line code, optimizations that span multiple messages (as in the example above) can be performed by the compiler. When object types are known, error checking is performed statically at compile time rather than dynamically.

5. Program Reusability

Much of the work in traditional programming consists of specializing standard algorithms for particular uses. The specializations performed by the programmer include instantiation of algorithms for particular data representations, combination of separate algorithms into a composite algorithm, and optimization based on knowledge of the conditions of use of the algorithm. In this section, compiler features that are necessary for compilers to perform such specializations are discussed; these features make it possible for standard algorithms to be written as generic programs and reused for different applications.

5.1. Data-type Independence

Most programming languages require that programs be written in terms of specific data types. Lisp is particularly bad in this regard because data access is performed by function calls; thus, dependencies on particular storage structures are built into program code. In GLISP, when a procedure is inherited as the response to a message or property reference, the static types of the actual arguments are substituted for the types of the formal arguments of the procedure; the inherited procedure can be specialized for the actual argument types either by open compilation (analogous to macro expansion) within the referencing program or by specializing it as a closed procedure. The GLISP object descriptions provide a level of indirection between the names of properties and the representation of the properties. Since substructures and computed properties are referenced in the same way in GLISP program code, a procedure can reference a property that is stored in one use and computed in another.

Additional compiler features are needed to achieve data-representation independence. Programs often write data as well as reading it. GLISP can "invert" an algebraic expression that ultimately involves only a single occurrence of stored data; this allows a program to "store" into a computed property so long as that property has a single "equivalent" property that is stored. For example, given a CIRCLE object whose RADIUS is stored, it is legitimate to assign a value to the AREA of the CIRCLE; the compiler produces code to store a RADIUS value corresponding to the specified AREA value.

Programs also create new data objects. Vector addition, for example, involves the creation of a new vector whose elements are formed by adding the components of the input vectors. To create a new vector that is like the input vectors, it is necessary to be able to specify a type that is the same as the type of another datum. GLISP provides a TYPEOF operator, which returns the compile-time type of the expression that is its argument. Thus, a generic vector-addition function can be used for different kinds of vectors:

```
(VECTORPLUS (GLAMBDA (U,V:VECTOR)
   (A (TYPEOF U) WITH X = U:X + V:X
                      Y = U:Y + V:Y)))
```

A particular kind of vector, a FVECTOR, can be described as follows:

```
(FVECTOR (CONS (Y BOOLEAN) (X STRING))
   SUPERS (VECTOR))
```

Given an expression "F+G", where F and G are FVECTORs, the compiler will produce the code:

```
(CONS (OR (CAR F) (CAR G))
      (CONCAT (CDR F) (CDR G)))
```

The operator "+" is defined for VECTORs as the function VECTORPLUS; this definition is inherited by FVECTORs, so that the function VECTORPLUS is open-compiled with the type FVECTOR for its arguments. VECTORPLUS produces a new object whose type is the same as the type of its first argument; the "+" operators within VECTORPLUS are interpreted according to the types of the components of the FVECTOR type, so that the BOOLEAN components are ORed and the STRING components are concatenated.

5.2. Multiple Views of Data

Real program data are often not of a single, simple type but may be viewed in several different ways. GLISP provides mechanisms whereby features of objects may be inherited from multiple views. The first such mechanism is inheritance from multiple superclasses. When a property is inherited from a superclass, it is compiled recursively in the context of the original object; the references made by the definition of the inherited property may then involve inheritance from different superclasses. This feature allows the user to have a number of shallow inheritance hierarchies rather than one deep hierarchy; the shallow hierarchies tend to be "cleaner" because each deals with only a limited facet of behavior.

In some cases, it is desirable to view an object as an object of another type without actually materializing the data involved in the view type. GLISP provides a virtual view mechanism that permits such views. For example, in the GEV editor, an item of the displayed data is viewed as containing areas on the display screen (the area around the item's name and the area around its displayed value). Using virtual views of the item, these areas are defined in terms of computed quantities (e.g., the number of characters in the item's name determines the width of the name area); the procedure that tests whether a point is inside an area can then be inherited to test whether the mouse pointer is selecting the item. The code that is produced by the compiler is written in terms of the data that is actually stored, so that an "area" datum does not need to be constructed in order to use the inherited generic procedure.

5.3. Combination of Algorithms

Real programs are typically composed of a number of smaller component algorithms that are combined and specialized for their particular use. For example, a number of iterative programs can be viewed as being composed of the following components:

Iterator: Collection → Element*

Filter: Element → Boolean

Viewer: Element → View

Collector:

- Initialize: nil → Aggregate
- Accumulate: Aggregate × View → Aggregate
- Report: Aggregate → Result

The Iterator enumerates the elements of the collection in temporal order; the Filter selects the elements to be processed; the Viewer views each element in the desired way; and the Collector collects the views of the element into some aggregate. For example, finding the average monthly salary of plumbers in a company might involve enumerating the employees of the company, selecting only the plumbers, viewing an employee record as "monthly salary", and collecting the monthly salary data for the average.

GLISP allows the operation of such an iterative program to be expressed as a single generic function; this function can then be instantiated for a given set of component functions to produce a single Lisp function that performs the desired task. Instantiation of generic functions is somewhat similar to instantiation of program plans in the Programmer's Apprentice [8]; in GLISP, there is a single language for both generic and concrete programs, and instantiation occurs by recursive expansion of code at compile time. Such an approach allows programs to be constructed very quickly. The Iterator for a collection is determined by the type of the collection, and the element type is likewise determined. A library of standard Collectors (average, sum, maximum, etc.) is easily assembled; each Collector constrains the type of View that it can take as input. The only remaining items necessary are the Filter and Viewer; these can easily be acquired by menu selection using knowledge of the element type (as is done in GEV, described below).

6. Compiler Knowledge

The GLISP compiler runs within a variety of Lisp systems; it embodies knowledge about the underlying Lisp system that helps make GLISP programs transportable. Implementors of Lisp systems have unfortunately introduced many variations in the names, syntax, and semantics of even the basic system functions of Lisp. GLISP performs the mapping from operations on various data types into the corresponding function calls in the host Lisp system; it also defines basic data types with GLISP object descriptions, so that standard properties of these data types are available in a dialect-independent manner. The compiler performs type inference for Lisp system functions, so that the types of their results will be known without requiring explicit declarations.

GLISP encourages the development of abstract data-type packages that mediate the interaction between user programs and idiosyncratic system features. For example, the GEV system uses Window/Menu data-type packages that allow the same code to run on Lisp machines with bitmap displays and on time-sharing systems with ordinary terminals.

The compiler performs symbolic simplification of the Lisp code it generates; this improves efficiency and allows the user to use the representational power of GLISP without paying a run-time penalty. Particular attention is paid to optimization of set operations and loops over sets to avoid unnecessary construction of intermediate sets. Symbolic simplification also provides conditional compilation in a clean form. The user may declare to the compiler that certain data have values that are considered to be compile-time constants. Compile-time execution of operations on constants can produce constant values for conditional tests; symbolic simplification of the resulting code causes unreachable program code to vanish. For large programs with many options, such as symbolic algebra packages, elimination of code for unwanted options can provide substantial savings in program space and execution time without changes to the original source code.

7. GEV: Knowledge-based Data Inspection

GEV (for GLISP Edit Value) is a program, written in GLISP, that interprets Lisp data according to its GLISP data-type descriptions and displays it in readable form in a window. The display contains three sections: the edit path that led to the current object, the data that are actually stored in the object, and computed properties of the object. Figure 7-1 shows an example. The user can "zoom in" on an item of interest, which will be displayed in greater detail according to its type description; this allows the user to browse quickly through a semantic network of related data. A data-type description can specify that certain computed properties should

```
GEV Structure Editor Window
HPP           ~ HPP
CONTRACTS     ~ (Advanced A.I. Archit- ...)
#4            ~ GLISP
LEADER        ~ GSN
-------------------------------------------
NAME            Gordon S. Novak Jr.
INITIALS        GSN
TITLE           VISITOR
PROJECT       ~ HPP
SALARY          30000.0
SSNO            455827977
BIRTHDATE     ~ July 21, 1947
PHONE         ~ (415) 497-4532
OFFICE        ~ MJH 244
HOME-ADDRE-   ~ Palo Alto, CA
HOME-PHONE    ~ (415) 493-5807
-------------------------------------------
CONTRACTS     ~ (GLISP)
AGE             35
MONTHLY-SA-     2500.0

QUIT     POP     EDIT     PROGRAM
PROP     ADJ     ISA      MSG
```

Figure 7-1: A GEV window display.

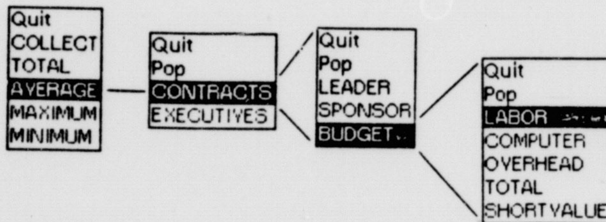
be displayed automatically whenever an object of that type is displayed; other computed properties can be requested by menu selection. The SHORTVALUE property of an object is used to display the object when "seen from afar"; for example, the SHORTVALUE for a PERSON object could be defined to be the person's initials. A tilde ("~") indicates that a SHORTVALUE is displayed rather than the actual Lisp value.

GEV allows the user to write looping programs interactively by menu selection. When the "program" command is selected, GEV first displays a menu of operations that can be performed. Next, a menu of possible sets over which the program could iterate is displayed. Finally, successive menus are presented to allow the desired property of the object to be selected; this process terminates when the user gives a "done" command or when a terminal value is reached. From these selections, a GLISP program to perform the specified operations is written, compiled, and run; this process normally takes less than a second. The result is printed and added to the GEV window; Figure 7-2 illustrates this process. The "program" feature allows the user to write significant programs rapidly without knowing the format of the data and without knowing any programming language. Since GEV interprets data according to GLISP object descriptions, it can be used for inspection of any Lisp data for which such descriptions are supplied.

```
GEV Structure Editor Window
HPP            ~ HPP

TITLE          ~ Heuristic Programming Proj-
ABBREVIATI-      HPP
ADMINISTRA-    ~ TCR
CONTRACTS      ~ (Advanced A.I. Archit- ...)
EXECUTIVES     ~ (EAF MRG GSN TCR)

BUDGET           659307.2
AVERAGE LA-      54000.28
```




```
AVERAGE BUDGET LABOR OF HPP CONTRACTS = 54000.28
```

Figure 7-2: Menu programming in GEV.

8. Summary

GLISP is an integrated programming system that uses declarative knowledge of the implementations of objects to generate code for operations on the objects. Recursive compilation relative to object types provides code efficiency comparable to ordinary Lisp with the representational power of object-centered programming. The GEV system interprets GLISP object descriptions to provide intelligent inspection and editing of data and menu-driven interactive program generation.

GLISP and GEV are fully implemented and are being used by a number of university and industrial research labs for implementation of AI systems.

9. How to Obtain GLISP

GLISP and GEV are available without charge over the ARPANET. GLISP files are stored in the directory <GLISP> on the host computer SUMEX-AIM.[3] At the time of writing, GLISP is available for Interlisp, Maclisp, Franz Lisp, UCI Lisp, ELISP, and Portable Standard Lisp; Zetalisp and Common Lisp are planned. The manual is available as `GLUSER.MSS` (Scribe source form) and `GLUSER.LPT`, and it tells how to obtain the files for the different Lisp dialects. The file `GLISP.NEWS` contains news on recent developments.

---

[3] The login "anonymous guest" may be used for FTP transfers.

References

1. Bobrow, D. G., and Stefik, M. The LOOPS Manual. Tech. Rept. KB-VLSI-81-13, Xerox Palo Alto Research Center, 1981.

2. Borning, A., and Ingalls, D. A Type Declaration and Inference System for Smalltalk. *Proc. 9th Conf. on Principles of Programming Languages*, ACM, 1982.

3. Cannon, H. I. Flavors: A Non-Hierarchical Approach to Object-Oriented Programming. Tech. Rept. Working Paper, A.I. Lab, Massachusetts Institute of Technology, October, 1981.

4. Ingalls, D. The Smalltalk-76 Programming System: Design and Implementation. *5th ACM Symposium on Principles of Programming Languages*, 1978.

5. Novak, G. S. GLISP Reference Manual. Tech. Rept. HPP-82-1, Heuristic Programming Project, Computer Science Dept., Stanford University, February, 1983.

6. Novak, G. S. GLISP: A High-Level Language for A.I. Programming. *Proc. 2nd National Conference on Artificial Intelligence*, Carnegie-Mellon University, 1982.

7. Novak, G. S. "GLISP: A Lisp-based Programming System with Data Abstraction." *A.I. Magazine 4*, 3 (August 1983).

8. Waters, Richard C. "The Programmer's Apprentice: Knowledge Based Program Editing." *IEEE Transactions on Software Engineering SE-8*, 1 (January 1982).

Copyright © 1985 by KSL and
Comtex Scientific Corporation

FILMED FROM BEST AVAILABLE COPY