NEGOTIATED INTERFACES FOR SOFTWARE REUSE

GORDON S. NOVAK, JR., FREDRICK N. HILL,
MAN-LEE WAN & BRIAN C. SAYRS

FEBRUARY 1992 AI 92-169
Negotiated Interfaces for Software Reuse

Gordon S. Novak Jr., Fredrick N. Hill, Man-Lee Wan, and Brian C. Sayrs *
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

March 21, 1991

Abstract

A significant barrier to the reuse of software is the rigid interface presented by a subroutine. For nontrivial data structures, it is unlikely that the existing form of the data of an application will match the requirements of a separately written subroutine. We describe two methods of interfacing existing data to a subroutine: generation of a program to convert the data to the form needed by the subroutine and rewriting the subroutine, through compilation, to fit the existing data. Both methods can be invoked through easily used menu-based negotiation with the user. These methods have been implemented using the GLISP language and compiler.

1 Introduction

Much of the work of programming consists of rewriting standard algorithms for use in the particular application of interest. Reuse of previously written subroutines that implement the desired algorithms has the potential to reduce the cost of software, increase the speed of software production, and increase reliability. However, there presently is relatively little reuse of software in practice.

A significant barrier to the reuse of software is the rigid interface presented by a subroutine. The actual arguments presented in a subroutine call must exactly match, in number, order, and data type, the formal arguments in the subroutine's definition; other properties, such as the units of measurement in which numeric values are expressed, must match as

*This research was supported by the U.S. Army Research Office under contract DAAG29-84-K-0060. Computer equipment used in this research was donated by Xerox Corporation and Hewlett Packard. Authors' present addresses: Fredrick N. Hill, JamSync Engineering, 413 Tulane SE, Albuquerque NM 87106; Man-Lee Wan, Hewlett Packard, Fort Collins, CO; Brian C. Sayrs, Lockheed R&D, 3251 Hanover St., Palo Alto, CA 94304.
well. Strongly typed languages enforce some of these requirements explicitly; however, the requirements must be met in most programming languages whether they are enforced by the language or not; failure to meet the requirements will result in runtime errors or wrong answers. Because of these requirements, most reuse of software occurs where compatibility of arguments occurs naturally because the argument types are simple (e.g., sqrt requires only a single floating-point number) or are types whose form is made compatible by the language (such as arrays).

For nontrivial data structures, it is unlikely that the existing form of the data of an application will match the requirements of a separately written subroutine. The application data may use a different record format than the subroutine. The application data may lack some fields that are present in the existing definition of an object type (for example, a circle object might have been defined with a center data field, but it is possible to ask for some properties, e.g. the area, without the center being specified); the application data may also have some extra fields. The application data may be specified in a different form (e.g., a circle could be specified by its diameter rather than its radius) or using different units of measurement. If the application data fail to match exactly the arguments required by a subroutine, use of the subroutine normally requires that the data be converted into the desired form prior to the subroutine call, and possibly that the results be converted also (Figure 1), or that the subroutine be rewritten to work with the existing data.

In traditional programming, the complexity of the data conversion required to use a previously written subroutine discourages reuse. Data conversion requires additional handwritten code, detracting from the benefit gained by reuse. The data conversion process requires time to run, space for its code, and space to make new copies of the data (a serious disadvantage for large data sets).

We describe two methods for semi-automatic interfacing of application data to an existing subroutine. In the first method, data conversion is performed as described above, but the conversion program is automatically generated from a specification of how the application data match the data needed by the subroutine; this specification can be produced from an easily used menu-based negotiation with the user. In the second method, a connection type is produced (also from menu-based negotiation) that describes how the application data match the abstract data used by a generic subroutine; compilation of the generic routine through the connection type produces a specialized version of the algorithm that operates directly on the application data. Both methods make it relatively easy to reuse a subroutine for application data whose form is different from the form for which the subroutine was originally written.
2 Related Work

The two volumes edited by Biggerstaff and Perlis [2] contain numerous papers on software reusability. [3] contains papers on reuse in Ada. [1] and [12] are good selections of papers on A.I. techniques used in software engineering. [13] describes the Programmer’s Apprentice project at MIT; the clichés of the Programmer’s Apprentice are somewhat analogous to our generic routines.

2.1 IDL

The IDL (Interface Description Language) system [6] was designed to allow exchange of large structured data between separately written components of a large software system, such as an Ada compiler. IDL accepts the output of one phase of the compiler, reformats it into a “flat” form that can be written to an external file, and allows such a file to be read into a (possibly different) data structure for the following phase. The data structures involved may be large graph structures, so IDL allows for structure sharing. Use of IDL requires that precise textual specifications of the data formats be written by the programmer.

2.2 Object-oriented Programming

Object-oriented programming (OOP) has become popular and has been promoted as a way of achieving software reuse. To some extent OOP does foster reuse, but there are significant problems with OOP as a mechanism for reuse. One class of problems involves the message interface to objects. Not only must an object accept all the messages that will be sent to it, but the results returned must be objects that will accept all the messages that may be sent to them, and so on; typically, these requirements are not automatically checked. Thus, the type requirements of an object-oriented interface are often as rigid as those of an ordinary subroutine. Further, if reuse is accomplished by inheritance of methods from generic objects, there may be name conflicts. For example, a pipe might have an inside-radius and an outside-radius, so that it could not directly inherit methods from circle that expect a single radius property: while radius could be aliased to either inside-radius or outside-radius, it could not be aliased to both.

A second class of problems with OOP involves efficiency. Since most messages perform actions that are small (often only accessing data), the overhead of message interpretation is large relative to the cost of the operation. When systems are layered, the overhead of message interpretation is repeated many times. Another problem is that the opacity of objects prevents optimization: if multiple operations are accomplished by sending messages to an object, it will not be possible to optimize the operations by combining them. Materialization of objects that are used only temporarily also has a high cost. These efficiency problems inhibit reuse.

The techniques we describe provide the conceptual benefits of OOP while often avoiding
these problems.

3 GLISP Language and Compiler

GLISP [7, 8, 9, 10] is a high-level language with abstract data types that is compiled into Lisp. It allows the actual data structure of an object to be described, so that the programmer has control of the structure and representation of data. Many features of GLISP are easily understood as extensions to object-oriented programming. Associated with each object are computed properties and message operations that can be performed on it. Computed properties are similar to messages, but generally do not have side effects; the syntax of references to computed properties and stored data is the same, so that code does not depend on whether a given piece of data is stored or computed. Objects can inherit properties and messages from parent classes. Since generic objects and operations on them can be described separately from applications, generic programs can be written in terms of objects and operations on objects rather than in terms of the implementations of the objects.

GLISP supports run-time message sending; it also offers important extensions to object-oriented programming. The GLISP compiler keeps track of the types of objects at compile time and uses type inference to infer the types of derived objects. When the type of an object is known, the interpretation of a message to the object can often be determined at compile time, allowing elimination of the overhead of message interpretation at runtime. The message can be compiled as a direct call to a predefined function, or a specialized version of a generic function can be compiled, or the function can be open compiled, in effect macro-expanded in place. This expansion is recursive at compile time and, unlike most macro-expansion processes, maintains knowledge of object types. Recursive compilation allows a simple operation in source code to undergo multiple levels of expansion, with parts of the expansion being drawn from different object classes, depending on the types of the objects involved as expansion proceeds. Symbolic optimization removes unnecessary code and combines operations where appropriate, so that the resulting code is relatively efficient. Code inversion automatically inverts algebraic expressions and extraction of data from structures, allowing results to be stored back “through messages” when appropriate. These features are illustrated with examples in the following sections.

3.1 Data Structures

Data structures are a serious barrier to reuse in most languages. Often, the form of code is different depending on the type of data structure used or whether data is stored or computed; when this is the case, it prevents reuse of code for alternative implementations.

GLISP has a data description language that allows the actual data representation that is desired to be described; this language is sufficient for describing most data structures used in Lisp programs and has been extended to allow generic data structures (e.g. tuples and sets) to be described.
The following examples illustrate how GLISP allows the same code to be used with data that are represented in different ways. Two variants of circle are defined: the first stores the radius, while the second stores diameter using a different record structure.

(circle (list (center vector) (radius real))) ; the data structure
prop ((area (pi * radius ^ 2))) ; a computed property

dcircle (cons (diameter real) (color symbol)) ; different data
prop ((radius (diameter / 2))) ; radius is computed
 supers (circle)) ; inherits from circle

The syntax of GLISP allows stored and computed properties of objects to be referenced in the same manner; the following examples show source code and compiled code for functions that compute the areas of circles of different types:

(gldefun t1 (c:circle) (area c)) (LAMBDA (C)
 (* PI (EXPT (CADR C) 2)))

(gldefun t2 (c:dcircle) (area c)) (LAMBDA (C)
 (* 0.7853982 (EXPT (CAR C) 2)))

In the second example, the definition of area is inherited from circle and expanded; the reference to the radius is then expanded to be diameter / 2, and diameter is expanded as a data structure reference. Symbolic optimization folds the constants [14] in the resulting code.

In order to reuse code, it must be possible not only to "read" data that are computed rather than stored, but also to "store into" data that are not represented directly; this is done by automatic algebraic inversion for computed properties.

(gldefun t3 (c:dcircle v:real) ((area c) := v) (LAMBDA (C V)
 (RPLACA C (* 1.128379 (SQRT V)))
 C)

In this example, the computed property area appears on the left-hand side of an assignment; this property is expanded by the compiler, resulting in an expression similar to that shown in example t2 above. Algebraic inversion of this code is performed by inverting the outermost operator of the left-hand side and moving it to the right-hand side using algebraic laws. For example, the expression

1The syntax of function definitions is (gldefun <function-name> (<arguments>) <code>); gldefun stands for "GLisp DEfine FUNction". In actual code, the character : is written with a back-slash escape character for compatibility with Common Lisp; our examples omit the back-slash for readability.

2(EXPT x 2) is the way of writing x^2 in Lisp. CADR and CAR are references to the radius and diameter parts of the data structures.

3We find that properties into which one would want to "store" can usually be inverted automatically. When algebraic inversion is not possible or is ambiguous, a compiler error message results.
would become:

\[
(\text{EXPT (CAR C) 2}) \equiv (/ \text{V 0.7853982})
\]

and then:

\[
(\text{CAR C}) \equiv (\text{SQRT (/ \text{V 0.7853982})})
\]

Operator inversion is done repeatedly until a data structure reference is reached on the left-hand side; this reference is then changed into a store of the value resulting from the inversion.\(^4\) Thus, the expression \((\text{area c} := \text{v})\) is compiled so that it stores into the appropriate stored data of \(c\) a value that will cause its (computed) property \text{area} to have the value \text{v}. These examples illustrate features of the GLISP language and compiler that are used to implement the reuse capabilities described below.

4 Reformatting Data

A typical way to interface existing data to a subroutine is to write a data conversion program to reformat the data into the form required by the subroutine. Unfortunately, this method normally requires that a human programmer understand both the form of the existing data and the form required by the subroutine, and then write code to reformat the data.

Man-Lee Wan has written a program, LINK [15], that automatically writes the interface conversion program. It is assumed that the existing data have been described using the GLISP structure description language. Likewise, the subroutine is assumed to have a specification of the data that it requires. From these two descriptions, LINK conducts a menu-based negotiation to determine how the needs of the subroutine can be satisfied by the existing data. For example, a subroutine that draws a pie chart might expect as input a list of pairs, where each pair consists of a title for a slice of the pie and a numeric value representing the size of the slice:

\[
(\text{listof (list (title symbol) (value number))})
\]

An example of actual data in the form to be presented to the program is:

\[
((\text{COOKIES 27200.0}) \; (\text{CANDY 84400.0}) \; (\text{VEGETABLES 2720.0}))
\]

\(^4\)In this case, the data access operation \text{CAR} is changed into the corresponding store operation, \text{RPLACA}. The final compiled code shown for example t3 has been optimized by taking the numeric constant outside the \text{sqrt}. 
Specify parts of ITEM of type (LIST (TITLE SYMBOL) (VALUE NUMBER))

Figure 2: Interface Specification by Menu Selection

In order to satisfy the listof specification, the LINK program assumes that the application data contains a set of similar items and that each item contains data that can be used to derive the corresponding data within the listof item, i.e. title and value. For example, if the application data represented divisions of a company with various data about each division, the user could specify a pie chart of the divisions with the title of each pie slice being the name of the division and the value being net profit of the division (which might be a computed value).

The task of the LINK program is to match the specifications of the required data and the available application data, then create a program (in GLISP) to derive the data values, make the data structures, and call the subroutine.

Derivation of the correspondence between the specification and the available data is done recursively. When the specification requires a set of items (in this case, a listof items), semantic filtering is applied to the description of the available data to determine which stored or computed features of the available data can produce a set of items; a menu of such features is presented to the user for selection. The user can optionally specify a predicate to determine which members of the input set are to be included; this can be a predefined, named predicate associated with the member objects, or it can be an arbitrary predicate specified as an expression by the user through successive operator and operand menus. For example, to select items with age less than 30, the user could select < as an operator, select age from an operand menu (whose items are names of the stored and computed data of the item type that could produce a numeric value), and manually type in the value 30.

An example of this process is shown in Figure 2. The first menu is presented to the user,
in response to the listof specification, to allow choice of the set of items from which to construct the input; the user selects DIVISIONS. Within a listof specification, the parts of each item are assumed to be derivable from the corresponding data item of the available data set. In our example, the title and value should be derivable from the data for a division of the company. A menu for each data item is presented to the the user for selection; the menu options are obtained from the original data items and computed properties that can result in the specified type. In the figure, the user selects DIVISION-NAME as the TITLE of a pie slice. When the application data have multiple levels of structure, successive menus are presented to descend through the data types until the desired data is reached. In this example, the desired data NET-PROFIT is reached by first selecting FINANCIALS from the data for a DIVISION, then NET-PROFIT within the menu for FINANCIALS. Arithmetic expressions can be specified by menu when desired, using the data available with the type as operands.

This sequence of menu selections results in the following GLISP program:

\[
\text{(LAMBDA \text{(COMPANY9200:COMPANY)})} \\
\text{(FOR DIVISION9202 IN \text{(DIVISIONS COMPANY9200)})} \\
\text{COLLECT \text{(A \text{(LIST \text(TITLE SYMBOL)})}}} \\
\begin{align*}
\text{(VALUE NUMBER))} \\
\text{TITLE \text{(DIVISION-NAME DIVISION9202)}} \\
\text{VALUE \text{(NET-PROFIT \text{(FINANCIALS DIVISION9202)})})})
\end{align*}
\]

which is automatically compiled into the considerably less perspicuous Lisp code:

\[
\text{(LAMBDA \text{(COMPANY9070)})} \\
\text{(MAPCAR}} \\
\begin{align*}
\#'(\text{(LAMBDA \text{(DIVISION9111)})} \\
\text{(LIST \text{(CAR DIVISION9111)})} \\
\text{(LET ((SELF \text{(CADDR DIVISION9111)})}} \\
\text{(- \text{(CAR SELF)})} \\
\text{(+ \text{(CAR SELF)})} \\
\text{(+ \text{(CADDR SELF)})} \\
\text{(* \text{0.28 (CADDR SELF)})}) \\
\text{(CADDR COMPANY9070))})
\end{align*}
\]

Specification of the interface by menus takes less than 10 seconds, limited mainly by the reaction time of the user; compilation of the resulting program takes about 0.1 second. It is clear that even a skilled programmer would take much longer to produce such an interface program manually.

We can describe the process of data translation more formally as follows: given a source type $S$ and a goal type $G$, we wish to produce a function $f : S \rightarrow G$. In most cases, the function $f$ will simply transfer data fields or predefined properties of $S$ into the corresponding data fields of $G$. Acquisition of the transfer function $f$ can be done by recursion on the source type $S$, source code $s$, and goal type $G$ using the following cases on the type $G$:
1. $G$ is a simple type (integer, symbol, etc.):
   Obtain the function $f : S \rightarrow G$, where $f$ is:
   
   (a) Access to stored data of type $G$ within $S$.
   (b) A computed function, $f : S \rightarrow G$, that is predefined for the type $S$.
   (c) Constant data of type $G$, entered by the user.
   (d) If the user selects stored data or a predefined computed function of type $T$, this
       provides a function $f_1 : S \rightarrow T$. Recursively acquire a function $f_2 : T \rightarrow G$ and
       return $f = f_2 \circ f_1$.
   (e) If the user selects $\langle op \rangle$, make an expression $f_1 < op > f_2$ of type $G$ whose operator
       $\langle op \rangle$ is selected from an operator menu and whose operand(s) are recursively
       acquired as $f_1 : S \rightarrow G$ and $f_2 : S \rightarrow G$.

2. $G$ is a record type, $r((n_1, g_1), \ldots, (n_k, g_k))$,
   where $r$ is the record type and its components are named $n_1$, with type $g_1$, through
   $n_k$, with type $g_k$. Collect functions $f_1$ through $f_k$, where $f_i : S \rightarrow g_i$. Return code to
   create record $r$ from $f_1(s), \ldots, f_k(s)$, (a $r$ with $n_1 = f_1(s) \ldots n_k = f_k(s)$).

3. $G$ is a sequence, e.g. (listof $g$), where $g$ is a type.
   Obtain a function $f : S \rightarrow \text{sequence}(T)$ where $T$ is the type of an item in the sequence,
   a function $f_e : T \rightarrow g$ that maps the element type $T$ of the sequence to the goal element
   type $g$, and optionally a predicate $p_e : T \rightarrow \text{boolean}$ that selects the elements of the
   sequence that are to be included. Return the code
   (for $i$ in $f(s)$ when $p_e(i)$ collect $f_e(i)$).

4. $G$ is a named, non-simple type:
   
   (a) If there is stored data or a computed function of type $G$ that is defined for type
       $S$, the user may select it as described above.
   (b) If there is a known function $f : S \rightarrow G$ for translating from $S$ to $G$, including the
       function presently being developed, it can be selected by the user.\footnote{Allowing recursive use of the function that is being developed permits translation of recursive data types such as trees.}
   (c) If the user selects the option Parts, acquire the function $f : S \rightarrow \text{rep}(G)$, where
       $\text{rep}(G)$ is the data representation of type $G$, as described in case 2 above.

From these correspondences, LINK produces an interface program (in GLISP) that filters
the given data set and produces a new data set in the proper format, then calls the subroutine
with the converted data set. Figure 3 shows this process schematically. From the user's
perspective, the existing subroutine has been converted into a version that works directly
with the user's data in its original form (dotted box).

Use of such a program to produce interfaces has several advantages:
Figure 3: Interfacing existing data to a subroutine using LINK
1. It is fast: an interface such as the pie chart example described above can be created in seconds.

2. The user does not have to understand the interface to the subroutine, beyond being able to match names appearing in the interface description to corresponding data. For some programs and data sets, a non-programmer could create the interface and use the program without assistance.

3. Type checking is performed – not by giving error messages for type mismatches, but by presenting only correct types as choices.

4. Reusing programs by copying and modifying the data is an efficient method when the data sets involved are relatively small.

5. This technique could also be used to produce data in the form needed to use a server program that is accessed remotely over a network.

Herlihy and Liskov [4] describe a method for transmission of structured data over a network, with a possibly different representation at the destination. Their method requires that the programmer write procedures to encode and decode the data into transmissible representations; they also describe a method for transmission of shared structures. The LINK program does not handle data involving shared structures, although it could presumably be extended to do so in a manner similar to that of [4].

Margaret Reed-Lade has written a program [11] that can semi-automatically construct a grammar and can use the grammar to parse data, represented as tabular or running text, in the form input from an optical scanner. This program also produces a GLISP structure description for the resulting data, allowing the data to be fed into application programs using the LINK program. The combination of these two programs provides easy connectivity from printed data to an application using that data.

5 Compilation of Generic Programs

The GLISP compiler is able to specialize a generic routine, written in terms of abstract data types, so that it operates directly on application data whose types are instances of the abstract types. This is accomplished in the following way. When the generic routine is compiled, the actual types of the arguments replace the generic types that appear in the original definition of the routine. References to parts or features of these arguments give results whose types are derived through type inference, and these types are propagated during compilation. Operations on objects will in general be type-dependent, so that the same generic code can expand into quite different output code when used for objects of different types.

As an example, consider a generic function for addition of two-element \((x, y)\) vectors:
(gldefun vectorplus (u:vector v:vector)
    (a (typeof u) with x = (x u) + (x v)
        y = (y u) + (y v))
)

This function specifies the creation (using the (a ... ) construct) of a new vector whose type, (typeof u), is the same as the type of the first argument, u, and whose x and y components are the sums of the x and y components of the arguments. Specialization allows this generic function to be used for several different kinds of vectors, such as the following:

(vector (list (x integer) (y integer))
    msg ((+ vectorplus open t)))

(svvector (cons (y string) (x string))
    supers (vector))

(vofv (list (x vector) (y vector))
    supers (vector))

vector is a list of two integers; svvector is a vector of two strings, stored in a cons rather than a list; and vofv is a vector of vectors, whose components are vectors of integers. The operator + is overloaded as the function vectorplus, to be compiled open, in the specification of vector; svvector and vofv list vector as a superclass, so they inherit this definition. Given the above definitions, code that specifies addition of two vectors (shown in lower-case) is compiled into the corresponding versions shown in upper-case:

(gldefun t4 (u:vector v:vector) (u + v))
(LAMBDA (U V)
    (LIST (+ (CAR U) (CAR V))
        (+ (CADR U) (CADR V))))

(gldefun t5 (u:svvector v:svvector) (u + v))
(LAMBDA (U V)
    (CONS (CONCATENATE 'STRING (CAR U) (CAR V))
        (CONCATENATE 'STRING (CDR U) (CDR V))))

(gldefun t6 (u:vofv v:vofv) (u + v))
(LAMBDA (U V)
    (LIST
        (LET ((G63 (CAR U)) (G64 (CAR V)))
            (LIST (+ (CAR G63) (CAR G64))
                (+ (CADR G63) (CADR G64)))))
        (LET ((G65 (CADR U)) (G66 (CADR V)))
            (LIST (+ (CAR G65) (CAR G66))
                (+ (CADR G65) (CADR G66)))))

12
Example t5 shows how the + operator is interpreted as concatenation for the strings that are the vector elements. Example t6 shows how the generic function vectorplus is open-compiled three times (once for the vofv and once for each of its vector components) to perform a single addition of two vofv data.

5.1 Connection Types

The above examples illustrate the compilation of generic routines for data that are instances of an abstract type, i.e., that have the abstract type as a superclass and that have all of the data elements needed by the superclass as stored data or computed properties, with the same names used in the superclass. In general, these requirements will not be satisfied by application data:

1. Application data may use different names for data components than are used in the abstract type.

2. Application data may have components that are equivalent, but not identical, to those used in the abstract type, e.g. use of diameter rather than radius to specify the size of a circle.

3. There may be multiple ways of viewing application data as being of a given abstract type. For example, a pipe could be viewed as a circle using either its inside-radius or outside-radius as the radius of the circle; both views might be useful, so it would not be adequate simply to alias the name radius to one of them.

4. Application data may have names that conflict with names in the abstract type.

Any of these problems would prevent generic routines from being specialized for application data by direct inheritance, as was done in the examples of the previous section.

A connection type can be used to view application data as an instance of an abstract type. A connection type has the application data as its stored form, the abstract type as a superclass, and computed properties that express the data needed for the abstract type in terms of the data or properties available in the application data. Figure 4 illustrates how the orbit of a planet can be viewed as a circle through the connection type planet-orbit-as-circle. planet-orbit-as-circle has as its stored data a single item p of type planet, a computed property radius that is computed as the orbit-radius of p, and the superclass circle. Given these definitions, properties of the circle abstract type, such as its area, are available through the view:

(gldefun t7 (pc:planet-orbit-as-circle) (area pc))

(LAMBDA (PC)
  (* PI (EXPT (GET PC 'ORBIT-RADIUS) 2))))
This example illustrates how the connection type makes available the radius data needed for the circle abstract type, while hiding the conflicting name radius of the planet. Specialized versions of larger closed subroutines can be created by compiling generic routines through connection types. The resulting specialized subroutines operate directly on application data and are symbolically optimized, making the resulting code efficient.

5.2 Negotiated Interfaces for Connection Types

When the data structures involved are more complex than the examples described above, making connection types to allow a generic program to be specialized for an application becomes more difficult. Careful attention must be paid to type declarations in the connection type so that type inference will produce types at the right level of description; for example, in a tree structure the children property of the connection type must produce a set of items, each of whose type is the connection type, not the application type. Because of the difficulty of making correct connection types, automated assistance is needed. Fredrick Hill has written a program [5] called NI that allows a connection type to be constructed automatically from correspondences between application data and specifications of the abstract data required by a generic routine; these correspondences are acquired through an easily used menu-based negotiation process similar to that described earlier. Following the negotiation process, the generic algorithm is compiled using the connection type, which results in a version of the algorithm that is specialized for the application data.

An example used with Hill’s program is a generic program for drawing a diagram of a tree structure. Such a program would seldom be found in a program library because of the
wide variety of possible tree structures that might be used for applications.⁶

In the negotiation process for the tree-drawing program, the user specifies:

1. the application data type that is the node of the tree
2. how to tell whether a node is terminal
3. how to get the children of a node
4. what printed representation is desired for a node.

Semantic filtering is used to restrict the set of choices presented to the user so that only the most likely ones are presented. For example, the children of a node in a tree is expected to yield a set of items of the same type as the node; there will often be only one data item of this type. An “escape” option is provided that provides a full menu of possibilities, including less likely choices, since it is possible to get to some types from nearly any other type⁷.

The result of compiling the generic tree-drawing routine through the connection type is a specialized routine to draw the user’s tree in the desired fashion. A given tree can be drawn in various ways; Figure 5 shows a family tree drawn to show only the females (empty boxes represent unmarried males). Figure 6 shows an n-bit integer drawn as a tree; successors are obtained by dividing the bits in the binary representation of the integer into left and right halves until a single bit or a value of zero is reached. These examples illustrate how the same generic algorithm can be automatically recompiled to operate on quite different data. The technique of recompiling a program through a connection type is powerful: several pages of specialized Lisp code can be produced in a few seconds. In the case of the integer-as-a-tree shown in Figure 6, it took longer to check the tree by hand to see if it was correct than it took to produce the program.

Recompilation through connection types is more powerful than copying and translation of data to a different form because it can be used with algorithms that modify the structure of data. A simple example is an algorithm to insert a new record at the end of a queue, implemented as a linked list of records, by modifying the pointers. If this were done by copying the data into a different form, inserting the record, and copying back to the original form, it would be very inefficient. Compilation of a generic algorithm through connection

⁶Several Lisp systems feature tree-drawing packages, but they require data conversion to a specified form.
⁷For example, it is possible to get from a symbol to a number by finding the length of the symbol’s print name, but this is not considered a likely choice.
types allows the insertion to be done efficiently by direct manipulation of the pointers in the original data structures. The ability of GLISP programs to “store through” a message by code inversion is essential to this capability.

6 Conclusions

We have described two methods for achieving reuse of programs by interfacing application data to separately written subroutines. These methods improve on traditional programming practice because they allow interfaces to be created very quickly and require minimal understanding of the subroutine specifications by the user; in some cases they would be suitable for use by non-programmers. The methods improve on object-oriented programming by allowing use of programs for application objects that are not direct instances of generic objects, but can be viewed, in part, as such instances (perhaps in multiple ways); in addition, the compiled code is more efficient than equivalent code using runtime message interpretation.

The systems described here are already useful, but additional work remains to be done. The LINK program performs conversion between application data and subroutine, but does not convert the output of the subroutine. In simple cases, another conversion similar to what LINK already does would suffice, but in more complex cases it might be necessary to match the results with the original data in order to modify the original data. In its present form, LINK can translate data that are recursive, but does not handle data that share structure.

The NI program considers only a single main data type and a program that deals with that type. In general, it is necessary to deal with multiple types that are related. For example, a symbol table data structure involves the record that stores information about a symbol, as well as index tables that may be used to provide fast lookup of a symbol. Making connection types for such structures requires mechanisms for representing clusters of related types and referring to related types within a cluster for type inference; these will be described in a forthcoming paper.
References


Peregrinus Ltd., 1990.

Data Types”, ACM Trans. on Programming Languages and Systems, vol. 4,


Description Language”, Tech. Report CMU-CS-83 129, Computer Science


Types,” Proc. National Conference on Artificial Intelligence (AAAI-83),

on Programming Language Issues in Software Systems; ACM SIGPLAN
Notices, vol. 18, no. 3 (June 1983).


Structured Text” (thesis), Tech. Report AI-89-105, Artificial Intelligence
Laboratory, Univ. of Texas at Austin, 1989.

[12] Rich, C. and Waters, R., eds., Readings in Artificial Intelligence and


[14] Schaefer, Marvin, A Mathematical Theory of Global Program Optimization,

(thesis), Tech. Report AI-85-18, A.I. Lab, C.S. Dept., Univ. of Texas at
Austin, 1985.