Software Reuse through View Type Clusters

Gordon S. Novak Jr.
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Abstract

Reuse of software is inhibited by the dependence of the form of program code on the kind of data structures used. We describe methods by which generic programs can be specialized, by compilation through views, for use with a given data structure. A view describes how an actual data type can be viewed as an instance of an abstract type. Clusters of related views are needed for specialization of generic programs involving more than one data type. Descriptions of abstract data types allow view clusters to be constructed automatically or semi-automatically by reasoning about the properties of application types. These techniques, which have been implemented using the GLISP language and compiler, allow easy and rapid adaptation of generic procedures for an application by first viewing the application data types as the appropriate abstract types, then specializing the generic procedures by compilation through the view cluster.

1 Introduction

Much programming involves rewriting of standard algorithms for use in the application of interest. Reuse of software has the potential to reduce cost, increase the speed of software production, and increase reliability. Compared to its potential, however, there presently is relatively little reuse of software.

A significant barrier to the reuse of software is the rigid treatment of data types. The types of actual arguments to a procedure must exactly match the formal arguments in the procedure's definition; other properties, such as the units of measurement of numeric values, must also match. Strongly typed languages enforce some of these requirements, but the requirements must be met in any case to avoid incorrect results. Most reuse of software occurs where compatibility of types occurs naturally because the types are simple or have a form made compatible by the language (e.g., arrays).

For nontrivial data structures, it is unlikely that the data types of an application will match those of a separately written procedure. The application data may use a different record format, contain extra fields, lack some fields that are defined for an abstract type but not used by that particular procedure, use different names for fields, or specify data in an equivalent but different form. If the application data are not identical to the arguments required by a procedure, it will be necessary to make them match. The traditional ways of doing this, by converting the data into the expected form or converting the procedure to use the existing form, are difficult and discourage reuse since they require hand-written code.

We have previously described [17] two methods for semi-automatic interfacing of application data to an existing procedure. In the first method, a data conversion program is generated automatically from a specification of correspondences between the application data and the data required by the procedure; this specification is obtained through a menu-based negotiation with the user. In the second method, a connection type is produced (also using menu-based negotiation) that describes how the application data correspond to the abstract data used by a generic procedure. Compilation of the generic procedure through the connection type produces a specialized version that operates directly on the application data. Both methods make it relatively easy to reuse a procedure for application data whose type is different from the type for which the procedure was originally written.

These previously described methods deal with relatively simple individual data types. In this paper, we describe extensions to more complex data involving multiple related types. We also describe
methods for reasoning about the relationships between user data types and abstract types to create view clusters automatically or semi-automatically. Once the application types have been viewed as instances of the corresponding abstract types, the whole library of generic programs associated with the abstract types becomes available by automatic specialization.

The goal of this work is a programming system that allows easy and rapid reuse of generic algorithms for an application by first viewing the application data types as instances of corresponding abstract types, then creating specialized versions of the generic algorithms by compilation. In this way, a large knowledge base of algorithms becomes available for reuse at low cost.

2 Related Work

2.1 Software Reuse

Biggerstaff and Perlis [2] contains papers on theory and applications of software reuse. The Ada language was intended to foster reuse, but this goal has only partly been achieved [4]. Modula-2 allows reuse of code from external modules [9] [12]; typically, such modules implement stylized constructs such as List of <type>, where <type> can be specified.

Artificial Intelligence approaches to software engineering have included transformations from specifications to code and reuse of code fragments or specifications; [1], [14], and [18] contain papers describing these approaches. The Programmer's Apprentice [19] was based on reuse of programming clichés.

Functional languages such as ML [23] allow polymorphic functions to be defined; it is not clear that such languages are presently suitable for general applications use.

2.2 Specification and Transformation

The SETL language [3] allows programs to specify operations on sets without specifying the data implementation; some work has been done [21] on automatic selection of data representations for improved efficiency. Gries [6] has proposed the incorporation of transformations as a programming language construct in the POLYA language. The KIDS and DTRE systems developed at Kestrel Institute [22] allow programs to be developed by interactive application of refinement transformations to formal program and data type specifications. Goguen [5] has proposed the use of views to describe connections between theories in the OBJ language. Algebraic specification languages such as LARCH [7] allow formal specification of procedure interface signatures and properties.

2.3 Data Translation

IDL (Interface Description Language) [11] was designed to allow exchange of large structured data, possibly with structure sharing, between separately written components of a large software system; use of IDL requires precise specification of source and target data formats. Herlihy and Liskov [8] describe a method for transmission of structured data over a network, with a possibly different representation at the destination, using procedures to encode and decode the data into transmissible representations; they also describe a method for transmission of shared structures.

2.4 Object-oriented Programming

Object-oriented programming (OOP) has been promoted as a way of achieving software reuse, but there are problems with reuse in OOP. 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. Messages may have unexpected side-effects. Thus, the type requirements of an object-oriented interface may be as rigid as those of an ordinary subroutine. If reuse is accomplished by inheritance of methods from generic objects, there may be name conflicts.

A second problem with OOP is inefficiency. Since most messages perform small actions, the overhead of message interpretation is large, especially in layered systems. Opacity of objects prevents optimization: if multiple operations are done by messages to objects, it will not be possible to optimize the operations by combining them. Materialization of temporary objects is costly. These problems inhibit reuse.

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

3 GLISP Language and Compiler

GLISP [15, 16, 17] is a high-level language with abstract data types that is compiled into Lisp. It allows the data structure of an object to be specified, 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; a GLISP type is analogous to a class in OOP. Associated with each type are message-like operations; computed properties are side-effect-free operations involving only the object itself. As in OOP, there is a hierarchy of types; messages and properties can be inherited from parent types.

GLISP supports run-time message sending; however, it is desirable to compile in-line code rather than interpreted message calls. The compiler performs type inference while compiling expressions. When the type of an object is known at compile time, a message to the object can be compiled as a direct function call, or a specialized version of a generic function can be compiled, or the function can be expanded and compiled in-line. The type of the result of a message or property can be specified; this allows a particular view of an object to be selected. Symbolic optimization [20] folds operations on constants, removes unnecessary code, and combines operations where appropriate; this improves efficiency and provides conditional compilation by eliminating conditionals when the test can be evaluated at compile time.

3.1 Data Structures

Data representation is a barrier to reuse in most languages, since the form of program code differs for different data structures and for data that is computed rather than stored; this prevents reuse of code for alternative implementations of data. GLISP has a data description language that allows a data representation to be specified; this language can describe Lisp data structures and can be extended to describe data structures in other languages.

The same code is used with data that are represented in different ways [17]: a syntax similar to that of a Lisp function call can reference stored data, messages or computed properties, or an ordinary function:

( name object )

If name is the name of a field of the data structure of the type of object, a structure reference is generated; if name is the name of a computed property or message of the type of object or one of its superclasses, that definition is expanded (as a call to a specified function, by open compilation of a line of code or a function, or as a call to a specialization of a generic function). Thus, the same code might expand to a data structure reference, or an expression containing data references, or a function call, depending on the type of the object. Because the expansion process is recursive at compile time, a small amount of code can be expanded through several levels, resulting in a large amount of output code. In addition, it becomes possible to write higher-order programs that expand through multiple levels.

In order to reuse code, it must be possible not only to "read" data that is computed rather than stored, but also to "store into" computed data; this is done by automatic algebraic inversion or by converting structure accesses into stores.

3.2 Compilation of Generic Procedures

The GLISP compiler can specialize a generic procedure, 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. The generic procedure is compiled, with the actual types of the arguments replacing the types that appear in the definition of the procedure. References to data or computed properties of these arguments give results whose types are derived through type inference, and these types are propagated during compilation. Since operations on objects are type-dependent, the same generic code can expand into quite different output code when used for objects of different types.

3.3 View Types

Generic programs can be compiled for data that are direct instances of an abstract type, i.e., that have the abstract type as a superclass and that have all of the needed data elements 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: application data might use different names for data, have names that conflict, or have components that are equivalent (but not identical) to those in the abstract type; in addition, there might be multiple ways of viewing application data as being of a given abstract type. Any of these problems would prevent generic programs from being specialized for application data by direct inheritance.

A connection type or view 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. Given
these definitions, properties of the abstract type are available through the view. Specialized versions of closed subroutines can be created by compiling generic programs through connection types; such versions operate directly on application data and are symbolically optimized, making the resulting code efficient.

4 View Type Clusters

Typing mechanisms associated with existing programming languages generally treat data types individually and as a whole. This is not consistent with the ways that data are used in practice; in particular:

1. An application data structure often is a conglomeration of data fields that are used at different times for different purposes; code for a particular purpose may use only a few fields and ignore the rest.

2. A data structure may be closely related to other data structures. While some such relationships may be apparent (e.g., a pointer to a particular type), others may not be (e.g., the fact that an integer field is intended to be an index into a particular array).

3. Even relatively simple data may have multiple views, e.g., a data record that is part of a linked list has views both as an individual record and as a linked list.

In order to compile specialized versions of nontrivial generic procedures, we have found it necessary to have a mechanism for grouping related data types, procedures, constants, and facts; we call such groupings clusters, using the terminology of CLU [13].

4.1 A Simple Cluster: Linked-List

As a simple example, we consider the cluster linked-list. It might be supposed that this is a trivial data structure and that only a single data type is needed to represent it; indeed, authors have published software modules for handling linked lists containing a user-specified contents type [12]. However, if linked-list is truly to be generic, it must be usable for any legitimate implementation of a linked list, e.g.:

1. a record that contains a pointer to another record of similar type;
2. a linked list of disk blocks, in which each block contains the disk address of the next block, with an address of zero indicating the last block;
3. a linked list implemented within an array, where a link is an integer array index of the next item, with an illegal array index (such as -1) indicating the last item (the contents might be contained in other arrays with corresponding indexes);
4. a representation of a knight's tour on a chess board, in which the row and column of the next square occupied by the knight together constitute a link pointer.

These examples show that a careful specification is required for a generic linked-list.

4.1.1 Record-with-Pointer

Many languages provide a built-in pointer type. As the above examples illustrate, however, the notion of pointer needs to be generalized.

A pointer is a data structure whose value set includes at least one null value and some values that are not null. One null value is defined as the property null-value; the default null-value is nil (assuming Lisp as the target language). If a pointer value is not null, it denotes a data structure in a memory (the particular memory is usually implied as a part of the pointer specification; the default, of course, is the main memory of the computer). The message dereference to a pointer makes available the record it denotes and has the record type.

The null test defaults to a test for equality with the null-value.

4.1.2 Linked-List

A linked-list is a record-with-pointer in which the record contains a link that is an instance of the pointer type and additional fields that together constitute the contents. Figure 1 illustrates the abstract type linked-list and an application type that can be viewed as an instance of it.

In this example, the application type, box, has a field called next that points to the next element of the list and two data fields, color and size. A structure declaration for box and an instance of a box list are shown below.
linked-list:

- link
- contents

box:

- color
- size
- next

Figure 1: Abstract Linked List and an Instance

(box (list (color color)
       (size integer)
       (next (- box))) )

(setq mybox '(red 1 (green 2 (blue 3 nil))))

5 Making View Clusters

A human programmer would have no difficulty in understanding how box could be viewed as an instance of the abstract type linked-list. We have developed a program, called VIEWAS, that can automatically or semi-automatically make such views:

>(viewas 'linked-list 'box)

BOX-AS-LL

In this case, a view cluster BOX-AS-LL has been produced automatically by reasoning from a description of linked-list and the data type box. Once such a view has been made, all of the linked-list functions in the library become available for box by automatic specialization:

>(gldefun tl (b:box)
   (nreverse (linked-list b)))

>(tl '(red 1 (green 2 (blue 3 nil))))

The function tl specifies an nreverse (destructive reversal in place) of the linked-list view of its argument, a box. When this function is called, it is automatically compiled, and this results in compilation of a specialized version of the generic nreverse function.

Although nreverse is not difficult to write, it might well require a human programmer to stop, draw some diagrams of pointer structures, write a dozen or so lines of code, and try the code on some test cases. The reuse of the generic procedure by specialization through the view is clearly easier, faster, and less error-prone. The generic-nreverse function and specialized version are shown below.¹

(gldefun generic-nreverse
   (l:linked-list-pointer)
   (result (typeof l))
   (let (prev next)
      (prev := (null-value l))
      (while l is not null do
        (next := (rest l))
        ((rest l) := prev)
        (prev := l)
        (l := next))
      prev))

result type: BOX-AS-LL-POINTER
(LAMBDA (L)
   (LET (PREV NEXT)
      (SETQ PREV NIL)
      (TAGBODY GLLABEL1200
         (WHEN L
            (SETQ NEXT (CADDR L))
            (RPLACA (CDDR L) PREV)
            (SETQ PREV L)
            (SETQ L NEXT)
            (GO GLLABEL1200)))
      PREV))

The function generic-nreverse is written in such a way that it references only the abstract properties of a linked list—properties that must be defined in some way for any linked list, but that could be implemented in many different ways. In this example, the null-value, null test, and rest (value of the link field) are all defined relative to the implementation. The rest is both read and written in this function.

In some cases, there is more than one way to view an application type as an abstract type, or other choices need to be made. In such cases, VIEWAS prompts the user to make the choices:

>(viewas 'sorted-linked-list 'box)

Specify choice for SORT-VALUE
Choices are: (COLOR SIZE VOLUME WAVELENGTH)

1The character : is written with a back-slash escape character in Common Lisp. We omit this for readability.
Specify choice for SORT-DIRECTION
Choices are: (ASCENDING DESCENDING)

BOX-AS-SLL

In this case, the user has been prompted for the field to sort on (including computed values defined for the type box as well as stored values) and the direction of sorting; the user has selected an ascending sort on color. This sorting will use the comparison function defined for the type color, which will produce a sort in spectral order:

\[(\text{gldefun } t2 \text{(b:box new)} \text{(insert (sorted-linked-list b) new))})\]

\[(t2 \ ('\text{red} 1 \ ('\text{green} 2 \ ('\text{blue} 3 \ nil))'))\]

\[(\text{RED} 1 \ ('\text{YELLOW} 4 \ ('\text{GREEN} 2 \ ('\text{BLUE} 3 \ nil))))\]

5.1 Abstract Data Type Specification and Viewing Process

The specification of an abstract datatype that is used by VIEWAS has two main parts. The first part is a set of names and specifications that are to be matched against the application type; the second part is a pattern for the output cluster, which is instantiated by substitution of the values determined from the first part.

In the case of linked-list, the input to VIEWAS gives the name of the target cluster, linked-list, and the record structure that is to be matched as a linked-list element. The first item to be determined is the type of the pointer to this record. The pointer type defaults to a Lisp pointer to the record, but a different kind of pointer, such as an array index, will be used if it has been defined. The next item to be matched is the link, which points to the next record in the linked list. The link must have the pointer type; type filtering is used to restrict the possible matches to only those fields of the record that have this type. Often there is only one potential match, in which case it is selected automatically. The final element to be matched is the copy-contents-names, which consists of the names of all stored fields in the record other than the link; these names are used in copying the contents of a record, as described below.

Once the items required by the abstract data type specification have been matched in the application type, the results of the matching are substituted into a pattern to form the view type cluster used in specializing generic library procedures associated with the abstract type. At present, the library contains 28 procedures defined for linked-list. Thus, a single viewing of an application type as an abstract type provides access to a large library of procedures.

More complex abstract types require more items to be matched and may require selections to be made by the user. sorted-linked-list, for example, requires selection of data to sort on (which could be either stored or computed data) and a direction of sorting. The comparison predicate to be used is inherited from the type of the data that are compared; thus, a sort on names of type string will automatically use a string comparison. The generic procedures defined for sorted-linked-list explicitly test the value of sort-direction, which will be ascending or descending; since this value will be constant at compile time, the test will be eliminated and only the appropriate set of code for the selected direction will be kept. Thus, the inclusion of constant values in the view type cluster can be used to select optional features of generic procedures; conditional compilation occurs automatically because the symbolic optimizer eliminates tests that are constant at compile time and eliminates unreachable code.

5.2 Loop Macros

Programs frequently involve loops over collections of data. In writing generic procedures, there must be a way to write loops that does not depend on the kind of collection of data that might be used (e.g., array, linked list, or tree). We have implemented a set of loop macros that allow loops to be written in a structure-independent manner. For example, while length is provided as a generic library function for linked lists, the length function could also be written as a loop:

\[(\text{gldefun } t3 \text{(b:box)} \text{(for x in (linked-list b) sum 1))})\]

\[(t3 \ ('\text{red} 1 \ ('\text{green} 2 \ ('\text{blue} 3 \ nil))))\]

3

At compile time, the type of the collection argument of a looping statement is examined to see if a loop macro is defined for it; if so, the loop macro is expanded as in-line code. Macros are provided not only for simple loops, but also for operations such as summation, averaging, or gathering statistics. It may be necessary to collect data into a specified form; collection macros allow this, making it possible to write a loop that translates from one kind of collection to another (e.g., from an array to a linked list).
Data structures may involve multiple levels of structure of different types. For example, a symbol table might be constructed using an array of buckets, where the array is indexed by the first character of a symbol and the array element is a pointer to a bucket, i.e., a sorted linked list of symbols that begin with that character. A double iterator composes component iterators, allowing a loop over a compound structure:

```lisp
gldefun t4 (s:symbol-table)
     (for sym in s sum (name sym)))
```

This function produces a string that is the concatenation of the print names of all the symbols in the symbol table. The loop expands into a double loop (first a loop through the array of buckets, then a loop through the linked list of symbols within each bucket) and collects a result that is a string (since the name of a symbol is a string and the meaning of + is overloaded as concatenation for the string type, the sum collection macro produces a concatenated string as its result). The compiled code for the function t4 is 23 lines of Lisp.

### 5.3 Higher-Order Code

The linked-list abstract type has a contents that represents the contents of each linked-list element. In general, the contents may consist of multiple items of different types. This presents a problem for generic functions such as copy-list: it is necessary for copy-list to make a new list record and then to copy the contents fields into that record, but this may take multiple assignment statements. This is accomplished by a loop over the copy-contents-names defined in the view type; for each name in the copy-contents-names, a "function call" of that name on the target record is assigned the value of a "function call" of that name on the source record. Since the list of names is a constant at compile time, the loop is unrolled by the compiler and results in the needed sequence of assignment statements to transfer the contents from one record to another.

### 5.4 Output Languages other than Lisp

For a variety of reasons, Lisp is not widely used as an applications language. However, Lisp is similar to the tree form of intermediate code used by compilers; it is relatively easy to perform a set of source-to-source transformations on the Lisp output of our system to convert the result into another language.

We have implemented an initial version of a post-processor that allows specializations of generic algorithms to be created in C. This is done in the following way. A C record is defined as one of the possible record structures. Accesses to parts of a C record are compiled into Lisp code that can be post-processed into the appropriate C syntax. In addition, this Lisp code can be run within Lisp to create and access ersatz C records, so that the full power of the Lisp environment and our programming and data display tools can be used for rapid prototyping.

Conversion of the Lisp code to C occurs in several steps. A set of transformations is used to transform Lisp idioms into corresponding C idioms and to transform constructs that are legal in Lisp but not in C (e.g., returning a value from an if statement) into legal forms. These transformations are applied repeatedly until no further transformations apply. A second set of output transforms is used to transform the resulting code into C syntax. The final output may be substantially restructured from the Lisp code form.

As an example, the nreverse program previously described, when compiled for a C version of the BOX structure, called CBOX, results in the following C code:

```c
CBOX nreverse_9 (1)
CBOX 1;
  { CBOX prev, next;
     prev = 0;
     while ( 1 )
       {
         next = l->next;
         l->next = prev;
         prev = l;
         l = next;
       }
     return(prev);
  }
```

This C code is reasonably understandable and has no dependence on Lisp or the Lisp runtime system.

### 5.5 A Larger Example: Convex Hull

The convex hull of a set of points in the plane is defined as the smallest convex polygon that encloses all the points; a common analogy is to think of the points as nails driven into a board with a rubber band stretched around them: the convex hull is the shape assumed by the rubber band. Kant [lo] has studied highly qualified human subjects who were given the task of writing this algorithm. This is not easy: all subjects took considerable time; some failed to solve the problem or produced a solution that required excessive computation. Although convex
hull algorithms are described in textbooks and in the literature, getting an algorithm from such sources may also be difficult: details of the algorithm may be omitted in the published description, and a coded version of a published algorithm would require testing but still might contain bugs that were not uncovered by the test cases.

We have written a generic algorithm to find a convex hull; Figure 2 shows the execution of this algorithm on 10,000 approximately Gaussian-distributed random points. In an application, it is unlikely that the data would simply be points in the plane; more likely, the data would have other attributes. In order to find the convex hull in terms of the original data, using a prewritten convex hull algorithm, it would be necessary to make a new data set in the form required by this algorithm, find the convex hull, and then form a new version of the hull in terms of the original data. If there were a large number of points, this would require a large amount of storage and computation.

Specialization of generic algorithms is more efficient. As an example, we considered the task of finding the convex hull of a set of cities, each of which has a latitude and longitude, using a Mercator projection. Given a view cluster (in this case, 14 lines written by hand) to express this view, a specialized version of the convex hull algorithm (168 lines of code) was produced that operates directly on the original data (Figure 3):

6 Generic Graphical Editors

Once a view of an application type as an abstract type has been made, the compiler can create and cache small functions that interpretively compute the properties associated with the abstract type. These allow the application data to be treated as the abstract data using runtime message sending.

We have implemented direct-manipulation graphical editors for abstract types such as linked-list and array. Given a display method and editor for the contents of an element of a structure (which can easily be made using an interactive display maker), a generic program for displaying and editing the structured data can be used directly on the application data. Figure 4 shows how the box data can be displayed using the generic
linked-list editor. The user can perform operations such as moving forward or backward in the displayed list or excising a selected list element; the user can also zoom in on a selected element to display it in more detail or to edit the contents.

This technique for writing graphical editors allows a single editor program to handle all instances of an abstract type such as `linked-list`. It omits uninteresting detail (such as the actual contents of link fields) and displays the data in an easily understood form.

7 Conclusions

We have described methods for achieving reuse of generic procedures by using view clusters. These methods improve existing practice because they allow specialized versions of generic procedures to be created quickly and require only partial understanding of data types and procedure specifications by the user. The compiled code is efficient. These methods allow reuse of procedures for application objects that are not direct instances of generic objects, but can be viewed as such instances.

These techniques are knowledge-based software engineering in several senses. First, they are based on the reuse of programming knowledge stored in the form of generic algorithms. We envision a large library of generic algorithms that could quickly be adapted for applications. Second, the VIEWS program uses descriptions of abstract data types to reason about how application types can be viewed as instances of the abstract types. Third, the generic structure editors provide direct-manipulation editing of application data based on views.

Our techniques provide high payoff in amount of generated code relative to the size and complexity of input specifications. In addition, they require a relatively modest understanding of the details of procedures in the library in order to reuse them successfully.

Because of the data structure independence of generic programs, the techniques described here provide much latitude for restructuring programs to meet new requirements or to improve efficiency. Traditional languages reflect the kind of data structures in the form of the code that deals with those structures; this makes change of data structures a very costly process. In our system, the form of output code is derived from the definition of data structures; design decisions are stated in a single place and distributed by compilation, rather than being restated in many places by hand coding.

References


