Creation of Views for Reuse of Software with Different Data Representations

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

August 1, 1997

This article appears in *IEEE Transactions on Software Engineering*, vol. 21, no. 12, Dec. 1995, pp. 993-1005.

Abstract

Software reuse is inhibited by the many different ways in which equivalent data can be represented. We describe methods by which *views* can be constructed semi-automatically to describe how application data types correspond to the abstract types that are used in numerical generic algorithms. Given such views, specialized versions of the generic algorithms that operate directly on the application data can be produced by compilation. This enables reuse of the generic algorithms for an application with minimal effort. Graphical user interfaces allow views to be specified easily and rapidly. Algorithms are presented for deriving, by symbolic algebra, equations that relate the variables used in the application data to the variables needed for the generic algorithms. Arbitrary application data structures are allowed. Units of measurement are converted as needed. These techniques allow reuse of a single version of a generic algorithm for a variety of possible data representations and programming languages. These techniques can also be applied in data conversion and in object-oriented, functional, and transformational programming.

*G. S. Novak, Jr. is with the Department of Computer Sciences, University of Texas, Austin, TX 78712. An on-line demonstration of the software described in this paper is available on the World Wide Web via `http://www.cs.utexas.edu/users/novak` in the Automatic Programming Server demo, Mathematical View option. A workstation running X windows is required.

Index terms: view, program transformation, generic algorithm, software reuse, data conversion, visual programming, symbolic algebra, abstract data type.

1 Introduction

An algorithm should be like a mathematical theorem in the sense that once the algorithm has been developed, it should be reusable and should never have to be recoded manually. Like other engineering artifacts, however, algorithms that are used in an application must be adapted to fit the other parts of the application. Almost everyone is willing to reuse the standard algorithm for `sqrt`, but algorithms such as testing whether a point is inside a polygon are less likely to be reused. The argument and result types of `sqrt` match application types; however, there are many possible polygon representations, so an application is unlikely to use the same data types as a library program.

Strong typing is an important optimization: by checking types statically, a compiler can avoid generation of runtime type-checking code, thus providing type safety while saving time and storage. Unfortunately, rigidity of types inhibits reuse. Traditional ways of making application data match a procedure that is to be reused are costly and discourage reuse [24]. An effective method of reuse must minimize two costs:

1. Human cost: the time required by the programmer to find the program to be reused, to understand its documentation, and to adapt the reused program and/or application so that they fit.

2. Computational cost: the added cost of running a reused program, compared to a hand-written version.

This paper describes methods for reuse using *views* that describe how application data types correspond to the abstract types used in generic algorithms. Algorithms based on symbolic algebra create the views from correspondences that are specified via an easily used graphical interface. Given the views, a compiler produces efficient specialized versions of generic algorithms that operate directly on the application data. A single copy of a generic algorithm can be specialized for a variety of application data representations and for a variety of programming languages. The efficiency, encapsulation, and type safety of strong type checking are retained, while the barriers to reuse caused by type rigidity are eliminated. The techniques described here are an enabling technology that makes software reuse easy and practical.

The graphical interface is illustrated in Fig. 1. In this example, we assume that the user has existing data that describes a Christmas tree; the user would like to reuse a small program, which calculates the area of the side of a cone, on this data. A view of the type `xmas-tree` as a `cone` is made using the program `mkv` ("make view"):

```
(mkv 'cone 'xmas-tree)
```

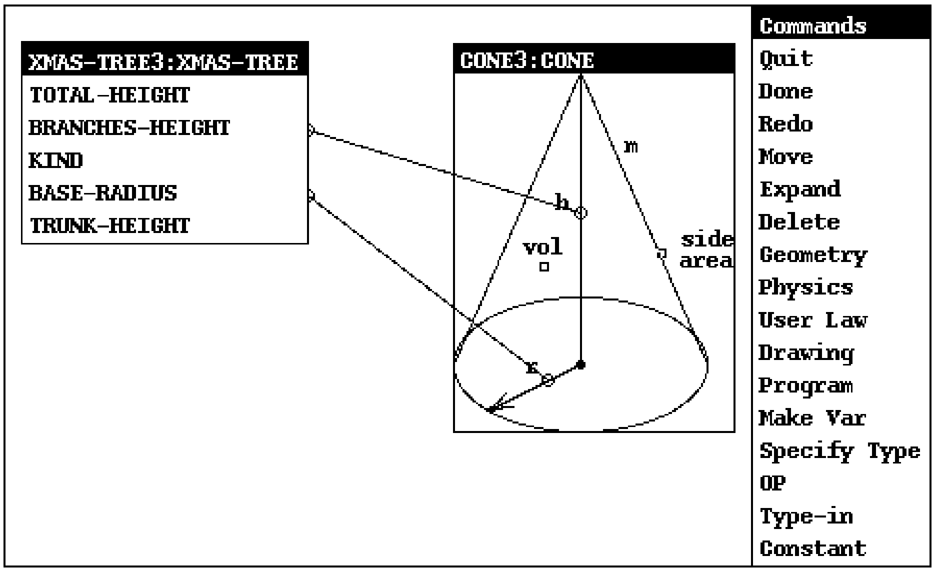

Figure 1: Viewing a Christmas Tree as a Cone

The view is specified by correspondences between the application data type and the abstract representation of a cone. The user first selects a "button" on the `cone` diagram, then selects a corresponding item within the menu of fields of the `xmas-tree` type. The system draws lines between the selected items to show the correspondences. Using these correspondences together with equations associated with a `cone`, the system constructs a view of a `xmas-tree` as a `cone`. Any generic procedure associated with `cone` can then be specialized and used for the `xmas-tree` data:

```
(gldefun t1 (tree:xmas-tree)
  (side-area (cone tree)))
```

This function, written in a Lisp-like functional form, requests the `side-area` of the `cone` view of the `tree`. The function is compiled into Lisp code that computes the `side-area` directly from the application data structure:

```
(LAMBDA (TREE)
  (* 3.1415926535897931
     (* (FIFTH TREE)
        (SQRT (+ (EXPT (FIFTH TREE) 2)
                 (EXPT (THIRD TREE) 2))))))
```

Mechanical translation can produce code for applications in other languages, such as C:

```
float t1c (tree)
  CMAS_TREE tree;
  { return 3.1415926535897931
```

```
        * tree->base_radius
        * sqrt(square(tree->base_radius)
            + square(tree->branches_height));
}
```

The user obtains this specialized code without having to understand the algorithm and without having to understand the implementation of the `cone` abstract data type. The user only needs to select the items in the diagram that correspond to the application data.

In this paper, we concentrate on numerical data of the kinds used in scientific and engineering programs. We have previously described [33] [34] [35] techniques for reuse of generic algorithms that deal with discrete data structures such as linked lists and trees; those methods, together with the methods described in this paper, allow reuse of algorithms that involve both discrete data structures and numerical data. The generic algorithms that can be specialized range from simple formulas, such as the formula for the area of a circle, to larger procedures such as testing whether a point is inside a polygon.

Section 2 gives a formal definition of views. Section 3 presents the algorithms that construct views from correspondences using algebraic manipulation of equations. Section 4 describes the graphical user interface for specifying views. Section 5 describes several ways in which views enable software reuse: by specialization of generic procedures, by translation of data to a desired form, and in object-oriented, functional, or transformational programming. Section 6 surveys related work, and a Section 7 presents conclusions. Finally, an on-line demonstration of the program is described.

2 Representation and Views

Traditional data types combine two issues that should be separated to facilitate reuse:

1. the way in which application data are represented
2. procedures associated with a conceptual kind of object

These are termed *implementation inheritance* and *interface inheritance* in object-oriented programming [9]. When the two are combined, the code of a procedure implicitly states assumptions about details of data representation. This inhibits software reuse, since any *assumptions* made in writing a procedure become *requirements* that must be met if the procedure is to be reused. For effective reuse, assumptions must be minimized. Object-oriented programming and functional programming partly separate the two issues, but still make some representation assumptions and also incur performance penalties, as we discuss later. Views make a clean separation between representation and procedures; compilation techniques yield good performance and can produce code in multiple target languages.

There are many ways in which representations of equivalent data can differ:

1. names of individual variables,

2. data representations and units of measurement of variables,

3. data structures used to aggregate variables,

4. the set of variables chosen to represent an object, and

5. the conceptual method, or *ontology* [30], of a representation.

For example, a vector could be represented using Cartesian coordinates or polar coordinates. In order to reuse a generic algorithm for application data, it must be possible either to translate the data into the form expected by the algorithm, or to modify the algorithm to work with the existing data; both can be done using views.

A view describes how an application data type corresponds to an abstract type. In effect, the view encapsulates the application data type and makes it appear to be of the abstract type. Fig. 2 illustrates how an application type `pipe` can be viewed as a `circle` that is defined in terms of `radius`. The `radius` of the `circle` corresponds to the `inside-diameter` of the `pipe` divided by 2. The view makes visible only the name `radius`, hiding the names of `pipe`; procedures defined for `circle` can be inherited through the view. Note that it is not the case that a `pipe` *is* a `circle`; rather, a `circle` is a useful view of a `pipe`. A second view of a `pipe` as a `circle`, using `outside-diameter`, is also useful.

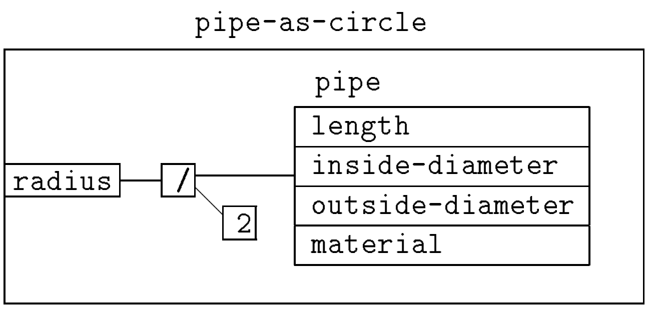

Figure 2: View as Encapsulation of Application Data

We formalize the notion of views as follows. An abstract type is considered to be an abstract record containing a set of *basis variables*. A view encapsulates the application type, hiding its names, and presents an external interface that consists of the basis variables of the abstract type. The view allows the basis variables to be both "read" and "written". The view emulates the abstract record by maintaining two properties:

1. *Storage property:* After a value z is "written" into basis variable v_i, a "read" of v_i will yield the value z.

2. *Independence property:* If a "read" of basis variable v_i yields the value z, and a value is then "written" into some basis variable v_j , $j \neq i$, a "read" of v_i will still yield z.

These two properties express the behavior normally expected of a record: stored values can be retrieved, and storing into one field does not change the values of other fields. If these properties are maintained, then the view faithfully emulates a record consisting of the basis variables, using the application type as its internal storage. The view inherits all the generic procedures associated with the abstract type. The view thus makes the application type appear to be a full-fledged implementation of the abstract type. Smith [45] uses the term *theory morphism* for a similar notion; Gries [17] uses the term *coupling invariant* and cites the term *coordinate transformation* used by Dijkstra [7].

The example of Fig. 2 illustrates a simple view in which a `pipe` is viewed as a `circle` in terms of the basis variable `radius`. A "read" of `radius` is accomplished by dividing the `inside-diameter` by 2; a "write" of `radius` is implemented by multiplying the value to be written by 2 and storing it into the `inside-diameter`. Source code that accesses the `radius` through the view and equivalent compiled code in C are shown below.

```
(gldefun t5 (p:pipe)
  (radius (circle p)) )

float t5 (p)
  PIPE *p;
  { return p->inside_diameter / 2; }

(gldefun t6 (p:pipe r:real)
  ((radius (circle p)) := r)
  p)

PIPE *t6 (p, r)
  PIPE *p; float r;
  { p->inside_diameter = 2 * r;
    return p; }
```

Because the view makes the application data appear to be exactly like the abstract type, any generic procedure that is defined for the abstract type will produce the same results using the application data through the view. With optimized in-line compilation of the code for access to data through views, the specialized versions of generic algorithms operate directly on the application data and are efficient.

In practice, it is often necessary to relax the storage and independence properties slightly:

1. The storage and independence properties are assumed to hold despite representation inaccuracy, *e.g.* floating-point round-off error. For example, if an application type that uses polar coordinates is viewed as a Cartesian vector, the value of `x` that is "read" may be slightly different from the value of `x` that was "written".

2. A view may be partial, *i.e.*, may define only those basis variables that are used. For example, to compute the area of a circle, only the `radius` is needed. Any use of an undefined basis variable is detected as an error by the compiler.

3 Creation of Views from Correspondences

A view contains a set of procedures to read and write each basis variable of the abstract type. These procedures could be written by hand. However, it is nontrivial to ensure that the procedures are complete, consistent, efficient, and satisfy the storage and independence properties. The procedures to view an application type as a `line-segment` (Fig. 5 below) are 66 lines of code; it would be difficult to make such a view manually. This section describes algorithms to derive the procedures for a view from correspondences, using symbolic algebra. A later section describes the graphical user interface that makes it easy to specify the correspondences.

3.1 Basis Variables and Equations

```
(setf (get 'circle 'basis-vars) '(radius))

(setf (get 'circle 'equations)
  '((= diameter      (* 2 radius))
    (= circumference (* pi diameter))
    (= area          (* pi (expt radius 2)))))
```

Figure 3: Equations for Circle

Each abstract type defines a set of basis variables and a set of equations. The basis variables and equations for a simple `circle` abstract type are shown in Fig. 3; those for a `line-segment` are shown in Fig. 4. The equations are written in a fully parenthesized Lisp notation with the operator appearing first in each subexpression. The basis variables are specified by the designer of the abstract data type and constitute a contract between the implementer and user of the generic procedures: the implementer can assume that every use of the generic procedures will behave as if the set of basis variables is directly implemented, and the user can assume that if the view of the application data emulates a direct implementation of the basis variables, the generic procedures will work for the application data.

A view is created from the correspondences provided by the graphical interface and from the equations. Each correspondence is a pair, `(abstract-var application-var)`, that associates a variable of the abstract type with a corresponding variable of the application type.[1] Correspondences are processed sequentially.

A view must have a procedure to read and write each basis variable, while maintaining the storage and independence properties. The procedures are derived by symbolic algebra. Although powerful packages such as Mathematica [54] exist, we use a simple equation solver

[1]Our interface actually allows an algebraic expression in terms of variables or computed quantities from the application type.

```
(setf (get 'line-segment 'basis-vars)
      '(p1x p1y p2x p2y))

(setf (get 'line-segment 'equations)
      '((= p1     (tuple (x p1x) (y p1y)))
        (= p1x    (x p1))
        (= p1y    (y p1))
        (= p2     (tuple (x p2x) (y p2y)))
        (= p2x    (x p2))
        (= p2y    (y p2))
        (= deltax (- p2x p1x))
        (= deltay (- p2y p1y))
        (= slope  (/ deltay (float deltax)))
        (= slope  (tan theta))
        (= slope  (/ 1.0 (tan phi)))
        (= length (sqrt (+ (expt deltax 2)
                           (expt deltay 2))))
        (= theta  (atan deltay deltax))
        (= phi    (- (/ pi 2.0) theta))
        (= phi    (atan deltax deltay))
        (= deltay (* length (sin theta)))
        (= deltax (* length (cos theta)))
        (= deltay (* length (cos phi)))
        (= deltax (* length (sin phi))) ) )
```

Figure 4: Equations for Line-Segment

and a relatively large (and possibly redundant) set of simple equations, as in Fig. 4. The equations that describe abstract data types typically are simple, so that a simple equation solver has been sufficient.

Equations are solved by algebraic manipulations. The equation solver is given a formula and a desired variable. If the left-hand side of the formula is the desired variable, the equation is solved. Otherwise, an attempt is made to invert the right-hand side, using algebraic laws, to find the desired variable; for example, the equation `(= x (+ y z))` is equivalent to `(= (- x y) z)` or `(= (- x z) y)`. These manipulations are performed recursively until the desired variable is isolated. Of course, this procedure assumes that the desired variable occurs exactly once in the equation.

The equations that specify a variable as a `tuple` of other variables are used to describe grouping relationships. Application data might specify a point in terms of separate `x` and `y` components, or as a data structure that represents a point as a whole (either by containing `x` and `y` components directly or by having a view that defines `x` and `y`). The user of the graphical interface should be able to specify either the whole representation of the point, or one or more of its components; however, it would be incorrect to specify both the whole and a component. The treatment of `tuple` equations, described below, guarantees that only correct combinations can be specified. Corresponding to each `tuple` definition are equations that define how to extract the components from the `tuple`.

3.2 Incremental Equation Solving

An equation set is initialized by making a copy of the equations of the target abstract type. As each correspondence between an abstract variable and an application variable is processed, the equation set is examined to see whether any equations can be solved. This incremental solution of the equations accomplishes several goals:

1. It produces equations for computing variables of the abstract type in terms of stored variables of the application type. These equations are later compiled into code.
2. It produces efficient ways of computing the variables, as described below.
3. The algorithm returns a list of all variables that are defined or computable based on the correspondences entered thus far. The buttons for these variables are removed from the user interface, preventing the user from entering a contradictory specification.

The following algorithm, which we call `var-defined`, is performed for each correspondence, (abstract-variable application-variable).

1. The abstract variable is added to a list of defined variables and to a list of solved variables, *i.e.*, variables that are computable from the correspondences specified so far.
2. Each equation is examined to determine what unsolved variables it contains:
   (a) If there is exactly one unsolved variable, and the equation can be solved to produce a new equation defining that variable, then
      i. The right-hand side of the new equation is symbolically optimized using a pattern-matching optimizer.
      ii. The new equation is saved for later use in generating code.
      iii. The variable is added to the list of solved variables.
      iv. The equation is deleted.
   (b) If there are no unsolved variables, the equation is deleted. This case can occur if the equation set contains multiple equations for computing the same variable. Assuming that the equation set is consistent, the equation will represent a mathematical identity among variables that have already been defined.
   (c) If the equation defines a `tuple` variable, and some component of the `tuple` has been solved, the variable is added to a list of deleted `tuple` variables, and the equation is deleted.

(d) If the equation contains a deleted `tuple` variable, the equation is deleted. A deleted `tuple` variable can never become defined.

3. If any variables were solved in step 2.a., step 2 is repeated.

4. Finally, a list of variables is returned; this list includes the newly defined variable, any variables that were solved using equations, and any deleted `tuple` variables.

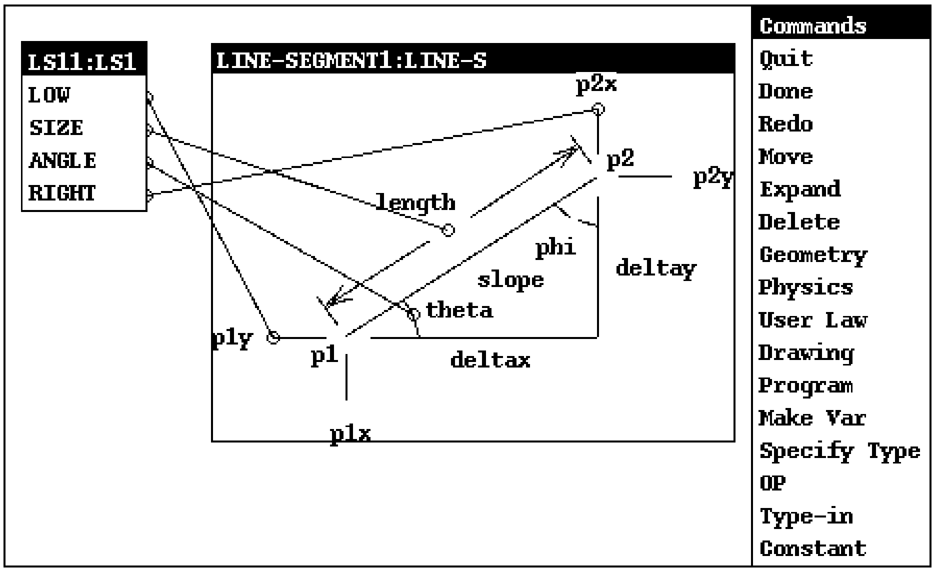

Figure 5: Viewing `LS1` as a Line Segment

Fig. 5 shows correspondences between an application type `LS1` and a `line-segment`. When the correspondences are specified using the graphical user interface, `var-defined` is called as each individual correspondence is specified. The correspondences are saved so that a view can be remade without using the graphical interface. Each correspondence is a pair of an abstract variable and a field of the application type; for the example of Fig. 5 the correspondences are:

```
((P1Y    LOW)
 (LENGTH SIZE)
 (THETA  ANGLE)
 (P2X    RIGHT))
```

Fig. 6 shows the calls to `var-defined` and the actions performed as each correspondence is processed; each action is labeled with the number of the step in the algorithm.

After all correspondences have been processed, two results have been produced:

1. a list of abstract variables that are defined as references to the application type.

2. a list of equations that define other abstract variables.

```
1. Enter var-defined, var = P1Y
  2c. deleting tuple
               (= P1 (TUPLE (X P1X) (Y P1Y)))
  2d. deleting eqn  (= P1X (X P1))
  2d. deleting eqn  (= P1Y (Y P1))
4. exit, vars (P1 P1Y)
1. Enter var-defined, var = LENGTH
4. exit, vars (LENGTH)
1. Enter var-defined, var = THETA
  2a. solved eqn    (= SLOPE (TAN THETA))
  2a. solved eqn    (= PHI (- (/ PI 2.0) THETA))
  2a. solved eqn    (= DELTAY
                       (* LENGTH (SIN THETA)))
  2a. solved eqn    (= DELTAX
                       (* LENGTH (COS THETA)))
  3.  repeating step 2.
  2a. solved eqn    (= DELTAY (- P2Y P1Y))
          giving    (= P2Y (+ DELTAY P1Y))
  2b. deleting eqn  (= SLOPE (/ DELTAY
                                (FLOAT DELTAX)))
  2b. deleting eqn  (= SLOPE (/ 1.0 (TAN PHI)))
  2b. deleting eqn  (= LENGTH
                       (SQRT (+ (EXPT DELTAX 2)
                              (EXPT DELTAY 2))))
  2b. deleting eqn  (= THETA
                       (ATAN DELTAY DELTAX))
  2b. deleting eqn  (= PHI (ATAN DELTAX DELTAY))
  2b. deleting eqn  (= DELTAY
                       (* LENGTH (COS PHI)))
  2b. deleting eqn  (= DELTAX
                       (* LENGTH (SIN PHI)))
  3.  repeating step 2.
  2c. deleting tuple
               (= P2 (TUPLE (X P2X) (Y P2Y)))
  2d. deleting eqn  (= P2X (X P2))
  2d. deleting eqn  (= P2Y (Y P2))
4. exit, vars (P2 P2Y DELTAX DELTAY PHI
               SLOPE THETA)
1. Enter var-defined, var = P2X
  2a. solved eqn     (= DELTAX (- P2X P1X))
          giving     (= P1X (- P2X DELTAX))
  3.  repeating step 2.
4. exit, vars (P1X P2X)
```

Figure 6: Incremental Equation Solving

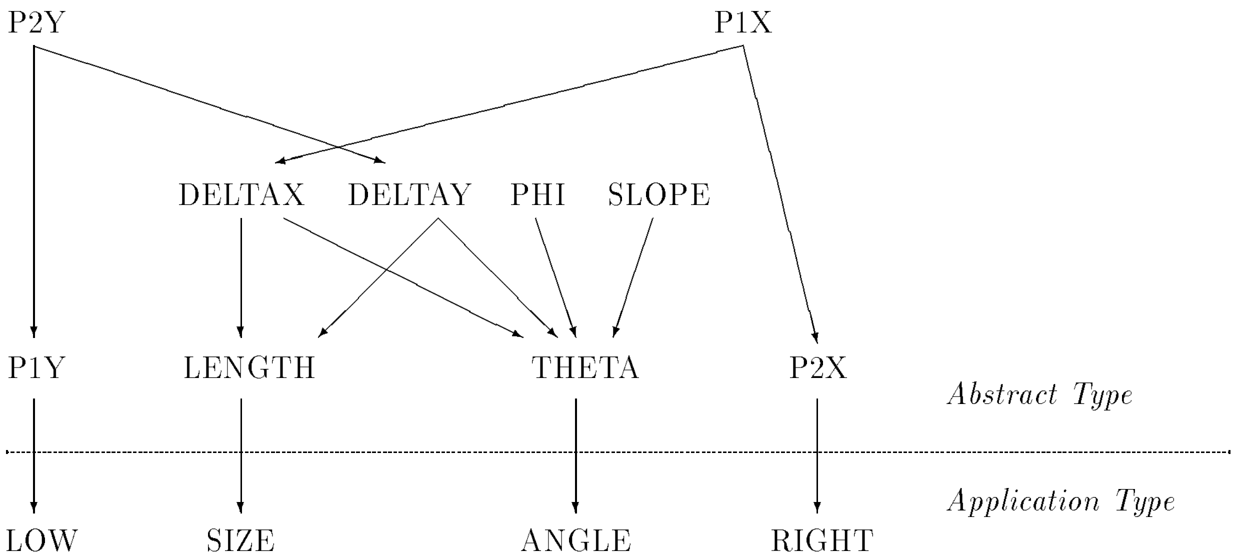

Figure 7: Variable Dependency Graph for `LS1`

The equations form a directed acyclic graph that ultimately defines each variable on the left-hand side of an equation in terms of references to the application type; Fig. 7 shows the graph for the `LS1` example. If each variable on the right-hand side of an equation is replaced by its defining equation, if any, and the process is repeated until no further replacements are possible, the result will be an expression tree whose leaves are references to the application type. [2] This is easily proved by induction. Initially, the only solved variables are those that are defined by correspondence with the application type. Each variable that becomes solved via an equation is defined in terms of previously solved variables. Therefore, the graph of variable references is acyclic, and replacement of variables by their equational definitions will result in an expression tree whose leaf nodes are references to the application type.

In general, the abstract type will define procedures that compute all of the variables shown in the diagram, as functions of the basis variables. Therefore, it is only strictly necessary for the view type to define procedures to compute the basis variables; all other variables could be derived from those. However, this approach might be inefficient. For example, it would be inefficient to calculate the `LENGTH` from the basis variables for the `LS1` application type, since the `LS1` type stores the `LENGTH` directly as the field `SIZE`. It is desirable to compute each variable as directly as possible from the application data, *i.e.*, using as few data references and operations as possible. The `var-defined` algorithm operates incrementally and produces an equation to calculate each abstract variable as soon as it is possible to do so; the equations therefore are close to references to the application type. It would be possible to guarantee optimal computation of each variable by implementing a search algorithm:

[2] This replacement process is performed by the GLISP compiler when code that uses the view is compiled.

1. Assign to each abstract variable that is defined as a field of the application type a cost of `1` and mark it solved.

2. Examine each equation in the equation set to determine which variables can be solved in terms of existing solved variables. Assign, as the cost of such a solution, the sum of the costs of its components and the cost of each operator. If the variable is unsolved or has a higher cost, adopt the new equation as its definition and the new cost as its cost.

3. Repeat step 2 until no further redefinitions occur.

We have not implemented this algorithm because the `var-defined` algorithm approximates it and has produced excellent results in practice; this algorithm would be useful if some operators had much higher cost than others.

3.3 Storing Basis Variables

Some generic procedures both read and write data; thus, it is necessary to define methods that "store" into basis variables through the view. We assume that values can be stored only into basis variables; this is a reasonable restriction, since it corresponds to a record consisting of the basis variables in an ordinary programming language. Each storing method is a small procedure whose arguments are an instance of the application type and a value that is to be "stored" into the basis variable. The procedure must update the application data in such a way that the storage and independence properties are maintained. Without this constraint, the method used to "store" a variable would be ambiguous, and generic algorithms might behave differently with different data implementations.

The variables of the abstract type that correspond directly to fields of the application type are called *transfer variables*; a list of these is saved. In Fig. 7, the transfer variables are `P1Y`, `LENGTH`, `THETA`, and `P2X`. Storing new values for all transfer variables following a change in value of a basis variable would accomplish a storing of the basis variable. Such a procedure could be derived in a trivial way:

1. compute the values of all basis variables, other than the one to be stored, from the application data;

2. compute values of the transfer variables from the basis variables;

3. store these values into the application data structure.

However, such a procedure might be inefficient. It is desirable to update the smallest possible subset of stored variables. The algorithm below accomplishes this.

1. A set of *basis equations* is created. This is done by initializing an equation set with all equations of the abstract type, then calling `var-defined` for each basis variable.

The result is a set of equations for computing each non-basis variable in terms of basis variables and, implicitly, a dependency graph that shows the dependency of all variables on basis variables.

2. The set `xfers` is computed; this is the subset of the transfer variables that depend on the basis variable that is to be stored. Dependency is determined by recursively computing the union of the leaf nodes of the expression tree for the transfer variable, as implicitly defined by the basis equations.

3. The set `dep` is computed; this is the subset of the basis variables that some member of `xfers` depends on.

4. Code is generated to compute each basis variable in `dep`, other than the basis variable to be stored, from the application data.

5. Code is generated to compute each transfer variable in `xfers` and to store it into the corresponding field of the application structure. Temporary variables are generated for intermediate variables that are used in computing the transfer variables if they are used more than once; otherwise, the intermediate variables are expanded using the basis equations.

The result of this algorithm is a procedure that revises the application data structure only as much as necessary to emulate a "store" into a basis variable while leaving the values of other basis variables unchanged. One such procedure is created for each basis variable. In the special case where a basis variable corresponds exactly to a transfer variable and does not affect the value of any other transfer variable, no procedure needs to be created.

```
(GLAMBDA (VAR-LS1 P1Y)
  (LET ((P1X (P1X VAR-LS1))
        (P2X (P2X VAR-LS1))
        (P2Y (P2Y VAR-LS1))
        DELTAY DELTAX)
    (DELTAY := (- P2Y P1Y))
    (DELTAX := (- P2X P1X))
    ((LOW (LS1 VAR-LS1)) := P1Y)
    ((SIZE (LS1 VAR-LS1)) :=
          (SQRT (+ (EXPT DELTAX 2)
                   (EXPT DELTAY 2))))
    ((ANGLE (LS1 VAR-LS1))
           := (ATAN DELTAY DELTAX))
    P1Y))
```

Figure 8: Method to Store `P1Y` into a `LS1` Data Structure

An example of a procedure to store the basis variable `P1Y` into an `LS1` is shown in Fig. 8. Although `P1Y` corresponds directly to the `LOW` field, it is also necessary to update the values of the `SIZE` and `ANGLE` fields in order to leave the value of the basis variable `P2Y` unchanged. The `RIGHT` field does not need to be updated.

3.4 Creating Application Data from Basis Variables

Some generic procedures create new data structures; for example, two vectors can be added to produce a new vector. Therefore, a view must have a procedure that can create an instance of the application type from a set of basis variable values. This is similar to storing a basis variable, except that all basis variables are stored simultaneously.

```
(GLAMBDA (SELF P1X P1Y P2X P2Y)
  (LET (DELTAY DELTAX)
    (DELTAY := (- P2Y P1Y))
    (DELTAX := (- P2X P1X))
    (A LS1
       LOW   P1Y
       SIZE  (SQRT (+ (EXPT DELTAX 2)
                      (EXPT DELTAY 2)))
       ANGLE (ATAN DELTAY DELTAX)
       RIGHT P2X)))
```

Figure 9: Method to Create a `LS1` Data Structure from Basis Variables

Fig. 9 shows the procedure that creates a `LS1` data structure from a set of `line-segment` basis variables. Two local variables, `DELTAX` and `DELTAY`, have been created since these variables are used more than once. When compiled by GLISP, the `A` function produces an instance of the `LS1` data structure with the specified component values. When `tuple` substructures are involved, additional `A` functions are inserted to create them as well. The GLISP compiler invokes this method when compiling an `A` function; as a result, the use of views can be recursive. For example, a `line-segment` could be specified by two points `P1` and `P2`, each of which is in polar coordinates `r` and `theta` with a view as a Cartesian `vector`. In this case, creating a new `line-segment` instance would also create new polar components.

3.5 Data Translation through Views

Suppose that there are two data types t_1 and t_2, each of which has a view as abstract type a. Then it is easy to translate data d_1 of type t_1 into type t_2 :

1. For each basis variable v_i of the abstract type a, compute its value from data d_1 using the view from type t_1 to a.

$$\begin{tikzcd} t_1 \arrow[r, "view"] & a & t_2 \arrow[l, "view"'] \end{tikzcd}$$

Figure 10: Application Data that Share a View

2. Create a data structure of type t_2 from the set of basis variables using the view from type t_2 to a.

However, this algorithm might not be efficient. For example, types t_1 and t_2 might store the same non-basis variable, which could be transferred directly without computing basis variable values. For this reason, we have developed another algorithm for creating data translation procedures. The algorithm begins by finding the unique abstract type a that is the intersection of the views from the source and goal types (the name of the view to be used for each type can be specified if the intersection is not unique). Next, the set `xfers` is computed; this is the set of transfer variables of the goal type, *i.e.*, the variables of the abstract type that correspond to stored fields in the goal type. Code is then generated to create an instance of the goal type using the values of the transfer variables from the shared view, a, of the source type. Since the view computes these transfer variables as close to the source data as possible, the resulting code is often more efficient, and never worse, than a version that computes basis variables first. Optimal code could be guaranteed by a search process, as described earlier.

3.6 Unit Conversion

Application data can use various units of measurement. Most programming languages omit units of measurement entirely: there is no way to state the units that are used, much less to check consistency of units. GLISP allows units to be specified, and it automatically performs unit conversion [37] and checks validity. Therefore, unit conversion is performed automatically for all uses of views described in this paper.

3.7 Error Checking

A user could specify a partial set of correspondences between the abstract type and application type: some generic procedures defined by the abstract type may involve only a subset of the basis variables. For example, it is possible to compute the `slope` of a `line-segment` if only the abstract variable `theta` is defined. However, any actual errors are detected by our system.

After the user enters correspondences, `mkv` issues a warning if any basis variables remain unsolved. A method to store a basis variable is produced only if that variable was solved and all basis variables in the set `dep` (described in section 3.3) were solved. A method to create an application data structure from basis variables is produced only if all basis variables were solved. An attempt to specialize a generic procedure that uses missing parts of a view type will result in errors detected by the GLISP compiler.

It is possible for the user to specify a correspondence that can be computed, but cannot be "stored". For example, a variable of the abstract type could be defined as the sum of two stored fields of the application type; it is not possible to determine unambiguously how to "store into" this variable. An attempt to store into such a variable will result in an error detected by the GLISP compiler.

Views do not replace or evade strong typing. Indeed, all of the code that is produced is correctly typed and can be mechanically translated to a strongly typed language. Views enhance type checking by checking units of measurement, and they provide encapsulation: when a view is used, only the operations defined by the view are available. The view mechanism provides the benefits of encapsulation while enhancing reusability and producing more efficient code than other encapsulation mechanisms [9].

4 Graphical User Interface

A graphical user interface makes it easy to specify correspondences between an application type and an abstract type using a mouse pointing device. The program, called `mkv` (for "make view"), is called with the goal abstract type and source type as arguments. `mkv` opens a window and draws a menu of the values available from the source type and a diagram or menu of variables of the abstract type (Fig. 1).

The user selects items from the goal diagram by clicking the mouse on labeled "buttons" on the diagram. The interface program highlights a button when the mouse pointer is moved near it; clicking the mouse selects the item. The user then selects a corresponding item from the menu that represents the application data; a line is drawn between the two items to show the correspondence. The user can also specify an algebraic expression, involving one or more application data fields, by selecting `OP` from the command menu and specifying an expression tree of operators and operands.

A diagram can present buttons for many ways to represent a given kind of data.[3] With such a diagram, it is likely that there will be buttons that correspond directly to the existing form of the data. Fig. 11 shows the diagram for a `line-segment`. Even though `line-segment` is a simple concept, a line segment could be specified in many ways: by two end points, or by one end point, a length, and an angle, *etc.* The diagram is intended to present virtually all the reasonable possibilities as buttons. This interface has several advantages:

1. Diagrams are easily and rapidly perceived by humans [26], and they are widely used in engineering and scientific communication [11].

[3]If there were alternative representations that were sufficiently different to require different diagrams, a menu to select among alternative diagrams could be presented to the user first.

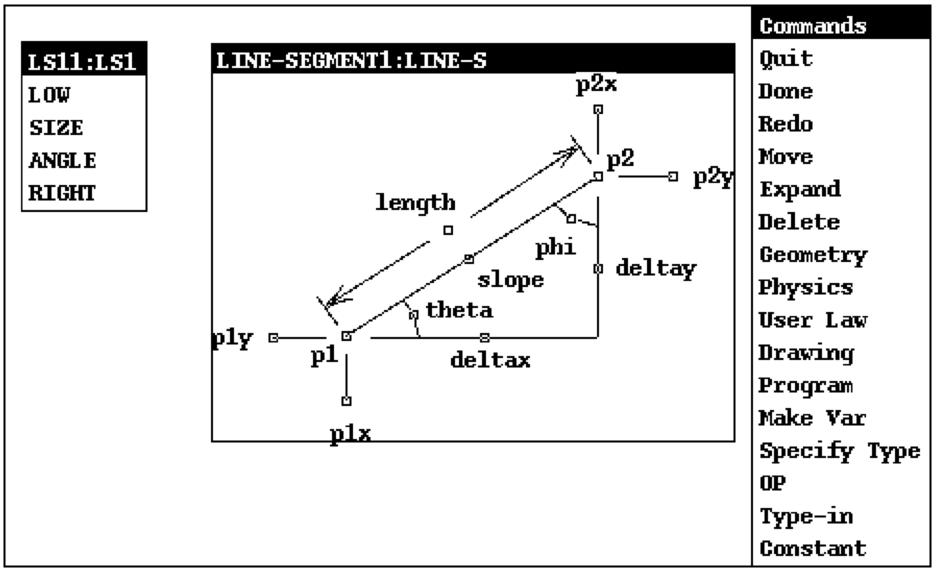

Figure 11: Initial Diagram for a Line Segment

2. The interface is self-documenting: the user does not need to consult a manual, or know details of the abstract data type, to specify a view.

3. The interface is very fast, requiring only a few mouse clicks to create a view.

4. The user does not have to perform error-prone algebraic manipulations.

It is easy to add diagrams for new abstract types. A drawing program allows creation of diagrams, including buttons. The only other thing that is needed is a specification of the basis variables and equations for the abstract type, as shown in Figs. 3 and 4.

5 Achieving Reuse with Views

Views are important for practical reuse because they achieve a clean separation between the representation of application data and the abstract data types used in generic procedures. The user obtains the benefits of reuse without having to understand and conform to a standard defined by someone else. In this section, we describe several ways in which views can be applied to achieve reuse:

1. by specialization of generic procedures through views

2. by translation of data

3. by creation of wrappers or transforms for use with object-oriented, functional, or transformational programming.

5.1 Specialization of Generic Procedures

The GLISP compiler [31, 32] can produce a specialized version of a generic procedure by compiling it relative to a view. The result is a self-contained procedure that performs the action of the generic procedure directly on the application data. The specialized procedure can be used as part of the application program. GLISP is a high-level language with abstract data types that is compiled into Lisp; it is implemented in Common Lisp [47]. GLISP types include data structures in Lisp and in other languages. GLISP is described only briefly here; for more detail, see [33] and [31].

Data representation is a barrier to reuse in most languages because the syntax of program code depends on the data structures that are used and depends on whether data is stored or is computed. This prevents reuse of code for alternative implementations of data. GLISP uses a single syntax, similar to a Lisp function call, to access features of an object [33]:

$$(\; feature \;\; object \;)$$

The interpretation of this form depends on the compile-time type of the *object*:

1. If *feature* is the name of a data field of *object*, code to access that field is compiled.
2. If *feature* is a message selector that is defined for the type of *object*, then
   (a) a runtime message send can be compiled, or
   (b) the procedure that implements the message can be specialized or compiled in-line, recursively.
3. If *feature* is the name of a view defined for the type of *object*, then the type of *object* is locally changed to the view type.
4. If *feature* is defined as a function, the code is left unchanged as a function call.
5. Otherwise, a warning message is generated that *feature* is undefined.

In this way, a generic program can access a feature of an application type without making assumptions about how that feature is implemented. This is similar to object-oriented programming; however, since actual data types are known at compile time, message sending can be eliminated and replaced by in-line compilation or by calls to specialized procedures. This is equivalent to making transformations to the code, and it significantly improves efficiency of the compiled code.

A view is implemented as a GLISP type whose stored form is the application data; the abstract type associated with the view is a super-class of the view type. The view type defines messages to compute the basis variables of the abstract type from the application type. As a simple example, consider a (handwritten, for simplicity) view of a `pipe` as a `circle` (Fig. 2):

```
(pipe-as-circle (p pipe)
  prop   ( (radius
             ( (inside-diameter p) / 2) ) )
  supers (circle))
```

The stored form of pipe-as-circle is named p and is of type pipe. The basis variable radius is defined as the inside-diameter of the pipe divided by 2. circle is a superclass, so all methods of circle are inherited by pipe-as-circle:

```
(gldefun t7 (p:pipe) (area (circle p)))

float t7 (p)
  PIPE *p;
  {
    return 0.78539816
           * square(p->inside_diameter);
  }
```

The function t7 has an argument p whose type is pipe. The code (circle p) changes the type of p to the view type pipe-as-circle. The definition of area is inherited from circle and compiled in-line; this definition is in terms of the radius, which is expanded in-line as the inside-diameter of the pipe divided by 2. The use of the view has zero cost at runtime because the optimizer has combined the translation from the view with the constant π, producing a new constant $\pi/4$.

Once a view has been made, all of the generic procedures of the abstract type are available by automatic specialization. Thus, a single viewing process allows reuse of many procedures. As a larger example, we consider a function that finds the perpendicular distance of a point to the left of a directed line segment (if the point is to the right, the distance will be negative). Although this is not a large function, it is not easy to find an effective algorithm in a reference book or to derive it by hand: graduate students assigned to produce such a procedure by hand for the LS1 data type report that it takes from 20 minutes to over an hour. Books often omit important features: [3] assumes that a human will determine the sign of the result. Even if a formula is found, it may not be expressed in terms of the available data. Some versions of the formula involve division by numbers that can be nearly zero. Reuse of a carefully developed generic procedure is faster, less costly, and less error-prone than writing one by hand.

Code that results from the viewing and compilation process is presented below. We do not expect a user of our system to read or understand such code. While some authors [42] have proposed that the user read and edit code that is produced by an automatic programming system, we do not: it is not easy to read someone else's code, and this is especially true for machine-generated code that has been optimized. We expect that users will treat the output of our system as a "black box", as is often done with library subroutines.

Fig. 12 shows the generic function line-segment-leftof-distance. Fig. 13 shows a specialized version of it for a LS1 record in C. This code was produced by GLISP compilation

```
(gldefun
  line-segment-leftof-distance
   (ls:line-segment p:vector)
    ( ( (deltax ls) * ( (y p) - (p1y ls) )
        - (deltay ls) * ( (x p) - (p1x ls) ) )
      / (length ls) ) )
```

Figure 12: Generic Function: Distance of Point to the Left of a Line Segment

```
float lsdist (l, p)
  CLS1 *l;
  CVECTOR *p;
  {
    return cos(l->angle)
           * (p->y - l->low)
           - sin(l->angle)
             * (p->x
                - (l->right
                   - l->size * cos(l->angle)));
  }
```

Figure 13: `leftof-distance` specialized for `LS1` in C

followed by mechanical translation into C; the resulting C code has no dependence on Lisp. Since the `LS1` type is quite different from the abstract type `line-segment`, this example illustrates that a single generic procedure can be reused for a variety of quite different implementations of data. The specialized version is efficient (two multiplies and division by the `length` were removed by algebraic optimization) and is expressed in terms of the application data, without added overhead. While repeated subexpressions sometimes appear in specialized code, these can be removed by the well-understood compiler technique of common subexpression elimination [43].

A useful viewpoint is that there is a mapping between an abstract data type and the corresponding application type, and an isomorphism between the two based on a generic algorithm and a specialized version of that algorithm. Just as compilation of a program in an ordinary programming language produces an equivalent program in terms of lower-level operations and data implementations, specialization of a generic procedure produces an equivalent procedure that is more tightly bound to a specific implementation of the abstract data. Similar viewpoints are found in mathematical definitions of isomorphism (*e.g.*, [39], p. 129), in denotational semantics [15], and in work on program transformation [45] [17]. [13] describes views in terms of such isomorphisms. However, we note that many applications use approximations that do not satisfy the strict mathematical definition of isomorphism.

5.2 Reuse by Data Translation

One way to reuse a program with data in a different format is to translate the data into the right form; translation is also required to combine separate data sets that have different formats. As use of computer networks increases, users will often need to combine data from different sources or use data with a program that assumes a different format. The ARPA Knowledge-Sharing Project [30] addresses the problem of sharing knowledge bases that were developed using different ontologies. Writing data translation programs by hand requires human understanding of both data formats. [40] presents a language for describing parameter lists and a system that produces interface modules that translate from a source calling sequence to a target calling sequence. Our paper [33] described automatic construction of translation procedures from correspondences; the techniques in this paper extend those.

Standardization of data representations and formats is one way to achieve interoperability. However, it is difficult to find standards that fit everyone's needs, and conformity to standards is costly for some users. Views provide the benefits of standardization without the costs. As described in Section 3.5, if there are views from two application types to a common abstract type, a data conversion procedure to convert from one application type to the other can be generated automatically. Thus, interchange of data requires only that the owner of each data set create a view that describes how the local data format corresponds to an abstract type. Only n views are needed to translate among any of n data formats, and knowledge of others' data formats is unnecessary. Use of remote procedure calls to servers across a network could be facilitated by advertising an abstract data type expected by the remote procedure; if users create views of their data as the abstract data, the translation of data can be performed automatically.

Materialization of a new data set may be computationally expensive, but the cost is minor for small data sets. It is also reasonable to translate a large data set incrementally, or as a whole if a large amount of computation will be performed on it; [25] found that translation of data between phases of a large compiler was a minor cost.

5.3 Object-Oriented, Functional, and Transformational Programming

The algorithms in this paper can be used with other styles of program development.

One benefit of object-oriented programming (OOP) is reuse of methods that have been pre-defined. At the same time, OOP systems often impose restrictions that application objects must meet, *e.g.*, that an application object must have the same stored data as its superclass or must provide certain methods with the same names that are used in the existing methods. Thus, reuse with OOP requires conformance to existing standards. Views allow new kinds of objects to be used with existing methods.

The term *adapter* or *wrapper* [9] denotes an object class that makes application data appear to be an instance of a target class. The wrapper object contains a pointer to the application data and performs message translation to implement the messages required of a member of the target class. In the `pipe` example shown in Fig. 2, a wrapper class `pipe-as-circle` would implement the `radius` message expected by the class `circle` by sending an `inside-diameter` message to the `pipe` and returning the result of this message divided by 2. The algorithms and user interfaces described in this paper could be used to create wrapper classes. The messages expected by the target class correspond to basis variables. The equations produced in making views can easily be converted into methods in the appropriate syntax for the OOP system. We have previously described [35] the use of wrapper objects to allow display and direct-manipulation editing of user data by generic editor programs.

There are two disadvantages of using wrapper objects [9]. It is necessary to allocate a wrapper object at runtime, which costs time and storage. Second, since translation of data is performed interpretively, there is overhead of additional message sending, and the same translation may be performed many times during execution. However, in cases where these costs are tolerable, wrapper objects are an easy way to achieve reuse.

Views can be used in a related way with functional programming. Functions can be created to calculate the value of each basis variable from data of the application type. In the `pipe` example, a function `radius(p:pipe)` would be created that returns the value of the `inside-diameter` divided by 2. Storing of basis variables could be implemented by storing into the application data, for functional languages that allow this, or by creating new data with the updated values in the case of strict functional languages.

The Polya language [17] [8] allows a user to specify a set of transformations that are made to the intermediate code of a generic algorithm; these transformations are equivalent to the transformations performed by the GLISP compiler [31]. The algorithms presented here could be used to generate transforms for a language such as Polya.

6 Related Work

6.1 Software Reuse

Krueger [24] is an excellent survey of software reuse; it also gives criteria for *effective* software reuse. Biggerstaff and Perlis [4] contains papers on theory and applications of software reuse. Mili [29] provides an extensive survey of approaches to software reuse, emphasizing the technical challenges of reuse for software production. Artificial intelligence approaches to software engineering are described in [1], [28], and [41]. Some papers from these sources are reviewed individually in this section.

6.2 Software Components

Weide [52] proposes a software components industry analogous to the electronic components industry, based on formally specified and unchangeable components with rigid interfaces. Because the components would be verified, unchangeable, and have rigid interfaces, errors in using or modifying them would be prevented. Views, as described in this paper, allow components to be adapted to fit the application.

6.3 Languages with Generic Procedures

Programming languages such as Ada, Modula-2 [21] [27], and C++ [48] allow parameterized modules; by constructing a module containing generic procedures for a parameterized abstract data type, the user obtains a specialized version of the module and its procedures. However, these languages allow much less parameterization than is possible using views; for example, it is not possible to define a procedure that works for either Cartesian or polar vectors, and it is not even possible to state units of measurement in these languages.

6.4 Functional and Set Languages

ML [53] [38] is like a strongly typed Lisp; it includes polymorphic functions (*e.g.*, functions over lists of an arbitrary type) and functors (functions that map structures, composed of types and functions, to structures). ML also includes references (pointers) that allow imperative programming. ML functors can instantiate generic modules such as container types. However, ML does not allow generics as general as those described here. Our system allows storing into a data structure through a view; for example, a radius value can be "stored" into a `pipe` through a view. Our system also allows composition of views.

Miranda [50] is a strongly typed, purely functional language that supports higher-order functions. While this allows generic functions to be written, it is often difficult to write efficient programs in a purely functional language [38]: a change to data values requires creation of a new structure in a functional language.

6.5 Transformation Systems

Transformation systems generate programs starting from an abstract algorithm specification; they repeatedly apply transformations that replace parts of the abstract algorithm with code that is closer to an implementation, until executable code is finally reached. Our views specify transformations from features of abstract types to their implementations.

Kant *et al.* [23] describe the Sinapse system for generating scientific programs involving simulation of differential equations over large spatial grids for applications such as seismic analysis. Sinapse accepts a relatively small program specification and generates from it a much larger program in Fortran or other languages by repeatedly applying transformations within Mathematica [54]. This system appears to work well within its domain of applicability.

KIDS [46] can transform general algorithms into executable versions that are highly efficient for combinatorial problems. The user selects transformations to be used and supplies a formal theory for the domain of application. This system is interesting and powerful, but its user must be mathematically sophisticated.

Gries and Prins [16] proposed a system that would use syntactic transformations to specify the implementation of abstract algorithms. [31] describes related techniques that were implemented earlier. Volpano [51] and Gries [17] describe transformation of programs by syntactic *coordinate transformations* for variables or for patterns involving uses of variables. Our views require fewer specifications because they operate at semantic (type-based and algebraic) levels, rather than at the syntactic level; most patterns of use are handled automatically by the algebraic optimization of the compiler.

Berlin and Weise [2] used *partial evaluation* to improve the efficiency of scientific programs. Using information that some features of a problem are constant, their compiler performs as many constant calculations as possible at compile time, resulting in a program that is specialized and runs faster. Our system incorporates partial evaluation by means of in-line compilation and symbolic optimization.

6.6 Views

Goguen describes a library interconnection language called LIL [12] and has implemented the language OBJ3 [13] [14] that incorporates parameterized programming and views. A view in OBJ3 is a mapping between a theory `T` and a module `M` that consistently maps sorts (types) of `T` to sorts of `M` and operations of `T` to operations of `M`. OBJ3 is based on formal logical theory using order-sorted algebra; it operates as a theorem prover in which computation is performed by term rewriting. The authors state ([14], p. 56):

> OBJ3 is not a compiler, but is rather closer to an interpreter. The associative/commutative rewrite engine is not efficient enough for very large problems...

Tracz [49] describes LILEANNA, which implements LIL for construction of Ada packages; views in LILEANNA map types, operations, and exceptions between theories. Our system creates views from correspondences between application types and mathematical objects; the possible correspondences are more general than the one-to-one correspondences specified in OBJ3 views. Our system produces efficient specialized procedures in ordinary programming languages and is intended to be used as a program generation system.

Garlan [10] and Kaiser [22] use views to allow multiple tools in a program development environment to access a common set of data about the program being developed. Their MELD system can combine *features*, which are collections of object classes and methods, to allow additive construction of a system from selected component features.

Hailpern and Ossher [18] describe views in OOP that are subsets of the methods defined for a class. They use views to restrict use of certain methods; for example, a debugger could use methods that were unavailable to ordinary programs. This system has been used in the RPDE[3] program development environment [19].

6.7 Data Translation

IDL (Interface Description Language) [25] allows exchange of large structured data, possibly including structure sharing, between separately written components of a large software system such as a compiler. IDL performs representation translation, so that different representations of data can be used by the different components. Use of IDL requires that the user write precise specifications of the source and target data structures.

Herlihy and Liskov [20] describe a method for transmission of structured data over a network, with a possibly different data representation at the destination. Their method employs user-written procedures to encode and decode the data into transmissible representations. They also describe a method for transmission of shared structures. [5] describes a system that automatically generates stub programs to interconnect processing modules that are in different languages or processors; this work is complementary to the techniques presented in this paper.

The Common Object Request Broker Architecture (CORBA) [6] includes an Interface Definition Language and can automatically generate stubs to allow interoperability of objects across distributed systems and across languages and machine architectures.

Purtilo and Atlee [40] describe a system that translates calling sequences by producing small interface modules that reorder and translate parameters as necessary for the called procedure.

In all of these cases, the emphasis is on relatively direct translation of data, focusing on issues of record structure, number representation, etc. The techniques described in this paper could be used to extend these approaches to cases where the ontology, or method of description of objects, differs. Specialization of generic algorithms is more efficient than interpretive conversion of data.

6.8 Object-oriented Programming

OOP is popular as a mechanism for software reuse; Gamma *et al.* [9] describe design patterns that are useful for OOP. We have described how views could be used to construct wrapper objects that make application objects appear to be members of a desired class. Use of views with the GLISP compiler extends good ideas in OOP:

1. OOP makes the connection between a message and the corresponding procedure at runtime; this is often a significant cost [9]. C++ [48] has relatively efficient message dispatching, at some cost in flexibility. Because GLISP [31] can specialize a method

in-line and optimize the resulting code in context, the overhead of interpretation is eliminated, and often there is no extra cost.

2. Interpretation of messages in OOP postpones error checking to runtime. With views and GLISP, type inference and in-line expansion cause this checking to be done statically.

3. Views provide a clean separation between application data and the abstract type, while some OOP systems require conformance between an instance and a superclass, *e.g.*, the instance may have to contain the same data variables. With views, there can be multiple views as the same type, *e.g.* a `pipe` can be viewed as a `circle` in two distinct ways. Views allow a partial use of an abstract type, *e.g.* the `area` of a `circle` can be found without specifying its `center`.

4. In OOP, the user must learn the available classes and messages. Some OOP operating systems involve over 1,500 classes, so this is nontrivial. With views, the user does not have to understand the abstract type, but only has to indicate correspondences between the application type and the abstract type. The user interface is self-documenting.

5. Our system allows translation to a separate application language, such as C, without requiring that the application be written in a particular language.

7 Discussion and Conclusions

We believe that generation of application code by compilation through views is an efficient and practical technique. Expansion of code through views is recursive at compile time, so that composition of views is possible. Generic algorithms are often written in terms of other generics. While the examples presented in this paper have been small ones for clarity, the approach does scale to larger algorithms. We have produced specialized versions of algorithms that comprise about 200 lines of code in C, *e.g.*, finding the convex hull of a set of points (which uses the `leftof-distance` generic function defined for a `line-segment`), finding the perimeter, area, and center of mass of a polygon, etc. Compilation and translation to C of a 200-line program takes approximately 10 seconds on a workstation; this is much faster than human coding. The output code is efficient, often better than code produced by human programmers. Because of the efficiency of the generated code, we consider specialized compilation of generic procedures to be the best way to achieve reuse using views.

Our approach has several significant advantages:

1. The user interface is self-documenting and allows views to be created quickly and easily.

2. Specialized versions of generic algorithms for an application can be created in seconds.

3. The specialized algorithms can be produced in a standard programming language that is independent of our system.

4. The code that is produced is optimized and efficient.

5. Static error checking is performed at compile time.

Related techniques have been used to create programs by connecting diagrammatic representations of physical and mathematical laws [36].

The sizes of the various components of the system, in non-comment lines of Lisp source code, are shown in the following table:

Component:	**Lines:**
Make Views: `mkv`	265
Symbolic Algebra	1,003
Graphical Interface	1,020
GLISP Compiler	9,097
Translation to C	831
Total	12,216

The Symbolic Algebra component includes algorithms used by `mkv` and described in this paper. The amount of code required for making views is not too large. Translation of specialized code into languages other than C should not be difficult: the C translation component is not large, and much of it could be reused for other languages; the translation uses patterns that are easily changed.

Acknowledgments

Computer equipment used in this research was furnished by Hewlett Packard and IBM.

I thank the anonymous reviewers for their helpful suggestions for improving this paper.

References

[1] *IEEE Trans. on Software Engineering*, vol. 11, no. 11, Nov. 1985.

[2] A. Berlin and D. Weise, "Compiling Scientific Code Using Partial Evaluation," *IEEE Computer*, vol. 23, no. 12, pp. 25-37, Dec. 1990.

[3] W. H. Beyer (ed.), *CRC Standard Mathematical Tables and Formulae*, 29th ed., Boca Raton, FL: CRC Press, 1991.

[4] T. Biggerstaff and A. Perlis (eds), *Software Reusability* (2 vols.), ACM Press / Addison-Wesley, 1989.

[5] J. Callahan and J. Purtilo, "A Packaging System for Heterogeneous Execution Environments," *IEEE Trans. Software Engineering*, vol. 17, no. 6, pp. 626-635, June 1991.

[6] "The Common Object Request Broker: Architecture and Specification," TC Document 91.12.1, Revision 1.1, Object Management Group, Dec. 1991.

[7] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

[8] S. Efremidis, "On Program Transformations," Ph.D. dissertation, Tech. Report TR-94-1434, Dept. of Computer Science, Cornell Univ., 1994.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[10] D. Garlan, "Views for Tools in Integrated Environments," in R. Conradi *et al.* (eds.), Lecture Notes in Computer Science, vol. 244: *Advanced Programming Environments*, Springer-Verlag, 1986.

[11] K. Gieck, *Engineering Formulas* (5th ed.), McGraw-Hill, 1986.

[12] J. A. Goguen, "Reusing and Interconnecting Software Components," *IEEE Computer*, pp. 16-28, Feb. 1986.

[13] J. A. Goguen, "Principles of Parameterized Programming," in [4], pp. 159-225.

[14] J. A. Goguen, "Introducing OBJ," Technical Report SRI-CSL-92-03, Computer Science Lab, SRI International, Menlo Park, CA, March 1992.

[15] M. J. C. Gordon, *The Denotational Description of Programming Languages*, Springer-Verlag, 1979.

[16] D. Gries and J. Prins, "A New Notion of Encapsulation," *ACM SIGPLAN Notices*, vol. 20, no. 7), pp. 131-139, July 1985.

[17] D. Gries and D. Volpano, "The Transform – a New Language Construct," *Structured Programming*, vol. 11, no. 1, pp. 1-10, 1990.

[18] B. Hailpern and H. Ossher, "Extending Objects to Support Multiple Interfaces and Access Control," *IEEE Trans. Software Engineering*, vol. 16, no. 11, pp. 1247-1257, Nov. 1990.

[19] W. Harrison, "RPDE^3: A Framework for Integrating Tool Fragments," *IEEE Software*, vol. 4, no. 6, pp. 46-56, Nov. 1987.

[20] M. Herlihy and B. Liskov, "A Value Transmission Method for Abstract Data Types," *ACM Trans. Programming Languages Syst.*, vol. 4, no. 4, pp. 527-551, Oct. 1982.

[21] R. F. Hille, *Data Abstraction and Program Development using Modula-2*, Prentice Hall, 1989.

[22] G. Kaiser and D. Garlan, "Synthesizing Programming Environments from Reusable Features," in [4], pp. 35-55.

[23] E. Kant, F. Daube, W. MacGregor, and J. Wald, "Scientific Programming by Automated Synthesis," in [28], pp. 169-205.

[24] C. W. Krueger, "Software Reuse," *ACM Computing Surveys*, vol. 24, no. 2, pp. 131-184, June 1992.

[25] D. Lamb, "IDL: Sharing Intermediate Representations", *ACM Trans. Programming Languages Syst.* vol. 9, no. 3, pp. 267-318, July 1987.

[26] J. Larkin and H. A. Simon, "Why a Diagram is (Sometimes) Worth 10,000 Words," *Cognitive Science*, **11:**65-99, 1987; also in [44].

[27] C. Lins, *The Modula-2 Software Component Library*, Springer-Verlag, 1989.

[28] M. Lowry and R. McCartney, eds., *Automating Software Design*, AAAI Press / MIT Press, 1991.

[29] H. Mili, F. Mili, and A. Mili, "Reusing Software: Issues and Research Directions," *IEEE Trans. Software Engineering*, vol. 21, no. 6, pp. 528-562, June 1995.

[30] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, P. Senator, and W. Swartout, "Enabling Technology for Knowledge Sharing," *AI Magazine*, vol. 12, no. 3, pp. 36-56, Fall 1991.

[31] G. Novak, "GLISP: A LISP-Based Programming System With Data Abstraction," *AI Magazine*, vol. 4, no. 3, pp. 37-47, Fall 1983.

[32] G. Novak, "GLISP User's Manual," Tech. Report STAN-CS-82-895, C.S. Dept., Stanford Univ., 1982; TR-83-25, A.I. Lab, C.S. Dept., Univ. of Texas at Austin.

[33] G. Novak, F. Hill, M. Wan, and B. Sayrs, "Negotiated Interfaces for Software Reuse," *IEEE Trans. Software Engineering*, vol. 18, no. 7, July 1992.

[34] G. Novak, "Software Reuse through View Type Clusters," *Proc. Seventh Knowledge-Based Software Eng. Conf.*, IEEE CS Press, pp. 70-79, Sept. 1992.

[35] G. Novak, "Software Reuse by Compilation through View Type Clusters," submitted for publication.

[36] G. Novak, "Generating Programs from Connections of Physical Models," *Proc. 10th Conf. on Artificial Intelligence for Applications*, IEEE CS Press, Mar. 1994, pp. 224-230.

[37] G. Novak, "Conversion of Units of Measurement," *IEEE Trans. Software Engineering*, vol. 21, no. 8, pp. 651-661, Aug. 1995.

[38] L. C. Paulson,, *ML for the Working Programmer*, Cambridge Univ. Press, 1991.

[39] F. Preparata and R. Yeh, *Introduction to Discrete Structures*, Addison-Wesley, 1973.

[40] J. M. Purtilo and J. M. Atlee, "Module Reuse by Interface Adaptation," *Software Practice and Experience*, vol. 21, no. 6, pp. 539-556, June 1991.

[41] C. Rich and R. Waters (eds), *Readings in Artificial Intelligence and Software Engineering*, San Francisco: Morgan Kaufmann, 1986.

[42] C. Rich and R. Waters, *The Programmer's Apprentice*, ACM Press, 1990.

[43] M. Schaefer, *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, 1973.

[44] H. A. Simon, *Models of Thought*, vol. 2, Yale Univ. Press, 1989.

[45] D. R. Smith, "KIDS: A Semiautomatic Program Development System," *IEEE Trans. Software Engineering*, vol. 16, no. 9, pp. 1024-1043, Sept. 1990.

[46] D. R. Smith, "KIDS – A Knowledge-based Software Development System," in [28], pp. 483-514.

[47] G. Steele, *Common Lisp: the Language*, Digital Press, 1990.

[48] B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1991.

[49] W. Tracz, "LILEANNA: A parameterized programming language," *2nd Int'l Workshop on Software Reuse: Advances in Software Reusability (IWSR-2)*, IEEE CS Press, Mar. 1993, pp. 66-78.

[50] D. Turner, "An Overview of Miranda," *SIGPLAN Notices*, Dec. 1986.

[51] D. Volpano. and R. Kieburtz, "The Templates Approach to Software Reuse," in [4], pp. 247-255.

[52] B. W. Weide, W. F. Ogden, and S. H. Zweben, "Reusable Software Components," in [55], pp. 1-65.

[53] Å. Wikström, *Functional Programming Using Standard ML*, Prentice-Hall, 1987.

[54] S. Wolfram, *Mathematica: a System for Doing Mathematics by Computer*, Addison Wesley, 1991.

[55] M. C. Yovits, ed., *Advances in Computers*, vol. 33, Academic Press, 1991.