Software Reuse by Specialization of Generic Procedures through Views

Gordon S. Novak Jr. *

August 1, 1997

This article appears in *IEEE Transactions on Software Engineering*, vol. 23, no. 7, July 1997.

Abstract

A generic procedure can be specialized, by compilation through views, to operate directly on concrete data. A view is a computational mapping that describes how a concrete type implements an abstract type. Clusters of related views are needed for specialization of generic procedures that involve several types or several views of a single type. A user interface that reasons about relationships between concrete types and abstract types allows view clusters to be created easily. These techniques allow rapid specialization of generic procedures for applications.

Index Terms – software reuse, view, generic algorithm, generic procedure, algorithm specialization, partial evaluation, direct-manipulation editor, abstract data type.

1 Introduction

Reuse of software has the potential to reduce cost, increase the speed of software production, and increase reliability. Facilitating the reuse of software could therefore be of great benefit.

*G. S. Novak, Jr. is with the Department of Computer Sciences, University of Texas, Austin, TX 78712. An Automatic Programming Server demonstration of this software is available on the World Wide Web via `http://www.cs.utexas.edu/users/novak`; running the demo requires X windows.

Rigid treatment of type matching presents a barrier to reuse. In most languages, the types of arguments of a procedure call must match the types of parameters of the procedure. For this reason, reuse is often found where type compatibility occurs naturally, i.e. where the types are basic or are made compatible by the language (e.g. arrays of numbers).

A truly generic procedure should be reusable for *any* reasonable implementation of its abstract types; a developer of a generic should be able to advertise "my program will work with your data" without knowing what the user's data representation will be. We seek reuse without conformity to rigid standards. We envision two classes of users: *developers*, who understand the details of abstract types and generic procedures, and *users*, programmers who reuse the generics but need less expertise and understanding of details. Developers produce libraries of abstract types and generics; by specializing the generics, users obtain program modules for applications.

A *view* provides a mapping between a concrete type and an abstract type[1] in terms of which a generic algorithm is written. Fig. 1 illustrates schematically that a view acts as an interface adapter that makes the concrete type appear as the abstract type[2]. The view provides a clean separation between the semantics of data (as represented by the abstract type) and its implementation, so that the implementation is minimally constrained. Once a view has been made, any generic procedure associated with the abstract type can be automatically specialized for the concrete type, as shown in Fig. 2. In our implementation, the specialized procedure is a Lisp function; if desired, it can be mechanically translated into another language. Tools exist that make it easy to create views; a programmer can obtain a specialized procedure to insert records into an AVL tree (185 lines of C) in a few minutes.

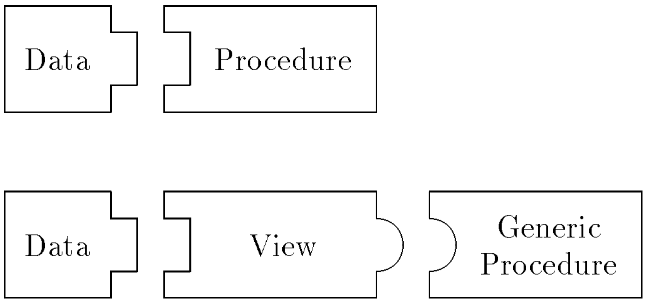

Figure 1: Interfacing with Strong Typing and with Views

[1] We consider an abstract type to be a set of *basis variables* and a set of generic procedures that are written in terms of the basis variables.

[2] Goguen [18] and others have used a similar analogy and diagram.

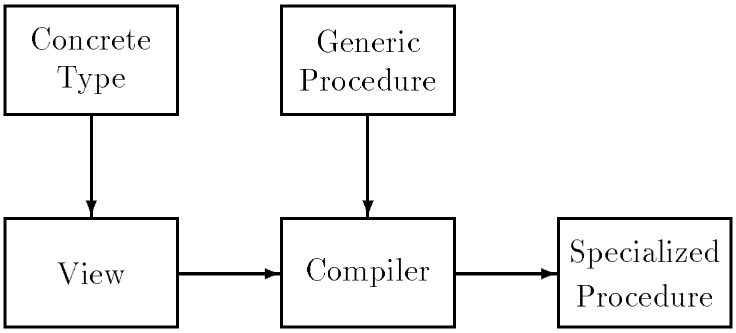

Figure 2: Specialization of Generic Procedure through View

This approach to reuse has several advantages:

1. It provides freedom to select the implementation of data; data need not be designed *ab initio* to match the algorithms.
2. Several views of a data structure can correspond to different aspects of the data.
3. Several languages are supported. Lisp, C, C++, Java, or Pascal can be generated from a single version of generic algorithms.
4. Tools simplify the specification of views and reduce the learning required to reuse software.
5. Views can be used to automatically:
   (a) specialize generic procedures from a library,
   (b) instantiate a program framework from components,
   (c) translate data from one representation to another,
   (d) generate methods for object-oriented programming, and
   (e) interface to programming tools for data display and editing.

This paper describes principles of views and specialization of generic algorithms, as well as an implementation of these techniques using the GLISP language and compiler; GLISP is a Lisp-based language with abstract data types.

Section 2 describes in conceptual terms how views provide mappings between concrete types and abstract types. Section 3 describes the GLISP compiler and how views are used in specializing generic algorithms. Section 4 discusses *clusters* of related views that are needed to reuse generic algorithms that involve several types or several views of a type. Section 5 describes the program VIEWAS, which reasons about relations between concrete types and abstract types and makes it easy to create view clusters. Section 6 describes higher-order code and a generic algorithm for finding a convex hull that uses other generics and uses several views of a single data structure. Section 7 describes use of views with object-oriented programming. Section 8 surveys related work, and Section 9 presents conclusions.

2 Views

2.1 Computation as Isomorphism

It is useful to think of all computation as simulation or, more formally, as isomorphism. This idea has a long history; for example, isomorphism is the basis of denotational semantics [20]. Goguen [18] [19] describes views as isomorphisms, with mappings of unary and binary operators. Our views allow broader mappings between concrete and abstract types and include algorithms as well as operators. We use isomorphism to introduce the use of views.

Preparata and Yeh [56] give a definition and diagram for isomorphism of semigroups:

> Given two semigroups $G_1 = [S, \circ]$ and $G_2 = [T, *]$, an invertible function $\varphi : S \to T$ is said to be an *isomorphism* between G_1 and G_2 if, for every a and b in S, $\varphi(a \circ b) = \varphi(a) * \varphi(b)$.

$$
\begin{tikzcd}
a \arrow[rr, "\varphi"] \arrow[dd] & & \varphi(a) \arrow[dd] & \\
& b \arrow[rr, "\varphi"] \arrow[dl] & & \varphi(b) \arrow[dl] \\
\circ \arrow[d] & & * \arrow[d] & \\
a \circ b & & \varphi(a) * \varphi(b) \arrow[ll, "\varphi^{-1}"'] &
\end{tikzcd}
$$

Since φ is invertible, there is a *computational* relationship: $a \circ b = \varphi^{-1}(\varphi(a) * \varphi(b))$. If $a \circ b$ is difficult to obtain directly, its value can be computed by encoding a and b using φ, performing the computation $\varphi(a) * \varphi(b)$, and decoding or interpreting the result using the inverse φ^{-1}. A diagram is said to *commute* [2] if the same result is obtained regardless of which path between two points is followed, as shown in the diagram above.

2.2 Views as Isomorphisms

Reuse of generic algorithms through views corresponds to computation as isomorphism. The concrete type corresponds to the left-hand side of an isomorphism diagram; the view maps the concrete type to the abstract type. The generic algorithm corresponds to an operation on the abstract type. By mapping from the concrete type to the abstract type, performing the operation on the abstract type, and mapping back, the result of performing the algorithm on the concrete type is obtained. However, instead of performing the view mapping explicitly

$$
\begin{tikzcd}[column sep=large]
\text{Concrete Type} \arrow[r, "view"] & \text{Abstract Type} \arrow[d] & & \text{Concrete Type} \arrow[d] \\
& \text{Generic Algorithm} \arrow[d] \arrow[r, phantom, "\overset{\textit{Compilation, Optimization}}{\Longrightarrow}"] & & \text{Specialized Algorithm} \arrow[d] \\
\text{Concrete Result} & \text{Abstract Result} \arrow[l, "view^{-1}"'] & & \text{Concrete Result}
\end{tikzcd}
$$

Figure 3: Specializing a Generic

and materializing the abstract data, the mappings are folded into the generic algorithm to produce a specialized version that operates directly on the concrete data (Fig. 3).

As an example, let concrete type *pizza* contain a value d that represents the diameter of the (circular) pizza. Suppose abstract type *circle* assumes a radius value r. A view from *pizza* to *circle* will specify that r corresponds to $d/2$. A simple generic procedure that calculates the *area* of a *circle* can then be specialized by compilation through the view. Because the view mapping is folded into the generic algorithm and optimized, the specialized algorithm operates directly on the original data and, in this case, does no extra computation (Fig. 4). Code to reference d in data structure *pizza* is included in the specialized code.

$$
\begin{tikzcd}[column sep=large]
pizza \arrow[r, "r = \frac{d}{2}"] & circle \arrow[d] & & pizza \arrow[d] \\
& area = \pi \cdot r^2 \arrow[d] \arrow[r, phantom, "\overset{\textit{Compilation, Optimization}}{\Longrightarrow}"] & & area = \frac{\pi}{4} \cdot d^2 \arrow[d] \\
area & area \arrow[l] & & area
\end{tikzcd}
$$

Figure 4: Example Specialization

2.3 Abstract Data Types and Views

An *abstract type* is an abstraction of a set of concrete types; it assumes an abstract record containing a set of *basis variables*[3] and has a set of named *generic procedures* that are written in terms of the basis variables.[4]

Any data structure that contains the basis variables, with the same names and types, implements the abstract type. To maximize reuse, constraints on the implementation must be minimized: it should be possible to specialize a generic procedure for *any legitimate implementation* of its abstract type.

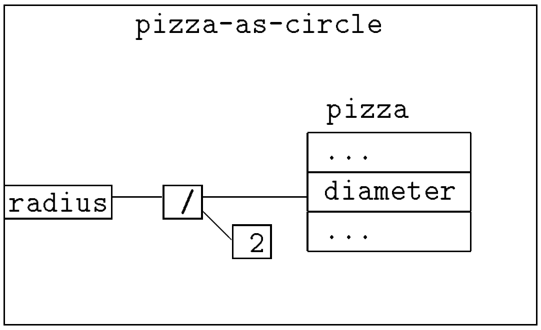

Figure 5: Encapsulation of Concrete Type by View

A view encapsulates a concrete type and presents an external interface that consists of the basis variables of the abstract type. Fig. 5 illustrates how view type `pizza-as-circle` encapsulates `pizza` and presents an interface consisting of the `radius` of abstract type `circle`. The `radius` is implemented by dividing field `diameter` of `pizza` by 2; other fields of `pizza` are hidden. Code for this example is shown in the following section. In the general case, the interface provides the ability to read and write each basis variable. A read or write may be implemented as access to a variable of the concrete record or by a procedure that emulates the read or write, using the concrete record as storage.[5]

A view implements an abstract type if it emulates a record containing the basis variables. Emulation is expressed by two properties:

1. *Storage:* After a value z is stored into basis variable v, reference to v will yield z.

[3] Although it is not required, our abstract types usually specify a concrete record containing the basis variables; this is useful as an example and for testing the generic procedures.

[4] This definition of abstract type is different from the algebraic description of abstract types [11] as a collection of abstract sorts, procedure signatures, and axioms. In the algebraic approach, an abstract type is described without regard to its implementation. In our approach, an abstract implementation is assumed because the abstract type has generic procedures that implement operations.

[5] Views can be implemented in an object-oriented system by *adapter* or *wrapper* objects [16], where a wrapper contains the concrete data, presents the interface of the abstract type, and translates messages between the abstract and concrete types. Our views give the effect of wrappers without creating them.

2. *Independence:* If a reference to basis variable v yields the value z and a value is then stored into some other basis variable w, a reference to v will still yield z.

These properties express the behavior expected of a record: stored values can be retrieved, and storing into one field does not change the values of others.

If a view implements an abstract type exactly, as described by the storage and independence properties, then any generic procedure will operate in the same way (produce the same output and have the same side effects) when operating on the concrete data through the view as it does when operating on a record consisting of the basis variables. That is, an isomorphism holds between the abstract type and concrete type, and its diagram commutes. This criterion is satisfied by the following variations of data:

1. Any record structure may be used to contain the variables. [6]

2. Names of variables may differ from those of the abstract type: views provide name translation, and the name spaces of the concrete and abstract types are distinct.

Some generics use only a subset of basis variables; only those that are used must be defined in a view. An attempt to use an undefined basis variable is detected as an error.

A view in effect defines functions to compute basis variables from the concrete variables; if a generic procedure is to "store into" basis variables, these functions must be invertible. Simple functions can be inverted automatically by the compiler. For more complex cases, a procedure can be defined to effect a store into a basis variable. The procedures required for mathematical views may be somewhat complex: in the case of a polar vector (r, θ), where the abstract type is a Cartesian vector (x, y), an assignment to basis variable x must update both r and θ so that x will have the new value and y will be unchanged. A program MKV ("make view") [54] allows a user to specify mathematical views graphically by connecting corresponding parts of the concrete type and a diagram associated with the abstract type; MKV uses symbolic algebra to derive view procedures from the correspondences.

For wider reuse, the storage and independence properties must be relaxed slightly. Even a simple change of representation, such as division of the diameter value by 2 in the *pizza* example, changes the point at which numerical overflow occurs; there could also be round-off error. Significant changes of representation should be allowed, such as representing a vector in polar coordinates (r, θ) when the basis variables are in Cartesian coordinates (x, y). If a polar vector is viewed as a Cartesian vector using transformations $x = r \cdot cos(\theta)$ and $y = r \cdot sin(\theta)$, the mapping is not exact due to round-off error, nor is it one-to-one; however, it is sufficiently accurate for many applications. Ultimately, the user of the system must ensure that the chosen representation is sufficiently accurate.

In some cases, a user might want to specify a contents type and let the system define a record using it, e.g. an AVL tree containing strings. This is easily done by substituting the contents type into a prototype record definition with the view mappings predefined.

The next section describes how views are implemented and compiled in GLISP.

[6] This could include arrays, or sub-records reached by a fixed sequence of pointer traversals.

3 GLISP Language and Compiler

GLISP [46, 47, 48, 49] ("Generic Lisp"), a high-level language with abstract data types, is compiled into Lisp; it has a language for describing data in Lisp and in other languages. GLISP is described only briefly here; for more detail, see [49] and [46].

3.1 Data-independent Code

A GLISP type is analogous to a class in object-oriented programming (OOP); it specifies a data structure and a set of methods. For each method, there is a name (selector) and a definition as an expression or function. As in OOP, there is a hierarchy of types; methods can be inherited from ancestor types. The methods of abstract types are generic procedures.

In most languages, the syntax of program code depends on the data structures used; this prevents reuse of code for alternative implementations of data. GLISP uses a single Lisp-like syntax. In Lisp, a function call is written inside parentheses: `(sqrt x)`. A similar syntax (*feature object*) is used in GLISP to access any feature of a data structure [49]:

1. If *feature* is the name of a field of the type of *object*, data access is compiled.
2. If *feature* is a method name (selector) of the type of *object*, the method call is compiled.
3. If *feature* is the name of a view of the type of *object*, the type of *object* is locally changed to the view type.
4. If *feature* is a function name, the code is left unchanged.
5. Otherwise, a warning message is issued that *feature* is undefined.

This type-dependent compilation allows variations in data representation: the same code can be used with data that is stored for one type but computed for another type. For example, type `circle` can assume `radius` as a basis variable, while a `pizza` object can store `diameter` and compute `radius`.

The GLISP compiler performs type inference as it compiles expressions. When the type of an object is known at compile time, reference to a feature can be compiled as in-line code or as a call to a specialized generic. Specialized code depends on the types of arguments to the generic. Compilation by in-line expansion and specialization is *recursive at compile time* and propagates types during the recursion; this is an important feature. Recursive expansion allows a small amount of source code to expand into large output code; it allows generic procedures to use other generics as subroutines and allows higher-order procedures to be expanded through several levels of abstraction until operations on data are reached.

Symbolic optimization folds operations on constants [62], performs partial evaluation [7] [12] and mathematical optimization, removes dead code, and combines operations to improve efficiency. It provides conditional compilation, since a conditional is eliminated when the test can be evaluated at compile time. Optimization often eliminates operations associated with a view, so that use of the view has little or no cost after compilation.

3.2 Views in GLISP

A view [46, 49, 50] is expressed as a GLISP type whose record is the concrete type. The abstract type is a superclass of the view type, allowing generics to be inherited [7]. The view type encapsulates the concrete type and defines methods to compute basis variables of the abstract type. As specialized versions of generics are compiled, the compiler caches them in the view type. Examples of abstract type `circle`, concrete type `pizza`, and view type `pizza-as-circle` are shown below; each gives the name of the type followed by its data structure, followed by method (`prop`), view, and superclass specifications.

```
(circle  (list (center vector) (radius real))
  prop   ( (area (pi * radius ^ 2)) ) )

(pizza (cons (diameter real) (topping symbol))
  views  ((circle pizza-as-circle)) )

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

`pizza-as-circle` encapsulates `pizza` and makes it appear as a `circle`; its record is named `p` and has type `pizza`. It defines basis variable `radius` as the `diameter` of `p` divided by 2 and specifies `circle` as a superclass; it hides other data and methods of `pizza` [8]. The following example shows how `area` defined in `circle` is compiled through the view; GLISP function `t1` is shown followed by compiled code in Lisp.

```
(gldefun t1 (pz:pizza) (area (circle pz)))

result type: REAL
(LAMBDA (PZ) (* 0.78539816 (EXPT (CAR PZ) 2)))
```

The code `(circle pz)` changes the type of `pz` to the view type `pizza-as-circle`. The `area` method is inherited from `circle` and expanded in-line; basis variable `radius` is expanded using `diameter`, which becomes a data access `(CAR PZ)`.

If a view defines all basis variables in terms of the concrete type, then any generic procedure of the abstract type can be used through the view. Because compilation by GLISP is recursive, generic procedures can be written using other generics as subroutines, as

[7] Only methods are inherited; data structures, and therefore state variables, are not.

[8] `pizza-as-circle` fails to define the basis variable `center`; this is allowable. An attempt to reference an undefined basis variable is detected as an error.

long as the recursion terminates at compile time.[9] A view type may redefine some methods that are generics of the abstract type; this may improve efficiency. For example, a Cartesian vector defines `magnitude` as a generic, but this value is stored as r in a polar (r, θ) vector.

When a basis variable is assigned a value, the compiler produces code as follows:

1. If the basis variable corresponds to a field of the concrete type, a store is generated.
2. If the basis variable is defined by an expression that can be inverted algebraically, the compiler does so. For example, assigning a value r to the `radius` of a `pizza-as-circle` causes $r * 2$ to be stored into the `diameter` of record `pizza`.
3. A procedure can be defined in the view type to accomplish assignment to a basis variable while maintaining the storage and independence properties. MKV [54] produces such procedures automatically.

A view can define a procedure to create an instance of the concrete type from a set of basis variables of the abstract type [54]. This is needed for generics that create new data, e.g. when two vectors are added to produce a new vector.

Several points about views are worth noting:

1. In general, it is not the case that an object *is* its view; rather, a view represents some aspect of an object. The object may have other data that are not involved in the view.
2. A view provides name translation. This removes any necessity that concrete data use particular names and eliminates name conflicts.
3. A view can specify representation transformation.
4. There can be several ways of viewing a concrete type as a given abstract type. For example, the same records might be sorted in several ways for different purposes.

4 Clusters of Views

Several languages (e.g. Ada, Modula-2, ML, and C++) provide a form of abstract data type that is like a macro: an instance is formed by substituting a concrete type into it, e.g. to make a linked list whose contents is a user type. This technique allows only limited software reuse. We seek to extend the principle that a generic should be reusable for *any reasonable implementation* of its data to generics that involve several abstract types.

Some data structures that might be regarded as a single concept, such as a linked list, involve several types: a linked list has a record type and a pointer type. Many languages finesse the need for two types by providing a pointer type that is derived from the record

[9] Recursion beyond a certain depth is trapped and treated as a compilation error.

type. In general, however, a pointer can be any data that uniquely denotes a record in a memory: a memory address, a disk address, an array index, an employee number, etc. To maximize generality, the record and pointer must be treated as distinct types.

A view maps a single concrete type to a single abstract type. A *cluster* collects a set of views that are related because they are used in a generic algorithm. For example, a polygon can be represented as a sequence of points; the points could be Cartesian, polar or some type that could be viewed as a point (e.g. a city), and the sequence could be a linked list, array, etc. There should not be a different generic for each combination of types; a single generic should be usable for any combination. A cluster collects the views used by a generic algorithm in a single place, allows inheritance and specialization of generics through the views, and is used in type inference.

A cluster has a set of *roles*, each of which has a name and a corresponding view type; for example, cluster `linked-list` has roles named `record` and `pointer`. A cluster may have super-clusters; each view type that fills a role specifies as a superclass the type that fills the same role in the super-cluster, allowing inheritance of methods from it. The view types also define methods or constants[10] needed by generic procedures; for example, cluster `sorted-linked-list` requires specification of the field or property of the record on which to sort and whether the sort is `ascending` or `descending`.

4.1 Example Cluster: Sorted Linked List

This section gives an example record, shows how a cluster is made for it using VIEWAS, and shows how a generic is specialized. We begin by showing the user interaction with VIEWAS to emphasize its ease of use; a later section explains how VIEWAS works.

The C structure of example record `myrec` and its corresponding GLISP type are shown below.[11]

```
typedef struct myrecord {
  int  color;
  char *name;
  int  size;
  struct myrecord *next;
  } MYREC;

(myrec (crecord myrec
                (color integer)
                (name  string)
                (size  integer)
                (next  (^ myrec)) ) )
```

[10]A constant is specified as a method that returns a constant value.

[11]The GLISP type could be derived automatically from the C declaration, but this is not implemented.

Suppose the user wishes to view `myrec` as a `sorted-linked-list` and obtain specialized versions of generics for an application. The user invokes VIEWAS to make the view cluster:

```
(viewas 'sorted-linked-list 'myrec)
```

VIEWAS determines how the concrete type matches the abstract type; it makes some choices automatically and asks the user for other choices: [12]

```
Choice for LINK = NEXT

Specify choice for SORT-VALUE
Choices are: (COLOR NAME SIZE)
name

Specify choice for SORT-DIRECTION
Choices are: (ASCENDING DESCENDING)
ascending
```

VIEWAS chooses field `next` as the `link` of the linked-list record since it is the only possibility; it asks the user for the field on which to sort and the direction of sorting. VIEWAS requires only a few simple choices by the user; the resulting cluster `MYREC-AS-SLL` and two view types are shown in Fig. 6. Cluster `MYREC-AS-SLL` has roles named `pointer` and `record` that are filled by corresponding view types; `MYREC-AS-SLL` lists cluster `SLL` (sorted linked list) as a super-cluster.

View type `MYREC-AS-SLL-POINTER` is a pointer to view type `MYREC-AS-SLL-RECORD`; it has the corresponding type `SLL-POINTER` of cluster `SLL` as a superclass. The generics of `SLL` are defined as methods of `SLL-POINTER`. View type `MYREC-AS-SLL-RECORD` has data named `Z16`[13] of type `MYREC`; it lists type `SLL-RECORD` as a superclass and defines the `LINK` and `COPY-CONTENTS-NAMES` needed for linked-list generics and `SORT-VALUE` and `SORT-DIRECTION` needed for sorted-linked-list generics.

After making a view cluster, the user can obtain specialized versions of generics. We do not expect that a user would read the code of generics or the specializations derived from them, but we present a generic and its specialization here to illustrate the process.

Fig. 7 shows generic `sll-insert`; it uses generics `rest` defined for `linked-list` (the value of field `link`) and `sort-before`. The notation `(^. ptr)` is short for `dereference` of pointer `ptr`. `sort-direction` is tested in this code; however, since this is constant at compile time, the compiler eliminates the `if` and keeps only one `sort-before` test, which is expanded depending on the type of `sort-value`.

[12] Lisp converts symbols to upper-case, so upper-case and lower-case representations of the same symbol are equivalent. In general, user input is shown in lower-case, while Lisp output of symbols is upper-case.

[13] Names with numeric suffixes are new, unique names generated by the system. The unique name `Z16` encapsulates `MYREC` and prevents name conflicts in the view type because features of `MYREC` can be accessed only via that name.

Cluster MYREC-AS-SLL:

```
(GLCLUSTERDEF
  (ROLES ((POINTER MYREC-AS-SLL-POINTER)
          (RECORD MYREC-AS-SLL-RECORD))
         SUPERS (SLL)) )
```

View type MYREC-AS-SLL-POINTER:

```
((^ MYREC-AS-SLL-RECORD)
  SUPERS (SLL-POINTER))
GLCLUSTER MYREC-AS-SLL
```

View type MYREC-AS-SLL-RECORD:

```
((Z16 MYREC)
  SUPERS (SLL-RECORD)
  PROP   ((LINK  ((NEXT Z16))
                  RESULT
                  MYREC-AS-SLL-POINTER)
          (COPY-CONTENTS-NAMES
                  ('(COLOR NAME SIZE)))
          (SORT-VALUE     ((NAME Z16)))
          (SORT-DIRECTION ('ASCENDING))))
GLCLUSTER MYREC-AS-SLL
```

Figure 6: Cluster and View Types

Specialization of a generic is performed when a function that calls it is compiled. Such a function can be defined by the user or created by the user interface; the following function specifies insertion of a new record into a sorted linked list:

```
(gldefun t2 (b:myrec new:myrec)
  (insert (sorted-linked-list b)
          (sorted-linked-list new)))
```

When this function is compiled, the compiler specializes generic `sll-insert` (Fig. 7) to produce `sll-insert-1`, which is mechanically converted to the target language (Fig. 8).

There can be several view clusters of the same kind for a given concrete type; for example, the same records could be used in sorted linked lists that are sorted in different ways, or a record could contain several pointer fields, each used in a different linked list. At present, the programmer must insure that multiple views do not conflict, e.g. a list cannot be sorted in two different ways at the same time.

```
; insert record new into sorted linked list lst
(gldefun sll-insert (lst:sll-pointer new:(typeof lst))
  (let ((ptr lst) prev)
     (prev := nil)
;    find the proper insertion point
;    invariant: all records before ptr sort before new
     (while ptr is not null and
            (if (sort-direction (^. lst)) = 'ascending
                then (sort-before (^. ptr) (^. new))
                else ~(sort-before (^. ptr) (^. new)))
         do (prev := ptr)
            (ptr := (rest ptr)))
;    splice in new before ptr
     ((rest new) := ptr)
     (if prev then ((rest prev) := new)
                   lst
              else new)))
```

Figure 7: Generic: Insert into Sorted Linked List

4.2 Uses of Clusters

Clusters serve several goals:

1. Clusters allow independent specification of the several views used in a generic.
2. A generic that performs a given function should be written only once; generics should reuse other generics when possible. Clusters allow generics to be inherited.
3. Clusters are used to derive the correct view types as generics are specialized.

4.2.1 Inheritance through Clusters

It is desirable to inherit and reuse generics when possible. In some cases, a cluster can be considered to be a specialization of another cluster, e.g. sorted-linked-list is a specialization of linked-list. Some generics defined for linked-list also work for a sorted-linked-list: the `length` of a linked-list is the same whether it is sorted or not. Generics should be defined at the highest possible level of abstraction to facilitate reuse.

A cluster can specify super-clusters. Fig. 9 shows the inheritance among clusters for the `MYREC-AS-SLL` example; each cluster can inherit generics from the clusters above.

The mechanism of inheritance between clusters is simply inheritance between types. Each type that fills a role in a cluster lists as a superclass the type that fills the corresponding

```
MYREC *sll_insert_1 (lst, new)
  MYREC *lst, *new;
  {
    MYREC *ptr, *prev;
    ptr = lst;
    prev = 0;
    while ( ptr &&
            strcmp(ptr->name, new->name)
               < 0 )
        {
          prev = ptr;
          ptr = ptr->next;
        }
    new->next = ptr;
    if (prev != 0)
        {
          prev->next = new;
          return lst;
        }
    else
      return new;
  }
```

Figure 8: Specialized Procedure in C

role in the super-cluster. These inheritance paths are specified manually for abstract types; VIEWAS sets up the inheritance paths when it creates view clusters.

Inheritance provides defaults for generic procedures and constants. For example, generics of `sorted-linked-list` use predicate `sort-before` to compare records; a generic `sort-before` is defined as `<` applied to the `sort-value` of records. Predicate `<`, in turn, depends on the type of the `sort-value`, e.g., string comparison is used for strings. A minimal specification of a `sorted-linked-list` can use the default sorting predicate, but an arbitrary `sort-before` predicate can be specified in the `record` view type if desired.

In some cases, inheritance of generics from super-clusters should be preventable. For example, `nreverse` (destructive reversal of the order of elements in a linked list) is defined for linked-list but should not be available for a sorted-linked-list, since it destroys the sorting order. Prevention of inheritance can be accomplished by defining a method as `error`, in which case an attempt to use the method is treated as a compilation error.

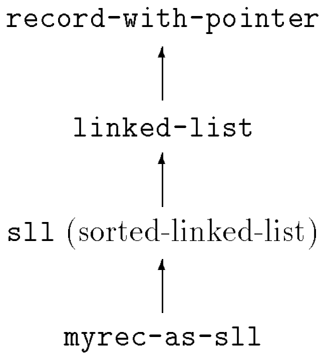

Figure 9: Inheritance between Clusters

4.2.2 Type Mappings

A cluster specifies a set of related types. A generic procedure is specified in terms of abstract types, but when it is specialized, the corresponding view types must be substituted. For example, at the abstract level, a `linked-list-record` contains field `link` whose type is `linked-list-pointer`, and `dereference` of a `linked-list-pointer` yields a `linked-list-record`. When a generic defined for `linked-list` is specialized, these types must be replaced by the corresponding view types of the cluster. Care is required in defining generics and view types to ensure that each operation produces a result of the correct type; otherwise, the generic will not compile correctly when specialized.

In general, if a generic function $f : a_1 \rightarrow a_2$, with abstract argument and result types is specialized for concrete types t_1 and t_2 using views $v_1 : t_1 \rightarrow a_1$ and $v_2 : t_2 \rightarrow a_2$, the specialized function must have signature $f_s : v_1(t_1) \rightarrow v_2(t_2)$. Smith [68] uses the term *theory morphism* for a similar notion. Dijkstra [15] uses the term *coordinate transformation* for a similar notion, in which some variables are replaced by others; Gries [23] uses the term *coupling invariant* for the predicate that describes the relation (between the abstract types or variables and their concrete counterparts) that is to be maintained by functions. Consider the following generic:

```
(gldefun generic-cddr (l:linked-list)
  (rest (rest l)) )
```

`generic-cddr` follows the `link` of a linked-list record twice: `rest` is the `link` value. [14] Now suppose that a concrete record has two pointer fields, so that two distinct linked-list clusters can be made using the two pointer fields. To specialize `generic-cddr` for both, `rest` must produce the view type, which defines the correct `link`, rather than the concrete type.

Fig. 10 abstractly illustrates type mappings as used in the `generic-cddr` example. The figure shows concrete types t_i that are viewed as abstract types a_i by views v_i. Suppose

[14] `rest` (or `cdr`) and `cddr` are Lisp functions; we use Lisp names for linked-list generics that are similar.

$$
\begin{tikzcd}
t_1 \arrow[r, "v_1"] \arrow[d, "f_s"] & a_1 \arrow[d, "f"] \\
t_2 \arrow[r, "v_2"] \arrow[d, "g_s"] & a_2 \arrow[d, "g"] \\
t_3 \arrow[r, "v_3"] & a_3
\end{tikzcd}
$$

Figure 10: Cluster: Views between Type Families

generic function $f : a_1 \rightarrow a_2$ is composed with function $g : a_2 \rightarrow a_3$. The corresponding specialized functions will be $f_s : v_1(t_1) \rightarrow v_2(t_2)$ and $g_s : v_2(t_2) \rightarrow v_3(t_3)$. Because the views are virtual and operations are actually performed on the concrete data, the compiled code will perform $g_s \circ f_s : t_1 \rightarrow t_3$. However, the result of function f_s, as seen by the compiler, must be $v_2(t_2)$ rather than t_2 because function g is defined for abstract type a_2 and is inherited by $v_2(t_2)$, but is undefined for concrete type t_2.

The roles of a cluster are used within generics to specify types that are related to a known view type. Each view type has a pointer to its cluster, and the cluster's roles likewise point to the view types; therefore, it is possible to find the cluster from a view type and then to find the view type corresponding to a given role of that cluster. The GLISP construct `(clustertype` *role code*`)` returns at compile time the type that fills *role* in the cluster to which the type of *code* belongs; this construct can be used as a type specifier in a generic, allowing use of any view type of the cluster. For example, a generic's result type can be declared to be the view type that fills a specified role of the cluster to which an argument of the generic belongs. `clustertype` can also be used to declare local variable types and to create new data of a concrete type through its view. Thus, type signatures and types used within generics can be mapped from the abstract level to the view level so that specialization of generics is performed correctly.

5 View Cluster Construction: VIEWAS

A view cluster may be complex, and detailed knowledge of the generic procedures is needed to specify one correctly. We expect that abstract types and view clusters will be defined by experts; however, it should be simple for programmers to reuse the generics. VIEWAS makes it easy to create view clusters without detailed understanding of the abstract types and generics. Its inputs are the name of the view cluster and concrete type(s). VIEWAS determines correspondences between the abstract types of the cluster and the concrete types, asking questions as needed; from these it creates the view cluster and view types.

```
(gldefviewspec
  '(sorted-linked-list (sorted-sequence)
     sll t
     ((record anything))
     ((type pointer (^ record))
      (prop link (partof record pointer) result pointer)
      (prop copy-contents-names (names-except record (link)) )
      (prop sort-value (choose-prop-except record (link)))
      (prop sort-direction (oneof (ascending descending)) ) )
     ((pointer pointer)
      (record record
         prop (link copy-contents-names sort-value sort-direction)))
     (sll)) )
```

Figure 11: View Specification used by VIEWAS

Fig. 11 shows the view specification for a sorted linked list. `((record anything))` is a list of formal parameters that correspond to types given in the call to VIEWAS; argument `record` can have a type matching `anything`. Next is a list of names and specifications that are to be matched against the concrete type; following that is a pattern for the output cluster, which is instantiated by substitution of the values determined from the first part. Finally, there is a list of super-clusters of the cluster to be created, `(sll)`.

The previous example, `(viewas 'sorted-linked-list 'myrec)`, specifies the name of the target cluster and the concrete type `myrec` to be matched. VIEWAS first matches the `record` with the `myrec` argument; then it processes the matching specifications in order:

1. `(type pointer  (^ record))`
   The first thing to be determined is the type of the `pointer` to the `record`. The `pointer` defaults to a standard pointer to the record, but a different kind of pointer, such as an array index, will be used if it has been defined.

2. `(prop link (partof record pointer) result pointer)`
   The `link` must be a field of the `record`, of type `pointer`. *Type filtering* restricts the possible matches; if there is only one, it is selected automatically.

3. `(prop copy-contents-names (names-except record (link)) )`
   These are the names of all fields of the record other than the `link`; the names are used by generics that copy the contents of a record.

4. `(prop sort-value  (choose-prop-except record (link)) )`
   The `sort-value` is compared in sorting; it is chosen from either fields or computed (method) values defined for the `record` type, excluding the field that is the `link`. A menu of choices is presented to the user.

5. `(prop sort-direction  (oneof (ascending descending)) )`
   This must be `ascending` or `descending`; the user is asked to choose.

After the items have been matched with the concrete type, the results are substituted into a pattern to form the view type cluster. Fig. 6 above shows cluster `myrec-as-sll` and view types `myrec-as-sll-pointer` and `myrec-as-sll-record` produced by VIEWAS. Properties needed by generics of `sorted-linked-list`, such as `sort-value`, are defined in terms of the concrete type. Generics defined for `sorted-linked-list` explicitly test `sort-direction`; since this value is constant, only the code for the selected direction is kept. This illustrates that switch values in view types can select optional features of generics. Weide [73] notes that options in reusable procedures are essential to avoid a combinatoric set of versions. For example, the Booch component set [9] contains over 500 components; Batory [5] has identified these as combinations of fewer design decisions.

The `linked-list` library has 28 procedures; one view cluster allows specialization of any of them. VIEWAS requires minimal input; it presents sensible choices, so the user does not need to understand the abstract types in detail. In effect, view specifications use a special-purpose language that guides type matching; this language is not necessarily complete, but is sufficient for a variety of view clusters. Some specifications prevent type errors and often allow a choice to be made automatically, as in the case of the `link` field. Others, e.g. `copy-contents-names`, perform bookkeeping to reduce user input. Specifications such as that for `sort-value` heuristically eliminate some choices; additional restrictions might be helpful, e.g., `sort-value` could require a type that has an ordering. VIEWAS is not a powerful type matcher, but it tends to be self-documenting, eliminates opportunities for errors, and is effective with minimal input. We assume that the user understands the concrete type and understands the abstract types well enough to make the choices presented. VIEWAS is intended for views of data structures; a companion program MKV [54] uses a graphical interface and algebraic manipulation of equations to make mathematical views. We have also investigated creation of programs from connections of diagrams that represent physical and mathematical models [51].

6 Higher-order Code

6.1 Compound Structures

Abstract types may be used in larger structures. For example, several kinds of `queue` can be made from a `linked-list`: `front-pointer-queue`, with a pointer to the front of a linked list, `two-pointer-queue`, with pointers to the front and the last record, and `end-pointer-queue`, with a pointer to the last record in a circularly linked list. A sequence of `queue`, in turn, can be used for a `priority-queue`. Generics for compound structures are often small and elegant; for example, insertion in a priority queue is:

```
(gldefun priority-queue-insert
   (q:priority-queue n:integer new)
     (insert (index q n) new) )
```

The code `(index q n)` indexes the sequence by priority `n` to yield a `queue`. `insert` is interpreted relative to the type of `queue`. This small function expands into larger code because its operations expand into operations on component structures, which are further expanded. A single definition of a generic covers the combinatoric set of component types.

6.2 Generic Loop Macros

A language with abstract types must provide loops over collections of data. Alphard [66] and CLU [38] allow iterators for concrete types; Interlisp [30] provided a flexible looping construct for Lisp lists. SETL [14] provides sets and maps and compiles loops over them, with implementations chosen by the compiler [63]. Generic procedures need loops that are independent of data structure (e.g., array, linked list, or tree); this is done by loop macros.

Expansion of generic procedures obeys strict hierarchy and involves independent name spaces. In expanding a loop, however, code specified in the loop statement must be interspersed with the code of the iterator, and the iterator must introduce new variables at the same lexical level; macros are used for these reasons. Names used by the macro are changed when necessary to avoid name conflicts.

GLISP provides a generic looping statement of the form:

`(for` *item* `in` *sequence* [`when` $p(item)$] *verb* $f(item)$ `)`

When this statement is compiled, iterator macros defined for the *verb* and for the type of *sequence* are expanded in-line. For example, consider:

```
(gldefun t3 (r:myrec)
  (for x in (sorted-linked-list r)
    sum (size x)))
```

This loop iterates through a sequence `r` using its `sorted-linked-list` view and inheriting the `linked-list` iterator; it `sums` the `size` of each element `x` of the linked list. Macros are provided for looping, summation, max, min, averaging, and statistics. Collection macros collect data in a specified form, making it possible to convert from one kind of collection to another (e.g., from an array to a linked list).

Data structures may have several levels of structure. 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 each array element is a pointer to a bucket, i.e., a sorted linked list of symbols. A `double-iterator` macro is defined that composes two iterators, allowing a loop over a compound structure:

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

`t4` concatenates the names of all the symbols. The loop expands into nested loops (first through the array of buckets, then through each linked list) and returns a string (since a `name` is a `string` and `+` concatenates `strings`). The compiled code is 23 lines of Lisp.

6.3 Copying and Representation Change

The GLISP compiler can recursively expand code through levels of abstraction until operations on data are reached; it interprets code relative to the types to which it is applied. In Lisp, `funcall` is used to call a function that is determined at runtime. In GLISP, a `funcall` whose function argument is constant at compile time is treated like other function calls, i.e., it is interpreted relative to its argument types. This makes it possible to write higher-order code that implements compositions of views.

The contents of a linked-list record may consist of several items of different types. Generic `copy-list` makes a new record and copies the contents fields into it, requiring several assignments. This is accomplished by a loop over the `copy-contents-names` defined in the view type. For each name in `copy-contents-names`, a `funcall` of that name on the destination record is assigned the value of a `funcall` of that name on the source record.

```
(for name in (copy-contents-names (^. l)) do
  ((funcall name (implementation (^. l)))
     := (funcall name (implementation (^. m))) ) )
```

Since the list of names is constant, the loop is unrolled. Each `funcall` then has a constant name that is interpreted as a field or method reference; the result is a sequence of assignments.

Since the "function call" on each side of the assignment statement is interpreted relative to the type to which it is applied, this higher-order code can transfer data to a different record type and can change the representation during the transfer, e.g., by converting the `radius` of a circle in one representation to the `area` in another representation, or by converting the data to reflect different representations or units of measurement [53]. For example, consider two different types `cira` and `cirb`, each of which has a `color` and lists `circle` as a superclass:

```
(cira (cons (color symbol)
            (cons  (nxt (^ cira))
                   (radius integer)))
  supers (circle))

(cirb (list (diameter roman)
            (color symbol)
            (next (^ cirb)))
  prop   ((radius (diameter / 2)))
  supers (circle))
```

These types have different records, and `cira` contains an integer `radius` while `cirb` contains `diameter` represented in `roman` numerals. After viewing each type as a `linked-list`, it is possible to copy a list from either representation to the other. This illustrates how higher-order code is expanded. First, the loop is unrolled into two assignment statements that transfer `color` and `diameter` from source record to destination record; then `diameter` is inherited from `circle` for the source record and encoded into Roman numerals for the destination record:

```
(gldefun t5 (u:cira &optional v:cirb)
  (copy-list-to (linked-list u) (linked-list v)))

(t5 '(RED (GREEN (BLUE NIL . 12) . 47) . 9))
 = ("XVIII" RED ("XCIV" GREEN ("XXIV" BLUE NIL)))
```

6.4 Several Views

Viewing concrete data as a conceptual entity may involve several views; e.g., a `polygon` can be represented as a sequence of points. Viewing a concrete type as a polygon requires a view of the concrete type t_1 as a sequence of some type t_2 and a view of t_2 as a `vector` (Fig. 12).

View v_2 from the element of the concrete sequence to a *vector* is specified declaratively by giving the name of the view. This view name is used in a `funcall` inside the polygon procedures; it acts as a type change function that changes the type of the sequence element to its view as a point. This effectively implements composition of views. A single generic can be specialized for a variety of polygon representations. For example, a string of characters can be viewed as a polygon by mapping a character to its position on a keyboard: does the string "car" contain the character "d" on the keyboard?

$$
\begin{tikzcd}
t_1 \arrow[r, "v_1"] & seq(t_2) \arrow[r, dashed] \arrow[d, "iterator"'] & seq(vector) \arrow[d, dashed, "iterator"'] \\
 & t_2 \arrow[r, "v_2"] & vector
\end{tikzcd}
$$

Figure 12: Polygon: Sequence of Vector

6.5 Application Languages

The Lisp output of our system can be mechanically transformed into other languages, including C, C++, Java, and Pascal. GLISP allows a target language record as a type; accesses to such records are compiled into Lisp code that can be transformed into target language syntax. The code can also run within Lisp to create and access simulated records; this allows the interactive Lisp environment and our programming and data-display tools to be used for rapid prototyping.

Conversion of Lisp to the target language is done in stages. Patterns are used to transform Lisp idioms into corresponding target idioms and to transform certain Lisp constructs (e.g., returning a value from an `if` statement) into constructs that will be legal in the target language. These Lisp-to-Lisp transformations are applied repeatedly until no further transformations apply. A second set of patterns transforms the code into nicely-formatted target language syntax. The result may be substantially restructured.

A C procedure `sll_insert_1` was shown in Fig. 8. This code is readable and has no dependence on Lisp. Versions of generic procedures containing a few hundred lines of code have been created in C, C++, Java, and Pascal. The C version of the convex hull program, described below, runs 20 times faster than the Lisp version.

6.6 A Larger Example: Convex Hull

The convex hull of a set of points is the smallest convex polygon that encloses them. Kant [32] studied highly qualified human subjects who wrote algorithms for this task. All subjects took considerable time; some failed or produced an inefficient solution. Although convex hull algorithms are described in textbooks [64] and in the literature, getting an algorithm from such sources is difficult: it is necessary to understand the algorithm, and a published description may omit details of the algorithm or even contain errors [22]. A hand-coded version of a published algorithm requires testing or verification.

Fig. 13 illustrates execution of a generic convex hull algorithm. We describe the algorithm

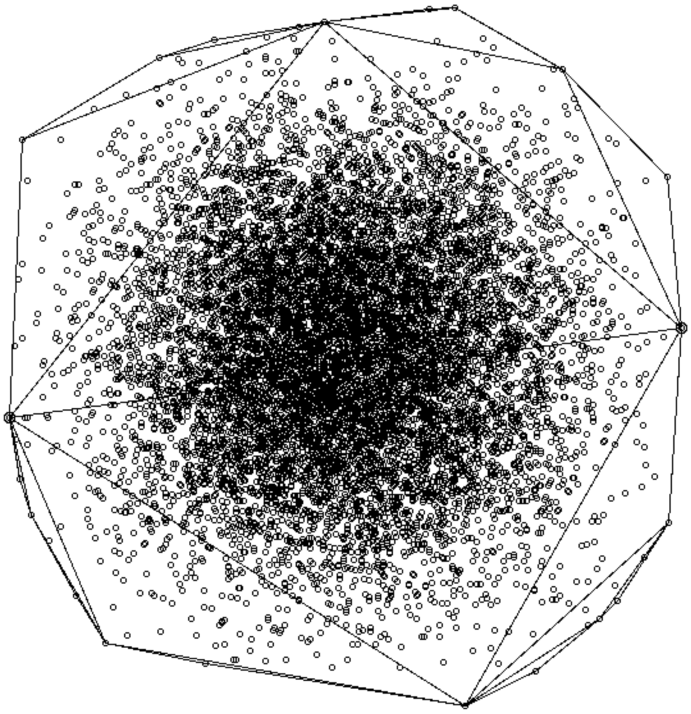

Figure 13: Convex Hull of Points

and illustrate its use on cities viewed as points. The algorithm uses several views of the same data and reuses other generics; it is similar to the QUICKHULL algorithms [57].

A convex hull is represented as a circularly linked list of vertex points in clockwise order (Fig. 14). An edge is formed by a vertex and its successor. Associated with a vertex is a list of points that may be outside the edge; an edge is *split* (Fig. 15) by finding the point that is farthest to the left of the edge. If there is such a point, it must be on the convex hull. The edge is split into two edges, from the original vertex to the new point and from the new point to the end vertex, by splicing the new point into the circularly linked list. The subsets of the points that are to the left of each new edge are collected and stored with the corresponding vertex points, and the edges are split again.

The algorithm is initialized by making the points with minimum and maximum `x` values into a two-point polygon with all input points associated with each edge; then each edge is split. Finally, the vertices are collected as a (non-circular) linked list. Fig. 13 shows the successive splittings. The algorithm rapidly eliminates most points from consideration.

Fig. 16 shows the type cluster used for convex hull. The `line` formed by a point and its successor is declared as a *virtual* `line-segment` [46]; this is another way of specifying a view. This allows the polygon to be treated simultaneously as a sequence of vertices and as a sequence of edges. Only vertices are represented, but the algorithm deals with edges as well. The `internal-record` specifies `circular-linked-list` under property `viewspecs`; this causes VIEWAS to be called automatically to make the `circular-linked-list` view

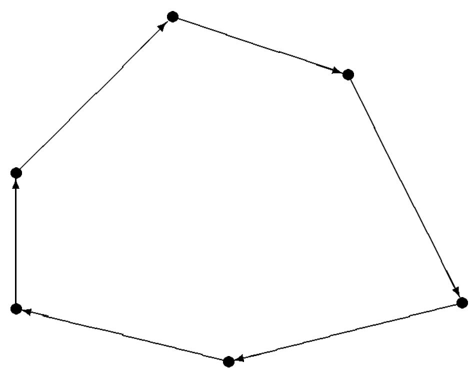

Figure 14: Convex Hull as Circular Linked List of Points

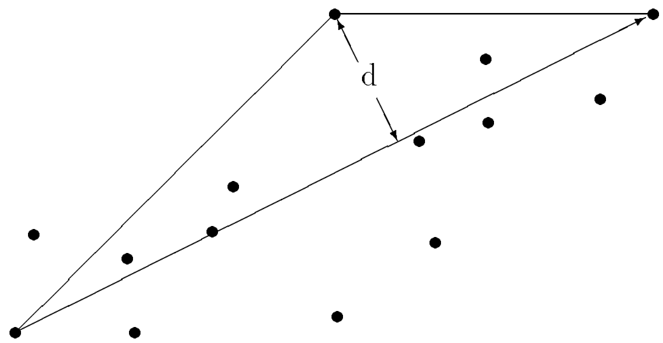

Figure 15: Splitting an Edge

so that procedure `splice-in` and the iterator of that view can be used.

Fig. 17 shows generic procedure `convex-hull`. This procedure initializes the algorithm by finding the two starting points; use of iterators `min` and `max` simplifies the code. Next, an initial circularly linked list is made by linking together the starting points, and function `split` is called for each. Finally, the vertex points are collected as a non-circular list.

Fig. 18 shows generic `cvh-split`, which uses iterator `max` and signed `leftof-distance` from a line-segment to a point; this generic is inherited since the `line` associated with a vertex is a virtual `line-segment`. `leftof-distance` expands as in-line code that operates directly on the linked list of vertex records; we can *think of* a vertex and its successor as a line-segment without materializing one. Operator `+_` specifies a `push` on a list to collect points. `cvh-split` also uses procedure `splice-in` of the `circular-linked-list` view of the points. The split algorithm views the same data in three different ways: as a vertex point, as an edge line-segment, and as a circularly linked list. We believe that use of several views is common and must be supported by reuse technologies.

A programmer should not have to understand the algorithm to reuse it. The concrete

```
(gldefclusterc
  'convex-hull-cluster
  '((source-point      (convex-hull-source-point vector))
    (source-collection (convex-hull-source-collection
                         (listof convex-hull-source-point)
                         prop ((hull convex-hull specialize t))))
    (internal-record   (convex-hull-internal-record
                         (list (pt     convex-hull-source-point)
                               (next   (^ convex-hull-internal-record))
                               (points (listof convex-hull-source-point)))
                         prop ((line ((virtual line-segment with
                                              p1 = (pt self)
                                              p2 = (pt (^. (next self)))) )))
                         msg  ((split cvh-split  specialize t))
                         viewspecs ((circular-linked-list)) )) )
  '())
```

Figure 16: Convex Hull Cluster

data might not be points *per se* and might have other attributes. To find the convex hull using a traditional algorithm would require making a new data set in the required form, finding the convex hull, and then making a version of the hull in terms of the original data. Specialization of generics is more efficient. For example, consider finding the convex hull of a set of cities, each of which has a latitude and longitude. Fig. 19 shows the city data structure and a hand-written view as a point using a Mercator projection. VIEWAS was used to make a convex hull view of the `city-as-point` data. Using this view, a specialized version of the convex hull algorithm (229 lines of Lisp) was produced (in about 5 seconds) that operates on the original data.

This example illustrates the benefits of our approach to reuse. The generic procedures themselves are relatively small and easy to understand because they reuse other generics. Reuse of the generic procedure for an application has a high payoff: the generated code is much larger and more complex than the few lines that are entered to create the views.

6.7 Testing and Verification

Users must have confidence that reused programs will behave as intended. Programmer's Apprentice [61] produced Ada code; the user would read this code and modify it as necessary. We do not believe a programmer should read the output of any reuse system. With our system, in-line code expansion and symbolic optimization can make the output code difficult to read and to relate to the original code sources. Reading someone else's code is difficult, and no less so if the "someone else" is a machine.

```
(gldefun convex-hull (orig-points:(listof vector))
  (let (xmin-pt xmax-pt hullp1 hullp2)
    (if ((length-up-to orig-points 2) >= 2)
        then (xmin-pt := (for p in orig-points min (x p)))
             (xmax-pt := (for p in orig-points max (x p)))
             (hullp1 := (a (clustertype internal-record orig-points)
                            with pt     = xmin-pt
                                 points = orig-points))
             (hullp2 := (a (clustertype internal-record orig-points)
                            with pt     = xmax-pt
                                 points = orig-points))
             ((next hullp1) := hullp2)       ; link circularly
             ((next hullp2) := hullp1)
             (split hullp1)
             (split hullp2)
             (for p in (circular-linked-list hullp1)
                  collect (pt p)) ) ))
```

Figure 17: Generic Convex Hull Procedure

We believe that a reuse system such as ours will reduce errors. Errors in reusing software components might arise from several sources:

1. The component itself might be in error.
2. The component might be used improperly.
3. The specialization of a component might not be correct.

Algorithms that are reused justify careful development and are tested in many applications, so unnoticed errors are unlikely. Humans introduce errors in coding algorithms; Sedgewick [64] notes "Quicksort ... is fragile: a simple mistake in the implementation can go unnoticed and can cause it to perform badly for some files." Reuse of carefully developed generics is likely to produce better programs than hand coding.

VIEWAS and MKV guide the user by presenting only correct choices. When views are written by hand, type checking usually catches errors. Although GLISP is not strongly typed (because of its Lisp ancestry), there are many error checks that catch nearly all type errors. Our experience with our system has been good, and we have reused generics for new applications; e.g., the generic for distance from a line to a point was reused to test whether a mouse position is close to a line.

Ultimately, it must be verified not only that software meets its specification but also that it is what the user really wants. With rapid prototyping based on reuse, developers

```
(gldefun cvh-split (cp:cvhpoint)
  (let (maxpt pts newcp)
    (pts := (points cp))
    ((points cp) := nil)
    (if pts is not null
      then (maxpt := (for p in pts when ((p <> (p1 (line cp)))
                                       and (p <> (p2 (line cp))))
                          max (leftof-distance (line cp) p)))
           (if maxpt and (leftof (line cp) maxpt)
             then (newcp := (a (typeof cp) with pt = maxpt))
                  (splice-in (circular-linked-list cp)
                             (circular-linked-list newcp))
                  (for p in pts do
                    (if (leftof (line cp) p)
                      then ((points cp) +_ p)
                      else (if (leftof (line newcp) p)
                             then ((points newcp) +_ p))) )
                  (split cp)
                  (split newcp) ) ) ))
```

Figure 18: Generic Split Procedure

can address a program's performance in practice and make modifications easily. Our system allows significant representation changes to be accomplished easily by recompilation.

Formal verification might be applied to specialized generics. Gries and Prins [21] suggest a stratified proof of a program obtained by transformation: if a generic is correct and an implementation of its abstract type is correct, the transformed algorithm will be correct. Morgan [42] extends these techniques for proofs of data refinements. Morris [43] provides calculational laws for refinement of programs written in terms of abstract types such as bags and sets. Related methods might be used for proofs of refinements with a system such as ours; a library of proven lemmas about generic components would greatly simplify the task of proving a software system correct.

7 Views and OOP

Views can be used to generate methods that allow concrete data to be used with OOP software; this is useful for reuse of OOP software that uses runtime messages to interface to diverse kinds of objects. The GLISP compiler can automatically compile and cache specialized versions of methods based on the definitions given in a type; for example, a method to compute the `area` of a `pizza-as-circle` can be generated automatically.

```
(city
  (list (name symbol)
        (latitude (units real degrees))
        (population integer)
        (longitude (units real degrees)))
  views ((point city-as-point)) )

(city-as-point (z17 city)
  prop
    ((x ((let (rad:(units real radians))
           (rad := (longitude z17))
           rad)))
     (y ((signum (latitude z17)) *
          (log (tan (pi / 4) +
                    (abs (latitude z17))
                       / 2)))))
  supers (vector))
```

Figure 19: City and Mercator Projection

We have implemented direct-manipulation graphical editors for `linked-list` and `array`. A display method and editor for a record can be made interactively using program DISPM, which allows selection of properties to be displayed, display methods to be used, and positions. Given a display method for a record, a generic for displaying and editing structured data containing such records can be used on the concrete data. Figure 20 shows data displayed by the generic `linked-list` editor. The user can move forward or backward in the displayed list or excise a selected list element; the user can also zoom in on an element to display it in more detail or to edit the contents. This technique allows a single generic editor to handle all kinds of `linked-list`. The display omits detail such as contents of link fields and shows the data in an easily understood form.

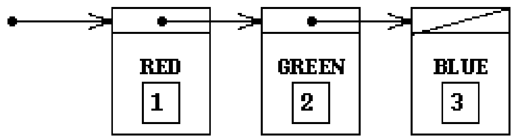

Figure 20: Linked List Display

8 Related Work

We review closely related work. It is always possible to say "an equivalent program could be written in language x"; however, a system for software reuse must satisfy several criteria *simultaneously* to be effective [34]. We claim that the system described here satisfies all of these criteria:

1. It has wide applicability: many kinds of software can be expressed as reusable generics.
2. It is easy to use. The amount of user input and the learning required are small.
3. It produces efficient code in several languages.
4. It minimally constrains the representation of data. Generics can be specialized for use with existing data and programs.

Brooks [10] contends that there are "no silver bullets" in software development. The system described here is not a silver bullet, but it suggests that significant improvement in software development is possible.

8.1 Software Reuse

Krueger [34] is an excellent survey of software reuse, with criteria for practical effectiveness. Biggerstaff and Perlis [8] contains papers on theory and applications of reuse; artificial intelligence approaches are described in [1], [39], and [60]. Mili [41] extensively surveys software reuse, emphasizing technical challenges.

8.2 Software Components

The Programmer's Apprentice [61] was based on reuse of *clichés*, somewhat analogous to our generics. This project produced some good ideas but had limited success. KBEmacs, a knowledge-based editor integrated with Emacs, helped the user transform cliché code; unfortunately, KBEmacs was rather slow, and the user had to read and understand the low-level output code. We assume that the user will treat the outputs of our system as "black boxes" and will not need to read or modify the code. Rich [59] describes a *plan calculus* for representing program and data abstractions; *overlays* relate program and data plans, analogous to our views.

Weide [73] proposed a software components industry based on formally specified and unchangeable components. Because the components would be verified and unchangeable, errors would be prevented; however, the rigidity of the components might make them harder to reuse. Our approach adapts components to fit the application.

Zaremski and Wing [77] describe retrieval of reusable ML components based on signature matching of functions and modules; related techniques could be used with our generics.

Batory [4] [5] [6] describes a data structure precompiler and construction of software systems from layers of plug-compatible components with standardized interfaces. The use of layers whose interfaces are carefully specified allows the developer to ensure that the layers will interface correctly. We have focused on adapting interfaces so that generics can be reused for independently designed data.

8.3 Languages with Generic Procedures

Ada, Modula-2 [28], and C++ [69] allow modules for parameterized abstract types such as $STACK[type]$. Books of generic procedures [37] [44] contain some of the same procedures that are provided with our system. In Ada and Modula-2, such collections have limited value because such code is easy to write and is only a small part of most applications. The class, template, and virtual function features of C++ allow reuse of generics; however, Stroustrup's examples [69] show that the declarations required are complex and subtle. Our declarations are also complex, but VIEWAS hides this complexity and guides the user in creating correct views. The ideas in VIEWAS might be adapted for other languages.

8.4 Functional and Set Languages

ML [74] [55] is like a strongly typed Lisp; it includes polymorphic functions (e.g., functions that operate on lists of an arbitrary type) and functors (functions that map structures 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. Our system allows storing into a data structure through a view and composition of views [52].

Miranda [71] is a strongly-typed functional language with higher-order functions. While this allows generics, it is often hard to write functional programs with good performance.

SETL [14] provides sets and set operations. [63] describes an attempt to automatically choose data structures in SETL to improve efficiency. Kruchten *et al.* [35] say "slow is beautiful" to emphasize ease of constructing programs, but inefficient implementations can make even small problems intractable.

8.5 Transformation Systems

Transformation systems repeatedly replace parts of an abstract algorithm specification with code that is closer to an implementation, until executable code is reached. Our views specify transformations from features of abstract types to their implementations.

Kant *et al.* [33] describe Sinapse, which generates programs to simulate spatial differential equations, e.g. for seismic analysis. Sinapse transforms a small specification into a much larger program in Fortran or C; it is written using Mathematica [75] and appears to work well within its domain.

Setliff's Elf system [65] automatically generates data structures and algorithms for wire routing on integrated circuits and printed circuit boards. Rules are used to select refinement transformations based on characteristics of the routing task.

KIDS [68] transforms problem statements in first-order logic into programs that are highly efficient for certain combinatorial problems. The user must select transformations to be used and must supply a formal domain theory for the application. This system is interesting and powerful, but its user must be mathematically sophisticated.

Gries and Prins [21] proposed use of syntactic transformations to specify implementation of abstract algorithms. Volpano [72] and Gries [23] describe systems in which a user specifies transformations for variables, expression patterns, and statement patterns; by performing substitutions on an algorithm, a different version of the algorithm is obtained. This method allows the user to specify detailed transformations for a particular specialization of an algorithm, whereas we rely on type-based transformations and on general optimization patterns. The ability to specify individual transformations in Gries' system gives more flexibility, possibly at the cost of writing more patterns.

The Intentional Programming project at Microsoft [67] is based on *intentions*, which are similar to algorithm fragments expressed as abstract syntax trees. Intentions can be transformed by *enzymes* at the abstract syntax tree level and can be parsed and unparsed into various surface syntaxes by methods stored with or inherited by the intentions. This work is in progress; its results to date are impressive.

Berlin and Weise [7] used *partial evaluation* to improve efficiency of scientific programs. Given that certain features of a problem are constant, their compiler performs as many constant calculations as possible at compile time, yielding a specialized program that runs faster. Our system includes partial evaluation by in-lining and symbolic optimization. Consel and Danvy [12] survey work on partial evaluation.

8.6 Views

Goguen [18] proposes a library interconnection language, LIL. This proposal has much in common with our approach, and Goguen uses the term *view* similarly; LIL has a stronger focus on mathematical descriptions and axioms. The OBJ language family of Goguen *et al.* [19] has views that are formal type mappings; operators are mapped by strict isomorphisms. Tracz [70] describes LILEANNA, which implements LIL for construction of Ada packages; views in LILEANNA map types, operations, and exceptions between theories. In our system, views are computational transformations between types; general procedures as well as operators can be reused.

Garlan [17] and Kaiser [31] use views to allow tools in a program development environment to access a common set of data. Their MELD system can combine *features* (collections of classes and methods) to allow "additive" construction of a system from selected features.

Meyers [40] discusses problems of consistency among program development tools and surveys approaches including use of files, databases, and views as developed by Garlan.

Hailpern and Ossher [25] describe views as subsets of methods of a class, to restrict certain methods to particular clients. Harrison and Ossher [26] argue that OOP is too restrictive for applications that need their own views of objects; they propose *subjects* that are analogous to class hierarchies.

8.7 Data Translation

IDL (Interface Description Language) [36] translates representations, possibly with structure sharing, for exchange of data between parts of a compiler, based on precise data specifications. Herlihy and Liskov [27] describe transmission of data over a network, with representation translation and shared structure; the user writes procedures to encode and decode data for transmission. The Common Object Request Broker Architecture (CORBA) [13] includes an Interface Definition Language and can automatically generate stubs to allow interoperability of objects across distributed systems and across languages and machine architectures. The ARPA Knowledge-Sharing Project [45] addresses the problem of sharing knowledge bases that were developed using different ontologies. Purtilo and Atlee [58] describe a system that translates calling sequences by producing small interface modules that reorder and translate parameters as necessary for the called procedure. Re-representation of data allows reuse of an existing procedure; it requires space and execution time, although [36] found this was a small cost in a compiler. [49] and this paper describe methods for data translation, but these do not handle shared structure.

Guttag and Horning [24] describe a formal language for specifying procedure interface signatures and properties. Yellin and Strom [76] describe semi-automatic synthesis of protocol converters to allow interfacing of clients and servers.

8.8 Object-oriented Programming

We have described how views can be used to generate methods for OOP. In OOP, messages are interpreted (converted to procedure calls) depending on the type of the receiving object; methods can be inherited from superclasses. The close connection between a class and its superclasses requires the user to understand a great deal about a class and its methods. In many OOP systems, a class must include all data of its superclasses, so reuse with OOP restricts implementation of data; names of data and messages must be consistent and must not conflict. Holland [29] uses *contracts* to specify data and behavioral obligations of composed objects. Contracts are somewhat like our clusters, but require that specializations include certain instance data and implement data in the same way as the generics that are to be specialized. A separate "contract lens" construct is used to disambiguate names where there are conflicts. Our views provide encapsulation that prevents name conflicts; views allow the reuse benefits of OOP with flexibility in implementing data.

Some OOP systems are inefficient: since most methods are small, message interpretation overhead can be large, especially in layered systems. C++ [69] has restricted OOP with efficient method dispatching. *Opacity* of objects prevents optimization across message boundaries unless messages are compiled in-line; C++ allows in-line compilation. Reuse in OOP may require creating a new object to reuse a method of its class; views allow an object to be *thought of* as another type without having to materialize that type.

9 Conclusions

Our approach is based on reuse of programming knowledge: generic procedures, abstract types, and view descriptions. We envision a library of abstract types and generics, developed by experts, that could be adapted quickly for applications. Programmers of ordinary skill should be able to reuse the generics. VIEWAS facilitates making views; easily used interfaces, as opposed to verbose textual specifications with precise syntax, are essential for successful reuse. Systems like VIEWAS might reduce the complexity of the specifications required in other languages. Views also support data translation and runtime message interpretation: a single direct-manipulation editor can handle all implementations of an abstract type.

These techniques provide high payoff in generated code relative to the size and complexity of input specifications. They require only modest understanding of the details of library procedures for successful reuse.

Our techniques allow restructuring of data to meet new requirements or to improve efficiency. Traditional languages reflect the data implementation in the code [3], making changes costly. Our system derives code from the data definitions; design decisions are stated in a single place and distributed by compilation rather than by hand coding.

The ability to produce code in different languages decouples the choice of programming tools from the choice of application language. It allows new tools to extend old systems or to write parts of a system without committing to use of the tool for everything. Just as computation has become a commodity, so that the user no longer cares what kind of CPU chip is inside the box, we may look forward to a time when today's high-level languages become implementation details.

10 Acknowledgements

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

We thank David Gries, Hamilton Richards, Ben Kuipers, and anonymous reviewers for their suggestions for improving this paper.

References

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

[2] M. Arbib and E. Manes, *Arrows, Structures, and Functors: The Categorical Imperative*, New York: Academic Press, 1975.

[3] R. Balzer, "A 15 Year Perspective on Automatic Programming," *IEEE Trans. Software Engineering*, vol. 11, no. 11 pp. 1257-1267, Nov. 1985.

[4] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components," *ACM Trans. Software Engineering and Methodology*, vol. 1, no. 4, pp. 355-398, Oct. 1992.

[5] D. Batory, V. Singhal, J. Thomas, and M. Sirkin, "Scalable Software Libraries," *Proc. ACM SIGSOFT '93: Foundations of Software Engineering*, Dec. 1993.

[6] D. Batory, J. Thomas, and M. Sirkin, "Reengineering a Complex Application Using a Scalable Data Structure Compiler," in *Proc. ACM SIGSOFT '94*, Dec. 1994.

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

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

[9] G. Booch, *Software Components with Ada*, Benjamin-Cummings, 1987.

[10] F. P. Brooks, "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, vol. 20, no. 4 pp. 10-19. April 1987.

[11] J. C. Cleaveland, *An Introduction to Data Types*, Addison-Wesley, 1986.

[12] C. Consel and O. Danvy, "Tutorial Notes on Partial Evaluation," *ACM Symp. on Principles of Programming Languages*, 1993, pp. 493-501.

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

[14] R. B. K. Dewar, *The SETL Programming Language*, manuscript, 1980.

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

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

[17] D. Garlan, "Views for Tools in Integrated Environments," in R. Conradi, T. Didriksen, and D. Wanvik (eds.), LNCS 244: *Adv. Prog. Environments*, Springer, 1986.

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

[19] J. A. Goguen, "Principles of Parameterized Programming," [8], pp. 159-225.

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

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

[22] D. Gries and I. Stojmenović, "A Note on Graham's Convex Hull Algorithm," *Information Processing Letters*, vol. 25, no. 5 (July 1987), pp. 323-327.

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

[24] J. V. Guttag and J. J. Horning, "Introduction to LCL, a LARCH/C Interface Language," TR SRC-74, D.E.C. Software Res. Ctr., Palo Alto, CA, 1991.

[25] 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.

[26] W. Harrison and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)," *Proc. OOPSLA-93, ACM SIGPLAN Notices*, vol. 28, no. 10, pp. 411-428, Oct. 1993.

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

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

[29] I. M. Holland, "Specifying Reusable Components Using Contracts," in O. Lehrmann Madsen (ed.), *Proc. 6th European Conf. Object-Oriented Prog. (ECOOP-92),* LNCS 615, pp. 287-308, Springer, July 1992.

[30] *Interlisp Reference Manual*, Xerox Palo Alto Research Center, 1983.

[31] G. Kaiser and D. Garlan, "Synthesizing Programming Environments from Reusable Features," in [8], pp. 35-55.

[32] E. Kant, "Understanding and Automating Algorithm Design," *IEEE Trans. Software Engineering*, vol. 11, no. 11, pp. 1361-1374, Nov. 1985.

[33] E. Kant, F. Daube, W. MacGregor, and J. Wald, "Scientific Programming by Automated Synthesis," in [39], pp. 169-205.

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

[35] P. Kruchten, E. Schonberg, and J. Schwartz, "Software Prototyping using the SETL Language," *IEEE Software*, vol. 1, no. 4 pp. 66-75, Oct. 1984.

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

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

[38] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder, *CLU Reference Manual*, Springer, 1981.

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

[40] S. Meyers, "Difficulties in Integrating Multiview Development Systems," *IEEE Software*, vol. 8, no. 1, pp. 49-57, Jan. 1991.

[41] 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.

[42] C. Morgan, "Data Refinement by Miracles," *Information Processing Letters*, vol. 26, pp. 243-246, 1988.

[43] J. M. Morris, "Laws of Data Refinement," *Acta Informatica*, vol. 26, pp. 287-308, 1989.

[44] D. Musser and A. Stepanov, *The Ada Generic Library*, Springer, 1989.

[45] 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.

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

[47] G. Novak, "GLISP User's Manual," Tech. Report STAN-CS-82-895, Stanford Univ., 1982; TR-83-25, Univ. of Texas at Austin.

[48] G. Novak, "Data Abstraction in GLISP," *Proc. SIGPLAN '83 Symposium, ACM SIGPLAN Notices*, vol. 18, no. 3, pp. 170-177, June 1983.

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

[50] G. Novak, "Software Reuse through View Type Clusters," *7th Knowledge-Based Software Engineering Conf.*, IEEE CS Press, 1992, pp. 70-79.

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

[52] G. Novak, "Composing Reusable Software Components through Views," *9th Knowledge-Based Software Engineering Conf.*, IEEE CS Press, 1994, pp. 39-47.

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

[54] G. Novak, "Creation of Views for Reuse of Software with Different Data Representations," *IEEE Trans. Software Engineering*, vol. 21, no. 12, pp. 993-1005, Dec. 1995.

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

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

[57] F. Preparata and M. Shamos, *Computational Geometry*, Springer, 1985.

[58] 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.

[59] C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice," *7th Intl. Joint Conf. Artificial Intelligence (IJCAI-81)*, pp. 1044-1052, 1981.

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

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

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

[63] E. Schonberg, J. T. Schwartz, and M. Sharir, "An Automatic Technique for Selection of Data Representations in SETL Programs," *ACM Trans. on Prog. Lang. and Systems*, vol. 3, no. 2 pp. 126-143, Apr. 1981.

[64] R. Sedgewick, *Algorithms*, Addison-Wesley, 1988.

[65] D. Setliff, "On the Automatic Selection of Data Structures and Algorithms," in [39], pp. 207-226.

[66] M. Shaw, W. Wulf, and R. London, "Abstraction and Verification in Alphard: Defining and Specifying Iterators and Generators," *Comm. ACM*, vol. 20, no. 9, pp. 553-564, Aug. 1977.

[67] C. Simonyi, "Intentional Programming - Innovation in the Legacy Age," Presented at IFIP WG 2.1 meeting, June 4, 1996. See `http://www.research.microsoft.com/research/ip/main.htm`

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

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

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

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

[72] D. Volpano. and R. Kieburtz, "The Templates Approach to Software Reuse," in [8], pp. 247-255.

[73] B. Weide, W. Ogden, and S. Zweben, "Reusable Software Components," in M. Yovits, ed., *Adv. in Computers*, vol. 33, Academic Press, 1991, pp. 1-65.

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

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

[76] D. M. Yellin and R. E. Strom, "Interfaces, Protocols, and the Semi-Automatic Construction of Software Adaptors," *Proc. OOPSLA-94, ACM SIGPLAN Notices*, vol. 29, no. 10, pp. 176-190, Oct. 1994.

[77] A. M. Zaremski and J. M. Wing, "Signature Matching: A Key to Reuse," *ACM Software Engineering Notes*, vol. 18, no. 5, pp. 182-190, Dec. 1993.