

Software Reuse By Specialization of Generic Procedures through Views ^[1]

Jamil A.S. Itmazi, PhD. Student, Granada University
Advanced technologies of Software Development
Granada, Spain, E-Mail : Jamil_de_palestina@yahoo.com

Supervised by

Dr. Juan Carlos Granja Álvarez

This research With his presentation is the final work of
(Especificación y Técnicas de Garantía del Software) Course
Granada, 21- Mayo 2003

Note : This paper mainly based on "Software Reuse by Specialization of Generic Procedures through Views", by Gordon S. Novak J, *IEEE Trans. on Software Engineering*, vol. 23, no. 7 (Jul. 1997), pp. 401-417. <http://www.cs.utexas.edu/users/novak/tose97.html>

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.

Table of contents

- Abstract.
- Index Terms.
- Table of contents
- 1. Introduction.
- 2. Views.
- 3. GLISP Language and Compiler.
- 4. Clusters of Views.
- 5. Related Work.
- 6. Conclusions.
- References.

1. Introduction

Algorithm : is a method for solving a class of problems on a computer. The complexity of an algorithm is the cost, measured in running time, or storage, or whatever units are relevant, of using the algorithm to solve one of those problems. Some problems take a very long time, others can be done quickly. Some problems *seem* to take a long time, and then someone discovers a faster way to do them (a 'faster algorithm'). [2]

Ok, I think that **Algorithms** are logical ways to do things. And how does this influence my programming? Well, you use algorithms to tell your application how to do the things you want it to do!

A generic algorithm is one that has (1) a formal specification, (2) a proof that it satisfies this specification, and (3) generic identifiers representing types and operations. [3]

So A generic algorithm is written by abstracting algorithms on specific types and specific data structures so that they apply to arguments whose types are as general as possible. This means that a generic algorithm actually has two parts: the actual instructions that describe the steps of the algorithm and the set of requirements that specify precisely which properties its argument types must satisfy. [4]

There are many cases in which the same algorithm may need to be implemented for several different data types

And we can say that, Generic algorithms

- Operate on containers
- Perform common tasks: searching, sorting, counting, replacing...

But Why “generic algorithms”?

- Algorithm because it encapsulates the implementation of a common task
- Without generic algorithms you wind up writing the same loops over and over again in different contexts
- Generic because it works for any container type.....and for any value type!

Generic Procedures : Generic procedures provide a mechanism by which a Scheme procedure can select different bodies for execution depending on the types of the parameters it was called with. Languages like ANSI Lisp and Dylan provide similar functionality. Generic procedures have several advantages over normal functions:

- must come up with unique names for procedures that perform the same operation on different types of objects. This avoids cluttering the name space. All these procedures can be defined separately but yet be part of the same, single generic procedure.
- The functionality of a generic procedure can be extended incrementally through code located in different places. This avoids "spaghetti code" where adding a new type of objects requires changes to existing pieces of code in several locations.
- Code using generic procedures has a high degree of polymorphism without having to resort to ugly and hard-to-maintain test-type-and-dispatch branching.

A procedure name is generic if it can be referenced with more than one actual argument type/kind/rank pattern. Most intrinsic procedures are generic, and in addition user-defined procedures may be made generic. The interface block is used to specify generic names for user-defined procedures. As with generic intrinsic procedures, it is the type, kind, and rank pattern of the actual argument list in a reference to a generic procedure that determines which underlying specific user-defined procedure is called. [5]

Generic Programming ^[6] : Generic programming is the sub-discipline of software engineering that deals with the conceptual categorization of computational domains, the reduction of algorithms to their minimal conceptual requirements, and strict performance guarantees for families of generic algorithms. The major application area of generic programming is the design, implementation, and documentation of software libraries.

Generic programming is also known as :

- algorithm-oriented programming,
- requirements-oriented programming,
- component-based programming,
- programming with concepts.

Generic programming, means identifying a new kind of abstraction. The central abstraction of generic programming is less tangible than earlier abstractions like the subroutine or the class or the module. It is a *set of requirements* on data types. This is a difficult abstraction to grasp because it isn't tied to a specific C++ language feature. There is no keyword for declaring a set of abstract requirements.

What generic programming provides in return for understanding an abstraction that at first seems frustratingly nebulous is an unprecedented level of flexibility. Just as important, it achieves abstraction without loss of efficiency. Generic programming, unlike object-oriented programming, does not require you to call functions through extra levels of indirection; it allows you to write a fully general and reusable algorithm that is just as efficient as an algorithm handcrafted for a specific data type. ^[4]

Reuse of software can : Reduce cost.
 Increase the speed of software production
 Increase reliability.

In most languages, the types of arguments of a procedure call must match the types of parameters of the procedure. so, reuse is often found where type compatibility occurs naturally,

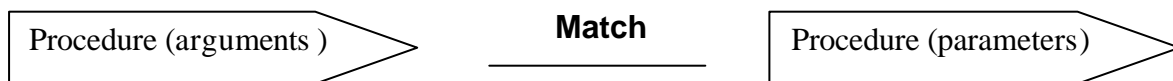


Fig. 1

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.

A *view* provides a mapping between a concrete type and an abstract type¹ in terms of which a generic algorithm is written. In Fig. 2 a view acts as an interface adapter that makes the concrete type appear as the abstract type. The view provides a clean separation between the semantics of data (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. 3

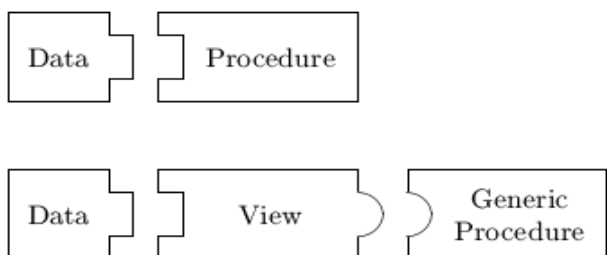


Fig. 2: Interfacing with Strong Typing and with Views

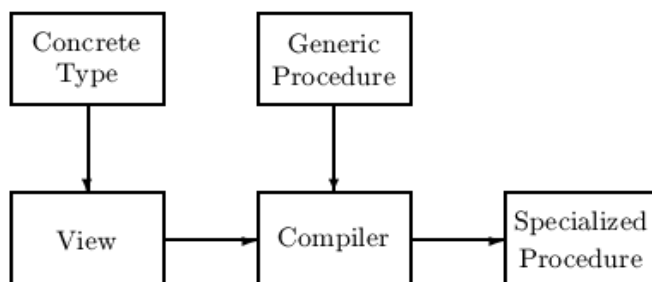


Fig. 3: Specialization of Generic Procedure through View

¹ (We consider an abstract type to be a set of *basis variables* & a set of generic procedures written in terms of the basis variables.)

This approach to reuse has several advantages:

1. It provides freedom to select the implementation of data.
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:
 1. specialize generic procedures from a library,
 2. instantiate a program framework from components,
 3. translate data from one representation to another,
 4. generate methods for object-oriented programming, and
 5. interface to programming tools for data display and editing.

2. Views

Computation as Isomorphism : think of all computation as simulation (or isomorphism). Our views allow broader mappings between concrete and abstract types and include algorithms as well as operators.

Views as Isomorphisms : Reuse of generic algorithms through views corresponds to computation as isomorphism. 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, & mapping back, the result of performing the algorithm on the concrete type is obtained. instead of performing the view mapping explicitly 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. 4).

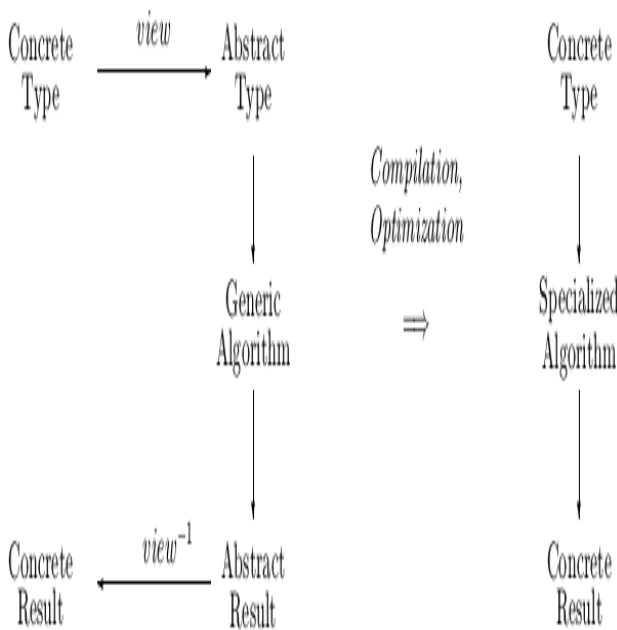


Fig. 4: Specializing a Generic

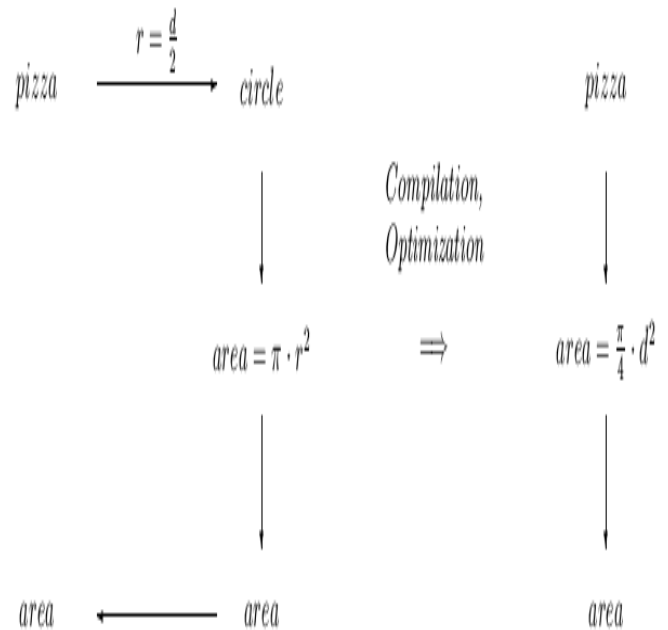


Fig. 5: Example Specialization

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. 5). Code to reference *d* in data structure *pizza* is included in the specialized code.

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* and has a set of named *generic procedures* that are written in terms of the basis variables.

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.

A view encapsulates a concrete type and presents an external interface that consists of the basis variables of the abstract type. Fig. 6 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.

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.

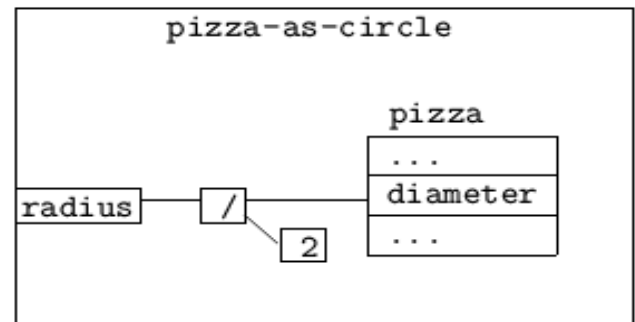


Fig. 6: Encapsulation of Concrete Type by View

A view implements an abstract type if it emulates a record containing the basis variables. Emulation is expressed by 2 properties: [7]

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

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.

3. GLISP Language and Compiler ^[8]

GLISP is a high-level language that is compiled into LISP. It provides a versatile abstract-data-type facility with hierarchical inheritance of properties and OO programming. GLISP programs are shorter and more readable than equivalent LISP programs. The object code produced by GLISP is optimized, making it about as efficient as handwritten LISP. An integrated programming environment is provided, including automatic incremental compilation, interpretive programming features, and an intelligent display-based inspector/editor for data and data-type descriptions. GLISP code is relatively portable; the compiler and the data inspector are implemented for most, major dialects of LISP and are available free or at nominal cost.

GLISP including LISP as a sublanguage, that is compiled into LISP. The GLISP system runs within an existing LISP system and provides an integrated programming environment that includes automatic incremental compilation of GLISP programs, interactive execution and debugging, and display-based editing and inspection of data. Use of GLISP makes writing, debugging, and modifying programs significantly easier; at the same time, the code produced by the compiler is optimized so that its execution efficiency is comparable to that of handwritten LISP. This article describes features of GLISP and illustrates them with examples. Most of the syntax of GLISP is similar to LISP syntax or PASCAL syntax, so explicit treatment of GLISP syntax will be brief.

GLISP programs are compiled relative to a knowledge base or object descriptions, a form of abstract data types. A primary goal of the use of abstract data types in GLISP is to *make programming easier*. The implementations of objects are described in a single place; the compiler uses the object

descriptions to convert GLISP code written in terms of user objects into efficient LISP code written in terms of the implementations of the objects in LISP. This allows the implementations of objects to be changed without changing the code; it also allows the same code to be effective for objects that are implemented in different ways and thereby allows the accumulation of programming knowledge in the form of generic programs. GLISP contains ordinary LISP as a sublanguage; LISP code can

be mixed with GLISP code, so that no capabilities of the underlying LISP system are lost. GLISP provides PASCAL-like reference to substructures and properties, infix arithmetic expressions, and PASCAL-like control statements. Object-centered programming is built in; optimized compilation allows object-centered programs to run efficiently.

GLISP is easily extensible for new object representations. Operator overloading for user-defined objects occurs automatically when arithmetic operators are defined as message selectors for those objects. The compiler can compile optimized code for access to objects represented in user-specified representation languages. GLISP has also been extended as a hardware description language for describing VLSI designs.

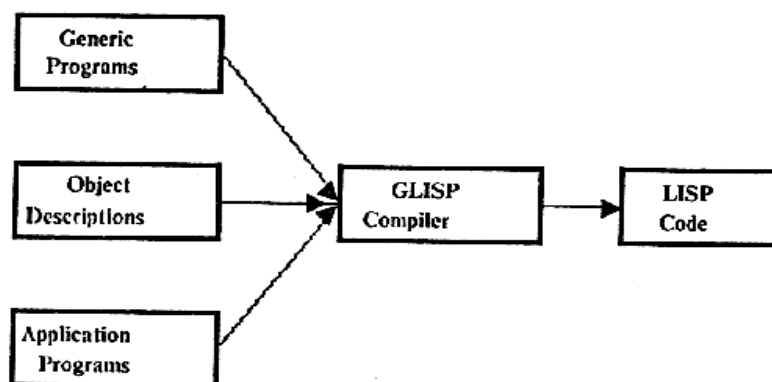


Figure 1. GLISP compilation.

Views in GLISP

A view 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. 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.

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 long as the recursion terminates at compile time. A view type may redefine some methods that are generics of the abstract type; this may improve efficiency. 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 defined by an expression can be inverted algebraically, the compiler does so.
3. A procedure can be defined in the view type to accomplish assignment to a basis variable while maintaining the storage and independence properties.

A view can define a procedure to create an instance of the concrete type from a set of basis variables of the abstract type. This is needed for generics that create new data.

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 2 types by providing a pointer type that is derived from the record type. In general, 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 & 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.

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.

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.

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.

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.

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.

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.

5. Related Work ^[9]

Languages with Generic Procedures

Programming languages such as Ada, Modula-2 & C++ 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 & its procedures. These languages allow much less parameterization than is possible using views.

Functional and Set Languages

ML is like a strongly typed Lisp; it includes polymorphic functions (*e.g.*, functions over lists of an arbitrary type) and factors (functions that map structures, composed of types and functions, to structures). ML also includes references (pointers) that allow imperative programming. ML factors can instantiate generic modules such as container types. 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 is a strongly typed, purely functional lang. 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 : a change to data values requires creation of a new structure in a functional language.

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.

Object-oriented Programming

OOP is popular as a mechanism for software reuse; Gamma *et al.* 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. C++ has relatively efficient message dispatching, at some cost in flexibility. Because GLISP 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. Views allow a partial use of an abstract type.
4. In OOP, the user must learn the available classes and messages. Some OOP operating systems involve over 1,500 classes, but. With views, the user only has to indicate correspondences between the application type and the abstract type.
5. Our system allows translation to a separate application language, such as C, without requiring that the application be written in a particular language.

6. Conclusions

Our approach is based on reuse of programming knowledge: generic procedures, abstract types & 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 & runtime message interpretation.

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, 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.

References

- [1] This paper mainly based on "Software Reuse by Specialization of Generic Procedures through Views", by Gordon S. Novak J, *IEEE Trans. on Software Engineering*, vol. 23, no. 7 (Jul. 1997), pp. 401-417. <http://www.cs.utexas.edu/users/novak/tose97.html>
- [2] *Algorithms & Complexity a full textbook* by H.S. Wilf, www.cis.upenn.edu/~wilf/AlgComp2.html
- [3] V.R. Yakhnis, J.A. Farrell, & S.S. Shultz, www.research.ibm.com/journal/sj/331/yakhins.html
- [4] *Generic Programming & the STL: Using & Extending C++ Standard Template Library*, by M.H. Austern, www.aw.com/catalog/academic/product/1,4096,0201309564,00.html?type=PRE
- [5] <http://www.navo.hpc.mil/pet/Video/Courses/f90/Extra/genproc.html>
- [6] <http://www.hep.ucl.ac.uk/~clarke/OOCourse/module20/24>
- [7] Novak, Gordon, "Composing Reusable Software Components through Views", This article appears in *Proc. 9th Knowledge-Based Software Engineering Conference (KBSE-94)*, pp. 39-47, Monterey CA, Sept. 1994, IEEE Computer Society Press.
- [8] Novak, Gordon, "A Lisp-based Programming System with Data Abstraction", *The AI Magazine*, vol.4, no.3 (Fall 1983), pp.37-47. *Readings from A.I. Magazine*, AAAI Press, 1988, pp.545-555.
- [9] Novak, Gordon, "Creation of Views for Reuse of Software with Different Data Representations", *IEEE Trans. Software Engineering*, vol. 21, no. 12 (Dec. 1995), pp. 993-1005. <http://www.cs.utexas.edu/users/novak/tose95.pdf>