A PROGRAM GENERATION SERVER
ON THE WEB

GORDON S. NOVAK JR.

AUGUST 1997

AI97-262
A Program Generation Server on the Web

Gordon S. Novak Jr.
Dept. of Computer Sciences
University of Texas at Austin
Austin, Texas 78712 USA
(512) 471-9569
novak@cs.utexas.edu

ABSTRACT
An automatic programming server on the Web writes programs and serves them to a user. The user describes the application data, specifies views that map data to abstract types known by the system, and selects desired procedures. Generic procedures are specialized using the views and translated to the desired language; source code is served to the user as a file. Data conversion programs can also be generated. A related system can synthesize scientific programs from diagrams that connect data and physical and mathematical principles. The economics of automatic programming and combination of program resources across the web are discussed.

Keywords
Automatic programming, views, software reuse, generic algorithms, specialization, graphical programming, data conversion

INTRODUCTION
The exponential increase in power and decrease in price of computation, memory, and communication bandwidth have resulted in a great demand for programming. Automatic programming is needed to raise the level at which programs are specified (cf. megaprogramming [26]) and to automate the production of code.

Existing approaches to automatic programming and software reuse have been monolithic: a user must make a large commitment of time, money, and training to adopt an automatic programming system, and then the decision is often all-or-nothing: the whole application may have to be written using a single language or set of conventions. An alternative approach is an automatic programming server on the World Wide Web that can write programs on demand and serve them to a user.

Such a system has been implemented.1

The Automatic Programming Server (APS) is based on reuse of generic procedures through views. A generic procedure specifies an algorithm in terms of abstract data types. A view [19] describes how a user's concrete type corresponds to an abstract type. Compilation of the generic through a view results in a version of the generic that is specialized for the user's data and optimized. The resulting program can be mechanically translated to a desired application language (currently, Lisp or C) before being served to the user.

The APS provides:

• an interface with which the user can define application data types,
• graphical interfaces that make it easy to create views,
• a library of generic procedures for each abstract data type, and
• on-line documentation of the generics.

The interface of the APS is easy to use. The user is assumed to understand the application data types and the basic concepts of the abstract types. Most interaction between the APS and the user is by means of menus and graphical interfaces.

GENERICS AND SPECIALIZATION
The APS is a knowledge-based server that produces specialized versions of generic procedures. Its stored knowledge includes descriptions of abstract data types, definitions of generic procedures, and knowledge of how to make views. Fig. 1 shows how a generic procedure is specialized by compilation through a view. The GLISP language in which generics are written, the concept of views, and the processes of specialization and language translation are discussed below.

1An on-line demo of the Automatic Programming Server is available via http://www.cs.utexas.edu/users/novak : X windows is required to run the demo.
GLISP and Generic Procedures

GLISP [15] is a Lisp-based language with abstract data types; it is compiled into Lisp. GLISP has a language for describing data types in Lisp and in other languages. A GLISP type is analogous to a class in object-oriented programming (OOP); it specifies a data structure and a set of methods.

The GLISP language facilitates the writing of generic procedures. The syntax of GLISP allows any feature of an object to be accessed by a form that looks like a Lisp function call: (feature-object). This form can be used to access a field of an object, invoke a method, or locally change the type of an object to a view type; of course, it can also be used as an ordinary function call. As in OOP, all accesses to an object have the same form; this allows generic code to be written that is independent of the representation of the actual object to which it is applied. A feature with a given name can be stored as a field for one type of object, but be computed by a method for another type; the same generic code will work for either.

Although GLISP code has much in common with OOP conceptually, generic procedures are written in such a way that method invocations can be looked up at compile time and compiled in-line. In-line compilation removes the overhead associated with OOP method dispatching and allows optimization across message boundaries; because of this, specialized code often incurs no performance penalty compared to handwritten code. In addition, the specialized code does not require a runtime message dispatching facility and can be translated to an ordinary programming language such as C.

Views

A view describes how a concrete type implements an abstract type. A concrete type may differ from an abstract type by having its own data structure, names of data fields, and units of measurement of data; a concrete type may even use a different set of variables to represent similar data (e.g., polar coordinates versus Cartesian coordinates). An abstract type is described in terms of a set of basis variables; any concrete type that implements a record containing the basis variables implements the abstract type. Alternatively, a concrete type can implement an abstract type by providing a set of methods that emulate a record containing the basis variables; this is analogous to a wrapper or adapter class in OOP [2].

A view is a GLISP type that has the concrete type as its stored form, has the abstract type as a superclass, and defines methods that emulate the basis variables of the abstract type, using the concrete type as storage. In general, methods must be provided not only to read each basis variable, but also to write basis variables and to make a new instance of the concrete type given a set of values of basis variables. Methods that store values of basis variables must maintain their independence, so that a store into one basis variable does not change the value of another.

It can be somewhat difficult to construct correct and consistent views. The APS provides convenient interfaces for the user to specify correspondences between a concrete type and abstract type; then the system constructs the views automatically. There are separate tools for specification and construction of data structure views and mathematical views.

Specialization

A generic procedure is specialized by compiling it through a view. The generic procedure is written in terms of the basis variables of the abstract types of its arguments. Each access to a basis variable in the generic procedure becomes an invocation of the corresponding method of the view type; the method is compiled in-line as accesses to the concrete record, with data translation operations as necessary. Compilation of a generic thus produces a specialized procedure that operates directly on the concrete data.

Figure 2: Pizza Viewed as Circle

As a simple example, suppose that the user has a C data
structure that represents a pizza, and that the size of the pizza is represented by a diameter field. Using the APS, the user can graphically connect the diameter to a corresponding diameter "button" on a picture of a circle (Fig. 2). A program to calculate the area of a pizza can then be produced by specialization:

```c
float area1 (pizza2)
    PIZZA *pizza2;
    {
        return 0.785398 * square(pizza2->diameter);
    }
```

Although the abstract type circle is defined in terms of a radius variable, the APS uses symbolic algebra to make an appropriate view for pizza using diameter. The specialized program has been optimized and has no extra costs.

**Language Translation**

Although Lisp is an excellent language for writing programming tools, it is not so often used as an application language. An APS must be able to produce code in widely used programming languages.

The GLISP data description language allows specification of records in other languages as well as Lisp. Data accesses to non-Lisp records are translated into Lisp code that can access simulated records within Lisp and can be translated into appropriate syntax in the target language.

Lisp code is essentially equivalent to syntax trees of the kind often used as intermediate code in compilers. It is relatively easy to unpars this code into the syntax of another programming language. Translation into C has been implemented in the APS. Translation occurs in two stages. The first stage is a Lisp-to-Lisp translation that includes use of pattern-based transformation rules and transformations performed by programs. These transform Lisp idioms into corresponding idioms in the target language and transform things that are legal in Lisp but not in some other languages (e.g., returning a value from an if statement) into legal forms. A second stage uses syntactic transformation rules to unpars the Lisp code into the syntax of the target language. A printing program produces nicely formatted and indented source code.

Because the translation process would be similar for many languages and because the syntax of the target language is expressed by patterns that are easy to write, it should be relatively easy to add other target languages to the APS.

Language translation allows a single copy of generic procedures to be used to generate programs in a variety of languages. It decouples the choice of programming tool from the choice of application language.

**USER INTERFACE**

An example of the main user interface screen is shown in Fig. 3. The interface has a set of input lines, in which the user can enter descriptions of data structures. A command menu is used to select actions to be performed by the system. The bottom of the screen is used to communicate results of actions to the user. The present interface is based on X windows; it is somewhat crude, but does allow interactive graphical access to the APS from a variety of Unix platforms.

In the example shown in Fig. 3, the user first defines a C record structure (shown in the first input line).2 The next task is to make a view of this data structure as an abstract type; the user selects the Make a View command, then specifies (by menu) that a data structure view is desired, then selects sorted-linked-list from a menu of available views. In making the sorted-linked-list view, the system will present menus to the user to select the field on which to sort and the direction of sorting; the user has selected an ascending sort on name. After the view has been made, any generic procedure associated with the abstract type can be specialized and delivered to the user. The user has requested a program to insert a record at its proper position in the sorted linked list. The generated program is shown in the bottom portion of the window (the window may be too small to contain the program, as in this figure; the full version of the program is served to the user as a file when the session is finished).

The user interface of the APS is easy to use; most of the user input is via menus and diagrams. The interface is based on concepts that a programmer already knows; this is important because learning to use a programming tool is often a major cost. Our intention is that the APS should be usable by a programmer with little training in the tool itself.

**Data Definition**

The APS accepts the user's data definition. This gives the user the freedom to select whatever data representation is needed, e.g., for efficiency or for compatibility with existing hardware or software. It allows the APS to be used to generate code that works with existing systems. It eliminates any requirement that the user conform to the APS; instead, the APS conforms to the user's data.

The description of data can include units of measurement for numeric data; appropriate unit conversions [18] are produced automatically as needed.

---

2This example shows a Lisp description of a C structure. Clearly, a C user would prefer to enter the C form; a parser for C data declarations is straightforward but not yet implemented.
Follow these steps:
1. Specify your data structures, one at a time, in the lines below. Click mouse in an input line; enter input on that line followed by Return. When done, click Define Data to define that type.
2. Click Make a View to view your data types as abstract data.
3. Click Make Programs to select the programs you want.

Your data structure:

\[
\text{e.g. } (\text{part } \text{(list } (\text{name string}) \text{ (size integer)} \text{ (next } (^{\text{part}})))
\]

or for C, \( (\text{crecord cpart } (\text{name string}) \text{ (size integer)} \text{ (next } (^{\text{cpart}}))) \)

\[
(\text{crecord part } (\text{name string}) \text{ (size integer)} \text{ (next } (^{\text{part}})))
\]

/* The following code was produced by the Automatic Programming */
/* Server, University of Texas at Austin, 09 Apr 1996 11:55:05 */
/* INSERT inserts a new element at the appropriate place in a */
/* sorted structure, modifying the existing structure. */
/* The argument type is the record type of the structure. */
PART *sll_insert_1 (lst, new)
    PART *lst, *new;
{
    PART *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

Figure 3: User Interface Screen
This feature is important to allow combination of data sets that use different units.

**Data Structure Views**

A data structure view describes how a user record implements a data structure such as a linked list or AVL tree. The data structure views that are currently supported by the APS demonstration include **linked-list**, **sorted-linked-list**, **binary-tree**, and **avl-tree**. A data structure typically involves a record containing one or more pointer fields. If the user's record contains the pointer fields, it is used as specified; if not, the system will make an appropriate carrier record containing the user's data. For example, an AVL tree that stores words (strings) can be made simply by specifying a string as the data and requesting an **avl-tree** view. A carrier record is easily made by substituting the user's data as the contents of a record pattern that has the appropriate view choices pre-defined. When making a view, the system asks questions of the user (usually by menu) as needed to specify the desired view; for example, in making a **sorted-linked-list** view, the system will ask what data to sort on and whether to sort **ascending** or **descending**.

**Mathematical Views**

Mathematical views describe data that represent mathematical objects such as points, line segments, or circles. Such objects can often be represented in different ways depending on the variables chosen, e.g. a circle could be described by specifying its radius or its diameter. A system called MKV ("make view") [19] allows mathematical views to be specified by graphical connections between a diagram that represents the mathematical object and a menu of fields of the user's data. MKV performs symbolic algebra to convert between the user's representation and the abstract representation used in generic algorithms. Optimization is performed on the generated code, so that the specialized algorithms that are produced are often better than those written by human programmers.

Examples of the MKV interface used by the APS are shown in Fig. 4. In the first example, a concrete representation of a vector as **range** (in feet) and **bearing** (in degrees, counter-clockwise from North) is connected to a diagram of a **vector**. A second example specifies a vector in a different way, using the **slope** of the line and its x value in meters. The data structure descriptions for these are as follows:

(rb1 (list (range (units integer ft))
  (bearing (units integer deg))))

(rb2 (list (slope real)
  (x (units integer m))))

Mathematical views are important not only for specialization of generic procedures associated with the mathematical objects, but also because they enable automatic data conversion between different representations of similar objects. The following section presents a program generated by the APS to translate between the two vector representations above.

**Data Translation**

\[
\begin{align*}
    t_1 & \xrightarrow{\text{view}_1} a & \xrightarrow{\text{view}_2} t_2
\end{align*}
\]

Figure 5: Application Data Related by Views

A view describes how a concrete type implements an abstract type, perhaps using a different set of variables than the abstract type. The view contains a method to compute each variable of the abstract type from the data of the concrete type; it also allows an instance of the concrete type to be created given a set of values of the basis variables of the abstract type. Therefore, if two concrete types \( t_1 \) and \( t_2 \) each have a view as the same abstract type \( a \) (Fig. 5), it is possible to translate from one concrete type to the other through their common view. With optimization, it is possible
to create an efficient translation program that bypasses the basis variables when appropriate [19].

The APS allows a user to obtain programs to translate data. To use this feature, the user first defines two data types and makes a view of each as the common view type. Then the Convert Data command can be used to obtain a conversion program. A data conversion program can translate between significantly different mathematical representations of data (such as polar and Cartesian representations of vectors) and can automatically convert units of measurement when needed. An example of a data translation program is shown in Fig. 6; this program translates between the types rb1 and rb2 shown above. This translation involves different choices of variables for the two vectors, different representation of a variable compared to the abstract type (the bearing is represented as degrees clockwise from North, while the abstract type assumes radians counter-clockwise from East), and translation between different units of measurement (feet and meters) in the two types.

; The following code was produced by the
; Automatic Programming Server, University of
; Texas at Austin, 11 Apr 1996 11:21:51
; Arg types:  RB1
; Result type:  RB2
(DEFUN RB1-TO-RB2 (VAR-RB1)
  (LET ((VAR-RB1-VIEW VAR-RB1))
    (LIST (TAN (- 1.570796
            (* 0.017453
                (CADR VAR-RB1-VIEW))))
      (ROUND
        (* 0.304800
           (* (CAR VAR-RB1-VIEW)
               (COS (- 1.570796
                      (* 0.017453
                          (CADR
                           VAR-RB1-VIEW))
                      ))))))))

Figure 6: Data Translation Program

Documentation
The APS includes on-line documentation for generic procedures and for concepts (such as AVL tree) provided by the APS. For each generic, there is stored documentation of the function, its arguments, its result type, and its effects. The documentation can be read by the user to determine which generics are desired. Documentation is organized as a tree structure; typically, the first level of the tree is broken down by abstract type and the next level by name of concept or generic procedure. Documentation is also incorporated into the generated code as comments. Comments are generated to describe the argument and result types of specialized procedures when appropriate.

The existing form of documentation seems adequate, assuming a somewhat knowledgeable user. A valuable addition would be access to documentation by keywords or even questions in natural language.

PROGRAMMING BY DIAGRAMS
A program called VIP [16] (for View Interactive Programming) uses a graphical interface similar to the one used by MKV for making mathematical views. VIP allows development of scientific programs by connecting diagrams that represent data and physical and mathematical principles. Data diagrams are menu-like representations of data fields. Diagrams for physical principles are pictorial and have “buttons” that represent variables. The user begins a session with VIP with a workspace that is blank except for boxes that represent input data. The user can select boxes that represent physical principles and add them to the workspace. Connections between boxes are made by clicking buttons or menu items; a connection signifies that the connected variables are equal.

Once a complete diagram has been specified, a program is constructed from the diagram by dataflow. Initially, input variables and constants are considered to be defined. When a variable is defined, it is propagated into boxes to which it is connected. A box may have equations associated with it; solving the equations symbolically may define other variables, which can be further propagated. As a side effect, code is generated as variables are solved. The result is a program that calculates the desired results from the given inputs. Unit conversion is performed automatically.

Fig. 7 shows a small example problem: calculation of the position of an aircraft from data provided by an air search radar. We assume that the radar provides as input the time difference between transmission and return of the radar pulse, as well as the angle of the radar antenna at the time the return pulse is detected. When the radar illuminates the aircraft, we assume that the aircraft transponder transmits the identity of the aircraft and its altitude. The position and altitude of the radar station are assumed to be known. These items are the inputs to the program.

In constructing the program with VIP, the user models the travel of the radar beam as uniform-motion. The user selects the Physics command, then kinematics from the Physics menu, then uniform-motion from the kinematics menu. The input value TIME-DIFF is connected to the time button \( \tau \) of the motion. Next, the user selects Constant and obtains the constant for the speed of light, denoted \( c \), and connects it to
the velocity \( \mathbf{v} \) of the motion. The distance \( d \) of the motion then gives the total (out-and-back) distance from the radar to the aircraft; by dividing this distance by 2, the one-way distance is obtained. This distance is connected to the hypotenuse of a \textit{Geometry} object, \textit{right-triangle}. The difference between the altitude of the aircraft and the altitude of the radar is connected to the \( y \) of this triangle. The \( x \) of this triangle is then the distance to a point on the ground directly underneath the aircraft. This distance and the angle of the radar give a range and bearing to the aircraft from the radar; by connecting these to another right triangle, \( x \) and \( y \) offsets of the aircraft from the radar are obtained. These are collected to form a relative position vector, \textit{RELPOS}, which is added to the radar’s UTM (universal transverse mercator) coordinates to form the output. Note that the + operator is generic and can be applied to structured objects such as vectors so long as the operator is overloaded for the structured objects in GLISP.

While the process described above is rather lengthy when described in words, the time taken by an experienced user to create this program using VIP was less than two minutes. Note that this problem involves several instances of conversion of units of measurement, a physical constant, and algebraic manipulation of several equations; all of these were hidden and performed automatically. A GLISP program is produced by VIP and compiled and mechanically translated into C; Fig. 8 shows the result.

This example calculation is conceptually simple. However, for a human programmer, the requirements of performing algebraic manipulation of equations, converting units of measurement, and finding values of physical constants keep it from being a simple program. VIP hides these sources of programming difficulty and potential error.

An interface similar to that of VIP might be used for a more advanced APS. VIP is powerful because it allows the user to specify the architecture of an application program without having to specify all the details. An APS based on connections of data sets and programs, with machine assistance to make the data connections fit correctly, could allow applications to be specified and constructed quickly and correctly.

**ECONOMICS OF AUTOMATIC PROGRAMMING**

The way in which most existing programming workbenches or automatic programming systems are delivered presents barriers to their use. For commercial systems, there is a cost barrier, including buying an expensive software package and perhaps an expensive workstation on which to run it. Even free software requires installation, which is often difficult. Training to use the system can be a large cost. Finally, there
is a commitment barrier: it may be necessary to adopt a single language, programming system, or set of conventions for the entire application, making the choice all-or-nothing.

An automatic programming system (larger than the demonstration version described here) could remove these barriers. Since the APS runs on a server computer owned by the service provider, there is no cost to acquire the software or to install it. A variety of network browsers can run the APS, so there is no need for a special machine on which to run the software.

Charges for the software that is produced by an APS could be made on the basis of lines of generated code, with a certain amount of free trial service and a money-back guarantee. This would make the cost of trying out the APS very low, which would motivate more people to try it. Because the APS produces source code in standard programming languages, the commitment barrier is reduced; the APS can be used to produce parts of an application where it is efficient to do so, without requiring that the entire application conform to a single set of standards.

The beneficial features of an APS can provide an environment in which automatic programming can grow and evolve into an industry. Funds received from generating programs can support the expansion of the program library. Competition will be encouraged, and there will be room for niche players who write programs aimed at specific markets.

COMBINING WEB RESOURCES
The availability of programs and data resources on the World Wide Web is rapidly increasing. It is likely that users will wish to combine data sets from different sources and to use the data with programs that were not designed for those data representations. As described above, the APS can automatically generate data conversion programs given two views of data as the same abstract type. If owners of data sets and programs each make a view of their data representation as a standard generic representation, these views could be retrieved via the net and used to automatically generate the data conversions. The number of views needed is equal to the number of data representations, and each data owner only needs to understand the local data and to relate it to the common generic model using a graphical interface.

Data conversion programs could be generated on request by agent programs; requests could be stated using a standardized language such as KQML [4]. If program generation is available as a web service, agents can use this facility without having to implement automatic programming themselves.

RELATED WORK


Sinapse [8] generates programs to simulate spatial differential equations. KIDS [23] transforms problem statements in first-order logic into programs that are highly efficient for certain combinatorial problems; it has a sophisticated user interface. Elf [22] generates wire routing algorithms for integrated circuits.

CONCLUSIONS AND FUTURE WORK
While the library of generic procedures currently offered by the APS is small, it is clear that the APS is useful in making specialized versions of these procedures. A user can obtain specialized programs to maintain an AVL tree—about 200 lines of C—in a few minutes, whereas a human programmer might take days to code and debug a similar program, even given a reference such as Knuth [9]. Humans often find mathematical algorithms difficult to derive and code, and they may make mistakes. Tests of human subjects have shown that a function to calculate signed distance from a directed line segment to a point takes from half an hour to an hour to write, and some of the programs contain errors, are inefficient, or have mathematical problems (e.g., possible division by zero). The APS can produce a correct and efficient program for this problem in a few minutes of user interaction. Specialization of carefully written generics is likely to reduce errors as well as programming cost.
Future work on the APS should include an ability to instantiate program frameworks from components. For example, a generic key-word-in-context (KWIC) index program framework can be instantiated by supplying a sequence of input records, a noise word dictionary, and a program for formatting entries for output [17]. Generic procedures such as those to maintain an AVL tree can be automatically instantiated as components to be used in the program framework. This approach allows customization of the program where desired, but allows minimal specification and automatic implementation for cases where standardized components are sufficient.

Graphical interfaces such as the one used by VIP may be good ways to specify larger programs. The VIP interface allows specification of major features such as connections between components; the details of the connections (e.g., temporal ordering of computations and algebraic manipulations) are filled in automatically. Intelligent defaulting and filling in of detail are essential to make graphical interfaces usable.

A future APS could add value by supplying data resources as well as programs. For example, dictionaries, almanac data, historical data bases, stock market data, and geographic databases could be furnished. Data subsets could be derived and included as parts of programs where appropriate. Real-time data resources such as current weather data could be furnished by supplying web links together with appropriate data translation programs. Since data formats for these data resources could be predefined, the user would not have to understand the details of data sets in order to use them.

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

REFERENCES


