A REWRITE RULE MACHINE: PROGRAMMING BY GENERIC EXAMPLE

UNCLASSIFIED

F/G 9/2

NL
A REWRITE RULE MACHINE

Programming by Generic Example

Final Report
July 1986

By: Joseph Goguen, Program Manager

SRI Project ECU 1243

Prepared for:

Office of Naval Research
Information Sciences Division
800 N. Quincy St.
Arlington, VA 22217-5000

Attn: Dr. Charles Holland, Code 1133

Contract No. N00014-85-6-0417

SRI International
333 Ravenswood Ave.
Menlo Park, CA 94025
(415) 326-6200
TWX: 9100-373-2046
Telex: 334486
Table of Contents

1 Introduction 1
2 Data Structures 3
3 Examples of Programming by Generic Example 7
4 Icons 10
5 Animation 11
6 Some Problems in Natural Multimedia Interaction 12
7 Discussion 13
8 Appendix: More Specifications 13

List of Figures

Figure 1: Some Default Tree Representations 4
Figure 2: Some Linear Data Type Representations 5
Figure 3: Iconic Variations of Data Type Representations 6
Figure 4: Representations for Card Files 6
Figure 5: Some Representations of Generics 7
Figure 6: Graphical Representation of an Action on Stacks 8
Figure 7: Iconic Representation of a Rewrite Rule 9
Figure 8: Tree Form of a Rewrite Rule 9
Programming by Generic Example

Joseph A. Goguen
SRI International
333 Ravenswood Avenue
Menlo Park CA 94025

Abstract

This paper presents some techniques for programming with iconic representations. These techniques promise to make programming in suitable ultra high level languages significantly easier and more intuitive. The languages that we have in mind are based on rewrite rules and/or object-oriented programming, and have user-definable abstract data types. One technique uses the notion of constructor (from the theory of algebraic specifications of abstract data types) to automatically generate graphical representations for data values. Another technique permits defining rewrite rules, as well as methods (in the sense of object-oriented programming), by the direct manipulation of iconic representations of generic examples of data values. Some illustrations are given, based on the OBJ functional programming language and its extension to object-oriented programming.

1 Introduction

It is notoriously difficult to develop, understand, debug, modify and maintain programs written in the usual textual formats. Here, we suggest programming by generic example as a way to significantly improve the programming process for suitable languages, by supporting the direct manipulation of multimedia iconic representations of program meaning, rather than merely textual representations of program syntax. The most important medium is graphics, but mouse manipulations, audio and even text can play auxiliary roles. Our intention is to make programming as direct and physical as possible, in contrast to the comparatively arbitrary conventions of standard textual representation. One could also generate animations of running programs, for debugging and documentation. Such displays can be made hierarchical (as in VLSI design systems) to avoid indigestible detail.

Our discussion focusses on programming-in-the-small, that is, on the construction of algorithms. It seems easier to support programming-in-the-large with graphics, using the

1Supported in part by Office of Naval Research Contracts N00014-82-C-0333 and N00014-85-C-0417, and a gift from the System Development Foundation.
well-known building block metaphor to display generic modules and various notions of imported module hierarchy, with ideas like those in OBJ2 [Futatsugi, Goguen, Jouannaud & Meseguer 85] and Clear [Burstall & Goguen 77, Burstall & Goguen 81] as a semantic basis.

The approach reported here is intended to be helpful to the Rewrite Rule Machine (RRM) project at SRI International [Goguen, Kirchner, Leinwand, Meseguer & Winkler 86]. As such, it is oriented towards languages, like OBJ2 [Futatsugi, Goguen, Jouannaud & Meseguer 85], that are based on rewrite rules. The intention is that the RRM should execute such rules with enormous efficiency. One of these languages, called FOOPS [Goguen & Meseguer 86], combines the power of abstract data types with that of object-oriented programming, and seems an especially natural candidate for programming by generic example. We hope that the research reported here will make programming in ultra high level languages like OBJ2 and FOOPS significantly easier and more intuitive.

Object-oriented programming, which originated in the Simula language [Dahl, Myhrhaug & Nygaard 70], is simple and intuitive, since it was developed for simulating real world objects: a method may generate or modify members of a class of objects, where each object has its own state. For example, the class Stack may have methods to push, pop and create new stacks; there may also be functions which query the top and height of a stack. Classes may have subclasses; for example, a class NatStack of natural numbers might have a subclass OrderedNatStack, whose elements must be maintained in decreasing order. Programming by generic example supports object-oriented programming by directly manipulating the graphical representations of objects; for example, one could move a data item from the top of one stack to the top of another by “picking it up” with a mouse, carrying it over, and then “putting it down” on the second stack.

There has been a great deal of work on various approaches to “visual programming.” For a recent collection, see [Computer 85]. Two classical systems are Thinkpad [Rubin, Golin & Reiss 85] and Pecan [Reiss 85]; a more recent system emphasizing programming-in-the-large is PegaSys [Moriconi & Hare 86]. [London & Duisberg 85] mention the connection with object-oriented programming (through Smalltalk [Goldberg & Kay 76]) and also have an interesting approach to animation. But none of this work attempts to automatically generate displays for data types, or indeed, attempts to deal with data abstraction in a systematic formally based manner.
2 Data Structures

Our first step is to provide graphical representations for values from abstract data types; these should be both suggestive to users and relatively easy to generate. It is known that every (computable) abstract data type has a finite set of abstract constructors that are sufficient for defining its values \([17, 9]\); in practice, this is a relatively small set. By definition, a set of constructors provides a (minimal) set of functions for constructing every value of an abstract data type; thus, every such value is given by an expression consisting only of constructors. The usual tree representation of this expression yields a default graphical representation, having constructors labelling all its nodes.

Let us consider two examples, arithmetic expressions and S-expressions (in the sense of Lisp). Figure 1 shows the default tree representations that are generated for values of these types, based on constructors as described above. The expressions represented are

\[ x + (2 * (y + z)) \]

and

\[ A . (B . ((C . nil) . D)) . \]

We can describe the arithmetic expression and S-expression abstract data types using notation from OBJ2. Although these OBJ2 textual representations are not what the user would actually deal with, they are useful for making explicit the connection with the underlying logical formalism, which (in this case) is initial algebra semantics for equational logic [13]. The basis for a non-textual presentation of these data abstractions would be a structural editor, with mouse-driven graphical menu selection of commands, with an icon editor\(^2\), and with dynamic object manipulation for defining equations. Since it is difficult to present the dynamic user interaction with such a system in this paper, which of course is purely static, we present static OBJ2 text; this also facilitates comparison with our other papers on OBJ.

OBJ2's basic entity is the object, a module encapsulating some executable code. The keywords \texttt{obj} \ldots \texttt{endo} delimit the text of the object. Immediately after the initial keyword \texttt{obj} comes the object name, \texttt{AEXP} or \texttt{SEXP} for these examples; then a declaration of what object(s) are imported, in the examples above, \texttt{INT} or \texttt{ID} (the built-in types for integers and identifiers, respectively). This is followed by declarations for new data sorts, here \texttt{Aexp} or \texttt{Sexp}, and then subsort declarations, indicating that integers are considered to

\(^2\)Note that one can use icons instead of character strings to represent module, sort and operation names.
Figure 1: Some Default Tree Representations

here Aexp or Sexp, and then subsort declarations, indicating that integers are considered to be arithmetic expressions, and that identifiers are considered to be S-expressions. Finally come declarations for the constructors, indicated by the keyword cop (which stands for “constructor operation”). These declarations include both constants\(^3\) and operations, such as x, nil, _* and _., each with information about the distribution and sorts of arguments and the sort of the result; underbar characters “_” are used to indicate argument places for mixfix operators; thus _._ is infix and length._ would be prefix. After the operator declarations, some equations might be given; however, these two examples do not involve any equations. (These examples take some liberties with OBJ2 syntax, for the sake of simplifying the present exposition.)

object AEXP is
  importing INT
  sort Exp
  subsort Int < Exp
  cops x,y,z : -> Exp
  cops +_,_,_ : Exp -> Exp
endo

object SEXP is
  importing ID
  sort Sexp
  Subsort Id < Sexp
  cop _._ : Sexp Sexp -> Sexp
  cop nil : -> Sexp
endo

---

\(^3\)Constants are considered to be a special kind of operation having an empty string of arguments for input.
For many common data types, we can automatically generate default representations that are more iconic than the default trees. In particular, for (linear) sequences of characters and for stacks of integers, we can get the usual linear representations, as shown in Figure 2 for the expressions

$$\text{add}(\text{f}, \text{add}(\text{o}, \text{add}(\text{o}, \text{add}(\text{p}, \text{add}(\text{s}, \text{nil}))))))$$

and

$$\text{push}(\text{5}, \text{push}(\text{7}, \text{push}(\text{211}, \text{push}(\text{9}, \text{push}(\text{329}, \text{empty})))))),$$

respectively. OBJ2 code for these two data types is given in the appendix, Section 8. This method works for linear data types, having a signature of constructors with only one new sort and with every new operation having at most one argument of that sort. If there is just one constant, the default representation assumes that it represents the empty structure, and gives it the empty representation. If there is more than one non-constant constructor, it will be necessary to label cells with constructor names; otherwise, this can be omitted.

Users could also be given interactive support for generating icons for constructors. This would permit still more iconic variations, like those shown in Figure 3 for Stacks and S-expressions. The support provided should include an icon editor and some simple options for combining icons. For example, one should be able to draw the “spring” shown in Figure 3 with an icon editor, and then indicate that it should be attached to the bottom of the default (linear) stack representation, displayed up-to-down, rather than left-to-right as it is in the default in Figure 2; similarly, one should be able to create the left and right “sides” and attach them to the top item on the stack, as in Figure 3.

Let us consider a somewhat more complex, but still linear, data type, a file of library cards, each having an accession number, an author, and a title. Just two constructors are involved, card with four arguments, and the constant empty. Figure 4 shows a default, a default linear, and an iconic representation for the file

$$\text{card}(17381, \text{W. Daniel Hillis, The Connection Machine}, \text{card}(16230, \text{Jeffrey Ullman, Computational Aspects of VLSI}, \text{empty}))$$

The OBJ code defining this type is just
Figure 3: Iconic Variations of Data Type Representations

```
obj CARDFILE is
    importing NAT, CHARSTRING
    sort File
    cop empty : File
    cop card : Nat Charst Charst File File
endo
```

Figure 4: Representations for Card Files
3 Examples of Programming by Generic Example

Iconic programming by generic example proceeds by indicating how to handle generic examples of data structures, by simply performing direct manipulations on their iconic representations; of course, one must also show how to handle the constants that occur in data type signatures (such as nil). There is a simple default representation for generics of linear abstract data types: first display one constructor cell, then a cell containing "...", then another constructor cell (this is the "generic" cell), then another "..." cell, and then a final constructor cell. By convention, subscripts will be used to indicate these elements, 1 for the first, k for the generic, and * for the last. For non-linear data types, something similar can be done, but laying out the representation may become a problem. Examples of these conventions are shown in Figure 5, for character sequence, S-expression, and card file.

![Figure 5: Some Representations of Generics](image)

In an example of programming by generic example, the programmer might move a data item from the top of one stack to the top of another with his mouse by “picking up” the top item from a generic stack representation (i.e., popping the first stack), then “carrying it over,” and finally “dropping it” onto the top of a generic representation of the second stack (i.e., pushing it onto the second stack); Figure 6 is intended to suggest these actions. The system will take this behavior as (one case in) the definition of a method. Such a behavior might be part of an algorithm that uses two ordered stacks of values to sort an input list by inserting new values from the list one at a time, flipping values from one stack to the other until the new value lies between the two top stack values. The sorting program will consist of the rewrite rules generated by a programmer's manipulations of three these generic structures. Of course this algorithm, which might be called “Tower of Hanoi” sorting, is not very good for concurrent computation—in fact, it is a very inefficient sequential algorithm—but it is good for illustrating programming by generic example.

Note that the system can automatically check whether or not all cases have been covered
Figure 6: Graphical Representation of an Action on Stacks

by the manipulations that a user provides. This is because the system knows that there are
two constructors for Stack, and therefore knows that two cases have to be covered: the
"initial" (or base) case of the empty stack, and the "loop" (or recursion) case, of a non-
empty generic stack constructed by push. In general, such a check can be more complex,
because of equations holding among constructors, and will require something like Thiel's
algorithm [Thiel 84].

For another example, let us consider a "simple library card file" abstract data type,
consisting of a list of cards, each with an accession number \( N \) and an author \( A \), and the
problem of writing a function, called author, to search for the author \( A \) associated with a
given accession number \( N \). The generic icon for the card file is like that shown in Figure 4,
but a little simpler since there is no title variable. Figure 7 shows a graphical form of a
rule defining the author function; here we use \( \# \) as a graphic symbol for the number
variable. Note that the user would actually define this rule by mouse manipulations, and
something special (e.g., with mouse clicks) would be needed to insure that the \( \# \) variable
occurs in the two places where it is shown. Also, note that Figure 7 shows yet another
graphical representation for function application, here of the author function to its
arguments \( \# \) and a generic card file. Figure 7 is somewhat misleading because it is static,
whereas what the programmer would actually do is dynamic. The lefthand side shows the
variables involved, including \( N \), which is the key for the search for an author \( A \) with that
accession number, and provides a template for matching. The righthand side is created by
first grasping the A cell with the mouse and then putting it down. We hint at the dynamic aspect of the rewriting process by placing a "fat arrow" between the two situations. The resulting program in text format is somewhat sophisticated; it consists of a single rule with a single tree variable $F'$ which will match any initial segment of a card file,

$$\text{author}(N, F'(\text{card}(A, N, F))) \Rightarrow A,$$

where $F$ is a variable denoting the rest of the file. The complete OBJ code for this simple card file is given in the appendix, Section 8. A tree form of the above rule is shown in Figure 8; again, we use the "fat arrow" convention. We also use a graphic symbol for the number variable, # instead of N, and another for the author variable A.

Figure 7: Iconic Representation of a Rewrite Rule

Figure 8: Tree Form of a Rewrite Rule

This form of rule is interesting because it is more powerful than the usual rewrite rule;

---

$^4$More technically, $F'$ is a "second order monadic variable."
however, it is actually equivalent to a facility that is already is OBJ2, called “rewriting modulo associativity.” This is not the place to discuss this equivalence; but it is reassuring to know that we are not really getting outside the framework of first order equational logic, which provides the logical foundation of OBJ and FOOPS.

Notice that for the object-oriented case, it will be necessary to distinguish between manipulations that delete a cell (like pop for stacks) and those that only copy a value from a cell (like top for stacks). This could be indicated, for example, by using a double push of a mouse button for the deletion case, and a single push for the copy case.

4 Icons

An icon is not merely a visual symbol, but rather a sign, possibly in mixed media, that is perceived to correspond to what it represents. This corresponds to the original sense of icon in [Peirce 65], as a “sign which refers to the object that it denotes by virtue of characters of its own.” Pierce carefully distinguishes an icon from a symbol, which is a “sign which is constituted a sign merely or mainly by the fact that it is used and understood as such.” Notice that in current computer jargon, the word “icon” is used for any graphic sign. Pierce also distinguishes the case of a sign $z$ being used as an index for an object $y$ if $x$ and $y$ are regularly connected, in the sense “that always or usually when there is an $x$, there is also a $y$ in some more or less exactly specifiable spatio-temporal relation to the $z$ in question” [Alston 67].

Of course, sign is the most general class; that is, everything is a sign. However, the three kinds of sign cannot always be rigidly distinguished; for example, the “smiley face” sign used for the author variable in Figure 8 actually has something of the character of both an icon and a symbol.

Not only objects, but also relationships and situations can be represented iconically. For example: the magnitude of a quantity might correspond to the size of its representation; the temperature of an object might correspond to the redness of its representation; the relation “followed by” might be represented as an ordered pair of “pointings-to” by a mouse; and an error state might be represented by the sound of a siren.
5 Animation

Once the constructors are known and icons have been chosen to represent them, it is possible to automatically generate a display for any given state of the runtime environment. This capability could be used, for example, to animate programs, i.e., to produce sequences of "frames" showing how the data structures change as the program is executed. This kind of animation will clearly be useful for understanding and debugging programs. Our basic method of program construction is a kind of inverse to this animation "playback," namely the construction of methods of transitioning from one frame to another, by the programmer's direct manipulation of icons.

It does not suffice to provide pretty pictures on an ad hoc basis, for example, by attaching display commands to existing code. In fact, it will be much better if the code is produced from the direct manipulations on the generalized icons. These can also be used as the basis for animation; since the user himself chooses to produce the code using certain representations, we can assume that he would also like to see it displayed that way.

It will be important to provide a zooming capability, in order to deal with large complex programs without the overwhelming detail. In fact, the user should define an abstraction to serve as the interface that he wants to see animated. The module and view capability of FOOPS, generalizing that of OBJ2 to object-oriented programming, seems ideally suited for this purpose.

An interesting point is that users will often prefer to see continuous, gradual movement of one situation to another, rather than a sudden discrete jump. For example, consider the action of "carrying" an item for the top of one stack to another described above. If it just jumps, it will even be hard to determine where it went; but if it moves continuously from one place to the other, users will understand what is intended much more easily.

A possibility which seems feasible, and which it would be very interesting to pursue, is to automatically generate a "sound track" for these "animated movies" of program execution. A great deal of research has been done on the structure of explanations (for example, [14]) and on how to generate them; and of course, generating the sounds of speech is no longer a difficult problem.
Some Problems in Natural Multimedia Interaction

This section mentions two problems that seem important, but have so far received little attention. Good solutions to these problems could be enormously helpful for the kind of system described in this report.

It is clear from experience that not every mode of interaction with a complex system is equally effective. The display must not be overcrowded or overcomplex: it must highlight the right details and hide others; and it must help to structure interactions in the right order. A proper icon for a programming concept may involve not only a display primitive, but also some understanding of the context in which it appears. In fact, we may want what is displayed to change as the context does: sometimes it might be hidden, sometimes it might be highlighted, sometimes smaller, sometimes larger, and sometimes perhaps even displayed in a different form; also, it might appear in different relationships to other objects.

One way to explore this very rich problem area is by observing the performance of skilled humans working in the same role that we would like the system to take. This research method has produced some surprising findings about the comprehensibility of text-based programming language features (see the Smoothtalk language [4]). For example, Smoothtalk does not have any variables as such, but rather uses descriptions, such as “the previous number” or even “it.” Also, Smoothtalk’s loop construct does not have an explicit begin marker, and what is to be iterated over is only indicated at the end of the construct. These conventions, although very different from those of conventional programming languages, are how people actually describe programs in natural language.

A basic issue that has been little addressed is the proper ordering of modes in programming: sometimes the programmer should be creating new code, sometimes planning, sometimes debugging, sometimes explaining the reason for a choice, sometimes documenting a sequence of choices, etc. We would like to know what rules govern the sequencing of these different modes of interaction. Another important problem is how to integrate representations in various media. For example: When is text appropriate? How should it be integrated with graphics? When (if at all) is computer generated speech appropriate? When is speech recognition appropriate? How should color be used? How can information overload be avoided? Some work relevant to such questions can be found in [12] and [8], which studies how speech acts are sequenced in aviation discourse.
7 Discussion

This report has introduced programming by generic example as a basic programming style for functional and object-oriented programming. The main ideas have been the following:

1. programming by direct manipulation of graphical representation of generic abstract data type values;
2. interactive support for defining data structures and their representations;
3. suggestive default representations for common data type structures, based on their constructors;
4. natural multimedia interaction with the system; and
5. audio-visual animation of programs by displaying the changing states of basic data types, with explanation.

There are (at least) five layers that should be considered:

1. underlying mathematical and psychological principles;
2. choice of display primitives for a given data structure;
3. choice of what to display;
4. choice of how to display it; and
5. providing iconic interactive modes that the programmer can use to express his intentions.

We believe such considerations can lead to a very intuitive programming style for the Rewrite Rule Machine [11]. This style is in direct correspondence with the underlying rewrite rule computational model and also utilizes the full power of interactive computer graphics. Program production should be substantially improved by the systematic use of this programming style, particularly in connection with the use of its inverse, namely animation, to support debugging.

8 Appendix: More Specifications

This appendix contains OBJ code for the sequence and stack examples mentioned in the body of the paper, and also gives full details of the author program for simple card files; this brings out some further facets of OBJ.
obj CHARSTRING is
   importing CHAR
   sort Charst
   cop nil : -> Charst
   cop add : Char Charst -> Charst
endo

obj STACK is
   importing INT
   sort Stack
   cop empty : -> Stack
   cop push : Int Stack -> Stack
endo

Of course, one would also like to define other operations that are not constructors, such as head and tail for character strings, and pop and top for stacks; however, only the constructors are directly relevant to the problems discussed in this paper. Also, note that if we defined these abstractions as objects, in the sense of object-oriented programming, things would have to be a little more complicated, as described in [Goguen & Meseguer 86].

obj CARDS is
   protecting NAT ID
   sort File
   cop empty : -> File
   cop card : Id Nat File -> File
   op author : Nat File -> Id
   var N : Nat
   var F : File
   var F' : File*
   eq : author(N, F'(card(A, N, F))) = A
endo

Here, var indicates that a variable declaration will follow; the "*" in "File*" indicates that the variable F' is a tree variable. Note that the operation author is not a constructor.

References


END

9-86