REPRESENTING PROGRAMS FOR REASONING

Elaine Rich and Aaron Temin

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-84-02 February 1984

Abstract - This paper describes the program-representation language MIRROR. MIRROR has been designed to support automatic, high-level reasoning about programs of the kind that must be done by an intelligent help system. MIRROR facilitates such reasoning in two important ways. It makes program structures as static as program logic allows. And it makes the structure of a program correspond as closely as possible to the structure of important objects and operations in the problem domain. It also provides mechanisms for indicating the relationships between program components and their corresponding concepts in the problem domain.

Table of Contents

List of Figures

1 Introduction

There are some questions about a program's behavior that are best answered by looking at its code. Seldom used branches of code or unplanned interactions of values lie undocumented anywhere else. However, even when source code is available it is difficult for anyone other than the author to understand. We are building a system that will look at code and answer questions about the program's behavior. This system should be able to serve as a user-oriented help facility that is far more flexible than other such facilities, whose only ability is to parrot back canned responses, usually triggered by simple keywords.

For such a system there are two critical issues -- how to represent the target program and how to answer questions about its behavior. In this paper we look at the first of these issues, and describe a help program language (called MIRROR), discuss it's features, and consider the implications for source program languages such that programs in these languages could be mechanically translated into MIRROR.

The specific system that has motivated much of this work is a help system [Rich 82] for the document formatting program Scribe [Reid 80].[1] Many of the examples in this paper will be drawn from that system.

2 Program Representations

A given program can be represented in many different ways, depending on how it is used (see Figure 1). The code read and written by humans is in the source language, and the other versions are gotten by a (semi) mechanical translation from the source program, although admittedly programming environments blur this distinction. It is necessary to emphasize that MIRROR is not (necessarily) a language a human would read or write. Several properties, particularly some that introduce redundancy in the representation, make the language poorly suited for humans to write. Other properties make it poorly suited for execution. MIRROR is, however, a good candidate for replacing the question mark in Figure 1. The reasons for this will be explained below.

Task to be Done	Representation into Which the Program is to be Transformed
Executed directly	Machine code
Interpreted	Symbolic program tree + symbol table
Examined and modified	Indented format + auxiliary information such as name reference charts
Proved correct	Verification conditions
Used to answer questions about itself	?

Figure 1: Uses of a Program and the Representations Required

[1] Other approaches to the design of intelligent help facilities are described in [Kehler 80, Genesereth 78, Shapiro 75, Wilensky 82].

3 Answering Questions from Code

The main method that is available for answering questions from code is *program slicing* [Weiser 81], and the goal of MIRROR is to make this as easy as possible. The question-answering process can be split into three parts:

1. Find out where to start the slice
2. Find the slice
3. Find out how to explain the results of the slice

The second step of this process has led to the consideration of how the syntax and semantics of existing source languages can be simplified (while maintaining equivalent overall expressive power). The other two steps provide the motivation for allowing MIRROR programs to be annotated with information that will allow the question answerer to perform the necessary mappings between objects in the problem domain and objects (such as variables and procedures) within the MIRROR program.

The following general properties must hold for a program representation that is to be used to answer questions about the program's behavior:

- It must be possible to determine a great deal about a program's behavior from a static analysis of its code.
- It must be easy to match code segments and the conditions under which those segments will be executed with important components of the questions to be answered.
- It must be possible to explain the behavior of an individual program segment in a single paragraph of reasonable length and intricacy.
- All code in the representation should be relevant to the domain of the program and of potential interest to a user. The representation should include as little "housekeeping" code as possible.

These general goals translate into the following specific features that are important in the representation language:

- Support for modularity that reflects a top-down decomposition of the program into manageable modules, from which we can infer what code to explain together, and what abstractions are reasonable.
- Typing mechanisms that help to constrain the search for particular objects of interest.
- Abstract data types, because routines connected to particular data types deserve different explanations than higher-level control code.
- Static and dynamic binding of types and sizes of variables, *as appropriate to the solution* instead of the programming system, to eliminate "housekeeping" code such as system dependent bounds checking. We call this *logical binding time.*
- Absolute minimization of aliasing, so that the use of program objects can be determined from a static analysis of specific code segments, regardless of how deeply nested the call to that code may be.
- Flattened (unnested) conditionals, so that the conditions under which a particular piece of code will be executed can be determined with as little search as possible.
- Separation of observing procedures (which do nothing but return values) from acting

procedures (which may modify system structures) so that the set of situations in which side effects need to be considered is as restricted as possible.

- Ability to return an arbitrary number of values from a function so that these values can be easily recognized when they are logically necessary.
- Control structures, such as loops, whose definitions correspond to the types that are used so that logical control flow is as explicit as possible.
- Informative procedure headings, because these headings summarize the function of the procedure and can be used without an examination of the procedure body.
- Annotations that describe each identifier so that mappings from question to code and code to answer can be done.

4 The Help Representation Language MIRROR

MIRROR has been designed to meet the criteria described above. MIRROR is similar to GLISP [Novak 87], though more restricted. An example of a piece of a MIRROR program is shown in Figure 3. In this section, specific language features that are motivated by the general requirements outlined above will be described. These requirements have much in common with good programming style and easy verification. Thus some MIRROR features will not appear to be new; others however are.

4.1 Modularity and Control

4.1.1 Encouraging Modularity

The importance of a good decomposition of a large program into a set of modules each of manageable size is now well understood, although the criteria to be used in forming that decomposition may vary [Parnas 72]. If good explanations are going to be generated from code, then the following three criteria are of paramount importance:

- Level of detail
- Size
- Complexity

The first of these is well accepted. It says that a program should be divided into modules that correspond to significant levels of abstraction in the problem itself. These abstractions can then be exploited in generating explanations. The second of these criteria says that a program should be divided into modules that are small enough that a description of a single module's behavior can form the basis of a simple, coherent explanation. This should be able to be done in one very short paragraph. The third criterion says that the structural complexity of a module should not be so great that a simple, understandable explanation cannot be produced. In the next section, the form of a MIRROR procedure that satisfies these criteria is described.

4.1.2 A Production Rule System

A MIRROR procedure is a production rule system (possibly consisting of only one rule). The major advantage of this is that a production rule makes an explicit association between each code segment (which forms the right hand side of a rule) and the guards (which form the left hand side of the rule) that must be true in order for the segment to be executed. The guards consist of a set of conditions combined

into a single boolean expression. The action part of each rule is either a sequence of four or fewer simple statements or it is a single loop with an optional initialization and/or closing statement. Thus the form of a production rule system is similar to that of the LISP COND statement. The simple case where there are no guards on a code segment is represented as a one-rule production system. The common if-then-else construct is represented as a two-rule production system. If there are more than two paths, they can be represented by as many rules as necessary and each will be at the same level.

A complete definition of the form of a MIRROR production rule system is given in Figure 2. This structure makes three guarantees to an explanation system that must reason about a MIRROR program:

1. The conditions under which a piece of code will be executed are clearly marked.
2. Because loops are difficult to explain, there can be no more than one per right hand side. Thus there is a limit on the complexity of each action.
3. No action will be too large to be explained in a single paragraph.

These guarantees support the modularity criteria described in the last section.

<production system> ::= (**RULES** <production set>)

<production set> ::= <condition action pair> | <condition action pair> <production set>

<condition action pair> ::= (<boolean expression> <action part>)

<action part> ::= <loop> | <statement sequence>

<statement sequence> ::= <simple stmt> <simple stmt> <simple stmt> <simple stmt>

<simple stmt> ::= <assignment statement> | <procedure call> | ϵ

Figure 2: The Form of a Production Rule System

Figure 3 shows an example of a three-rule system that appears in the MIRROR version of Scribe. It is the procedure that parses input from a user's file. It considers three cases: the next character is the character that signals the beginning of a command; the next character indicates an end of file; or the next character is an ordinary text character. We use **T** (for true) as a shorthand for the expression that is the conjunction of the negations of all of the previous guards. Thus it is only satisfied if none of the others is.

In order to simplify the semantics (and thus the explanation) of the guard notion, we require that the set of guards contained in a particular production system be mutually exclusive. Notice that this restriction makes MIRROR different from other guard-based systems, such as *guarded commands* [Dijkstra 78]. This restriction means that although the set of guards must be tested in some order at runtime, the behavior of the program can be determined without knowing what that order is. It is in general undecidable whether a given set of guards is mutually exclusive. But in most real programs, the logic causes them to be so. Sometimes to guarantee this it is necessary to add additional terms to the guard. An example of this is shown in Figure 4. Notice that in the revised version, the conditions under which statement B is executed can be determined locally by examining its guard. In the original version, doing so would require examining the guards of all of the preceding statements. Notice that the use of **T** (or true) as a guard does not violate this mutual exclusivity requirement since it is interpreted not as **T** but as a shorthand for the conjunction of the negations of all of the previous guards.

```
(ACTING-ROUTINE
  (PARSE-INPUT
    (FORMALS (NEXT-CHARACTER (CHARACTER BY-VALUE)
                  (ANNOTE (TEXT Next input character ))))
    (LOCALS (NEW-OBJECT OBJECT (ANNOTE (TEXT Next file object)))
            (STOP BOOLEAN (ANNOTE (TEXT Any more objects?)))))
    (RESULTS NEW-OBJECT STOP)
    (ANNOTE (TEXT Switch to correct routine for commands or text))
    (RULES ((COMMAND-CHARACTERP NEXT-CHARACTER)
            (SKIP-COMMAND-CHARACTER)
            (ASSIGNQ NEW-OBJECT (PARSE-COMMAND))
            (ASSIGNQ STOP NIL))
           ((EOF-CHARACTERP NEXT-CHARACTER)
            (ASSIGNQ STOP T))
           (T (ASSIGNQ NEW-OBJECT (PARSE-TEXT))
            (ASSIGNQ STOP NIL)))))
```

Figure 3: A MIRROR Production Rule System

Original production system:

```
(RULES ((NULL X) A)
       ((LISTP X) B)
       ((ATOM X) C))
```

Revised production system:

```
(RULES ((NULL X) A)
       ((AND (NOT (NULL X)) (LISTP X)) B)
       ((ATOM X) C))
```

Figure 4: Transforming Production Systems

4.1.3 Separation of Observing and Acting Procedures

The use of production systems to attach guards to code segments forces a clear-cut distinction between observing procedures (which may return values but must have no side-effects) and acting procedures (which may both return values and have side effects). Only observing procedures may occur in the guard portion of a production. This is important for two reasons:

- It reduces the space that must be searched by the matcher when it is trying to find code segments in which a particular action may occur. Only the action parts of each production system must be considered.
- It simplifies the search for the conditions under which a matched piece of code will be executed. If actions could occur in guards, then it would be necessary to know exactly the conditions under which a guard is tested in order to determine when the action it produces will occur. This eliminates the simplicity that is gained by requiring guards to be mutually exclusive.

Of course an alternative to this distinction would be to eliminate side effects entirely and take a totally functional approach to programming (such as that described in [Henderson 80]). We do not do that because there are situations in which that increases the difficulty of explaining what is going on. The reason for this is that functional programming obscures the distinction between modifying existing objects and creating new ones. This distinction has always been recognized as an important one from an implementation point of view, but it is, in fact, also important for the understanding of a program's logical structure. As an example, consider a piece of Scribe code that, as a result of a command in the user's file, updates the system state vector in such a way as to cause later output to be double spaced.

The state vector existed prior to the execution of that code. After execution of the code, there is a new value in the vector so we could simply view it as a new value that was produced by the code. But there are two other important things that are also true. First, the previous value no longer exists. And second, the new value is to be used everywhere in which the old value would previously have been used. (Another way to look at this is that the new value has the same name as the old value.) Both of these facts are obvious if we view the code that was executed as having had the side effect of modifying the state vector, while if we view the code as having created a new object, these facts can only be derived from a global analysis of the program. This same reasoning can also be applied to the modification of nonglobal objects that are passed to procedures as parameters.

Because of this need to represent the transformation of one value into another, MIRROR allows the modification of global variables (see Section 4.2.7) within acting procedures and it allows acting procedures to modify the values of their parameters (see Section 4.2.8). For an example of the use of such a modification to an input parameter, see Figure 7(*b*).

The argument given above on the importance of allowing the explicit modification of existing structures within acting procedures should not be interpreted as an argument against the use of function values. These values are important as they represent new objects that have been created. MIRROR allows both observing and acting procedures to create new objects and thus to return values.

MIRROR's ban on side effects in the left hand side of a production rule turns out not to be as restrictive as it may at first seem. If one examines existing LISP programs, in which the COND statement functions very much like the MIRROR rules, one sees that the majority of the cases in which side effects occur in the conditional part fall into one of the following three classes:

a. The side effect is in the first condition element. What is intended is that the side-effect-producing action should be performed and then the condition elements should be checked.

b. The side effect is in the last condition element and the corresponding action element is empty. What has happened is that the actual final condition element, "otherwise", has been omitted and the corresponding action has been shifted into its place.

c. The side-effect is a strictly local SETQ performed for efficiency. Although such side effects are not permitted in the MIRROR representation of a program, they may occur in the executable representation. If, however, the result that is being saved in the SETQ is important in the context of the problem domain, it should be computed outside the guard and given a name by which the explanation system can refer to it.

Figure 5 shows an example of each of these classes of side effects. Figure 6 shows how each of these programs will be represented in MIRROR.

4.1.4 Returning Multiple Values from Procedures

As was just described above, MIRROR views values returned from procedures as representing objects created by the operation of those procedures. In order for it to be clear exactly what has been created, it is important that the values that are returned reflect the created objects as closely as possible. To allow this, it is important that a procedure be allowed to return multiple values if it creates multiple objects. If this is not allowed, the procedure must either return a single artificial value that is constructed to contain the real values, or it must use modifiable parameters to return all but one of the created values, thus obscuring the distinction between modified and created values. As an example of the need for this capability, consider a game playing program that uses the recursive minimax procedure [Rich 83] to search a tree of game positions. This program is given as its input a node representing a particular game

```
(DEFINEQ (COMPUTE (LAMBDA ()
  (PROG (IN)
    (COND ((ZEROP (SETQ IN (READ))) (RETURN 0))
          ((GT IN 0) (RETURN (A)))
          (T (RETURN (B))))))))
```

(a)

```
(DEFINEQ (COMPUTE (LAMBDA ()
  (PROG (IN)
    (SETQ IN (READ))
    (COND ((ZEROP IN) (RETURN 0))
          ((GT IN 0) (RETURN (A)))
          ((RETURN (B))))))))
```

(b)

```
(DEFINEQ (COMPUTE (LAMBDA (X Y)
  (PROG (TEMP)
    (COND ((GT (SETQ TEMP (PLUS X Y)) 0)
           (RETURN (A TEMP)))
          (T (RETURN (B TEMP))))))))
```

(c)

Figure 5: LISP CONDs with Left-Side Side Effects

configuration. The program must return two values, the move to be made next and a number representing the backed-up evaluation of the merit of the input node.

MIRROR allows multiple values to be returned from procedures. To make the use of these values straightforward, it also provides a multiple assignment statement, which is a version of the standard MIRROR assignment statement, ASSIGNQ, discussed in more detail in Section 4.2.3. Thus the call to the game program described above would be written as

```
(ASSIGNQ
    (VALUE PATH) (MINIMAX NODE))
```

This will result in VALUE being set to the first value returned by MINIMAX and PATH being set to the second.

Multiple-valued procedures can also be composed with other procedures that require multiple input parameters, so their use is not restricted to ASSIGNQ statements and thus to the action parts of programs.

4.1.5 Looping Structures and Recursion

One part of the control structure of a program is described by its modularity. The other is described by control constructs such as looping and recursion. MIRROR provides such structures and depends on programmers to choose those that most clearly describe each program operation. Both iteration and recursion are allowed, since there are situations in which each is the easiest to explain. Figure 7 shows one example of each.

The program of Figure 7(*a*) can be described as follows:

> If there is one disk, move it to the goal. Otherwise, move n-1 disks to the temporary peg, move the remaining disk to the goal, and then move the n-1 disks to the goal.

```
(ACTING-ROUTINE
   (COMPUTE
     (FORMALS)
     (LOCALS (IN INTEGER (ANNOTE (TEXT Next integer)))
             (ANS INTEGER (ANNOTE (TEXT New maximum integer))))
     (RESULTS ANS)
     (ANNOTE (TEXT Compute based on difference))
     (RULES (T (ASSIGNQ IN (READ))
               (COMPUTE1 IN)))))

(ACTING-ROUTINE
   (COMPUTE1
     (FORMALS (INPUT (INTEGER BY-VALUE)
                     (ANNOTE (TEXT New value))))
     (LOCALS (ANS INTEGER (ANNOTE (TEXT Answer))))
     (RESULTS ANS)
     (ANNOTE (TEXT Compute function))
     (RULES ((ZEROP INPUT) (ASSIGNQ ANS 0))
            ((GT IN 0) (ASSIGNQ ANS (A)))
            (T (ASSIGNQ ANS (B)))))))
```

(a and b)

```
(ACTING-ROUTINE
   (COMPUTE
     (FORMALS (X (INTEGER BY-VALUE) (ANNOTE (TEXT First value)))
              (Y (INTEGER BY-VALUE) (ANNOTE (TEXT Second value))))
     (LOCALS (ANS INTEGER (ANNOTE (TEXT Answer))))
     (RESULTS ANS)
     (ANNOTE (TEXT Compute based on difference))
     (RULES ((GT (PLUS X Y) 0)
             (ASSIGNQ ANS (A (PLUS X Y))))
            (T (ASSIGNQ ANS (B (PLUS X Y)))))))
```

(c)

Figure 6: Left-Side Side Effects Removed

The program of Figure 7(*b*) can be described as follows:

> Go through the set, checking each element to see if it is bigger than the current maximum. If it is, then make it the current maximum.

Although both of these problems can be solved both iteratively and recursively, each has one relatively more natural solution.

4.2 Data

In order to be able to generate explanations of a program's behavior, it is important to understand the role of each object in the program. At one extreme, each object is a separate entity, and specific information about it must be recorded. For example, the Scribe variable LMARG contains the size of the left margin. At another extreme, all objects are the same. They are locations in memory. There is an intermediate level, though, at which objects can be grouped together with other objects on which the same operations can be performed. Objects within a group are very likely to affect each other, for example by assignment. Typing mechanisms provide a way to define these groups. Once these groups are defined, they provide a way to constrain the search of the question answerer to those objects that may influence the objects with which the question is directly concerned. The more detailed information about individual

```
(ACTING-ROUTINE
  (TOWER-OF-HANOI
    (FORMALS (START-PEG (STRING BY-REFERENCE)
                        (ANNOTE (TEXT Starting peg)))
             (GOAL-PEG (STRING BY-REFERENCE)
                        (ANNOTE (TEXT Goal peg)))
             (TEMP-PEG (STRING BY-REFERENCE)
                        (ANNOTE (TEXT Temporary peg)))
             (NUMDISKS (INTEGER BY-VALUE)
                        (ANNOTE (TEXT Number of disks to move))))
    (LOCALS)
    (RESULTS)
    (ANNOTE (TEXT Solve towers of hanoi problem))
    (RULES ((EQ NUMDISKS 1)
            (MOVE START-PEG GOAL-PEG))
           (T (TOWER-OF-HANOI START-PEG TEMP-PEG GOAL-PEG (MINUS1 NUMDISKS))
              (MOVE START-PEG GOAL-PEG)
              (TOWER-OF-HANOI TEMP-PEG GOAL-PEG START-PEG (MINUS1 NUMDISKS))))))
```

(a)

```
(OBSERVING-ROUTINE
  (FIND-BIGGEST
    (FORMALS (X (SET-OF-INTEGERS BY-VALUE)
                (ANNOTE (TEXT Set of elements))))
    (LOCALS (MAX INTEGER (ANNOTE (TEXT Maximal element))))
    (RESULTS MAX)
    (ANNOTE (TEXT Find the maximal integer in the set))
    (RULES (T (LOOP
                (INIT (ASSIGNQ MAX 0))
                (BODY (FOREACH I IN X DO (CHECKBIGGEST X MAX))))))))

(ACTING-ROUTINE
  (CHECKBIGGEST
    (FORMALS (ELEM (INTEGER BY-VALUE)
                   (ANNOTE (TEXT Next element to check)))
             (MAX (INTEGER BY-REFERENCE)
                  (ANNOTE (TEXT Maximum so far))))
    (LOCALS)
    (RESULTS)
    (ANNOTE (TEXT Compare maximum against next element))
    (RULES ((GT ELEM MAX)
            (ASSIGNQ MAX ELEM)))))
```

(b)

Figure 7: Good Uses of Recursion and Iteration

objects is also important to the question answerer, since it is needed to generate specific explanations in English. But it cannot be used effectively in the actual process of finding an explanation.

A data type in MIRROR consists of a domain (a set of values) and a set of operations. Types are defined similarly to the way clusters in CLU [Liskov 81] or packages in ADA [Ichbiah 80] are defined. All variables in MIRROR must be declared statically and their types must be specified. In addition to the usual information provided in a declaration, each MIRROR variable has associated with it an annotation, which provides the link between that variable and an object in the task domain. MIRROR is strongly typed, although coercion is allowed in a few special circumstances, as outlined below. All of these allowable coercions are *free coercions* [Geschke 77], since they involve no computation.

In addition to these general capabilities, which are currently implemented by building MIRROR on top of GLISP [Novak 82], MIRROR allows the following specific typing mechanisms.

4.2.1 Index Types

Index types signal to the explanation system that a particular variable will act as an array index. This knowledge is useful when a program is calculating with indexes, but the array that will be indexed is not directly apparent. Rather than searching for the actual access, the explanation system is alerted to it directly. To see the need for this, consider the problem of generating an explanation for the program shown in Figure 8. It is a QUICKSORT routine, taken from [Horowitz 76], whose only parameters are two integers. If m and n are declared as index types, then the explanation system will know that it is not really integers but rather pieces of an array that are being manipulated.

```
procedure QSORT(m,n)
   if m < n
       then [i <- m; j <- n + 1; key <- k[m]
           loop
               repeat i <- i + 1 until k[i] ≥ key;
               repeat j <- j - 1 until k[j] ≤ key;
               if i < j
                   then call INTERCHANGE(r[i],r[j])
                   else exit
           forever
           call INTERCHANGE(r[m],r[j])
           call QSORT(m,j - 1)
           call QSORT(j + 1, n)]
end QSORT
```

Figure 8: A Hard-to-follow Program

An index type is defined in terms of an underlying type and a set of array types. The underlying type must be an enumerated type, such as integer. All operations on the underlying type are automatically overloaded for the index type. Associating with an index type a set of array types (those for which the index type may be used) provides useful information without restricting the use of the type. Notice that it would help an explanation system even more if it knew exactly which array each index were to be used with. But in order to do so, it would be necessary to prohibit such constructs as the following, in which the index I is used to reference one array to choose a position and then to access a second array to do something in the chosen position:

```
I := 1;
WHILE I ≤ N ∧ A[I] ≠ X DO
   I := I + 1;
IF I > N THEN ERROR
         ELSE B[I] := Y;
```

The goal of index types is to make it obvious when a computation is being done whose purpose is the manipulation of an array component. Of course, it is also possible that an essentially numeric computation is being performed but the result must also be used to access an array element. This can be done by computing with an integer type and then doing an explicit type conversion to generate a value of an index type. This conversion corresponds to the idea that the positive integers can be viewed as ordinals and that an ordinal can be viewed as a function that can be applied to any linear structure to extract an individual element. Thus there can also exist conversion functions from the integers to pointer (or access) types applied to linear lists.

4.2.2 Built-In Structured Types

Arrays are simple, both to implement and to describe. Other common structured types, such as lists, are more complex. The design of the structured type mechanisms in MIRROR is motivated by the following two observations:

1. Addresses (e.g. pointers) are to data structures what gotos are to control structures. They are completely general structures that allow arbitrary interconnections among program objects. But because of this, programs that use them are difficult to understand.

2. There is a fundamental distinction between the use of an address to specify the next piece of a structure that happens not to be stored contiguously (this is called a *link* in MIRROR) and the use of an address to point to (select) a piece of a structured object (this is called a *pointer* in MIRROR). Arrays do not need links but they do need something corresponding to pointers (which are provided by index types). Linked lists need both.

Because of the difficulties posed by gotos, most current programming languages provide a set of higher-level control structures, such as loops, that make the need for gotos rare and confined only to those circumstances in which the required control structure cannot be expressed using the higher-level structures that are provided. MIRROR does this with control structures and it also does it with data types. It provides, as built-in types, the common structures list and set. Others maybe added if they are required. Unfortunately, though, we must assume that no matter how rich the built-in collection is, it will not always be adequate. In fact, this appears to occur more often with respect to types than it does with respect to control. When it does occur, programmers must manipulate addresses explicitly. Programs that do this are more difficult to explain than those that do not, but this is to be expected, for two reasons:

- They are usually more complicated.
- They are using structures that the explainer, who does know something about the built-in types, does not already know about.

MIRROR provides two address types, links and pointers. Neither may be used in an unrestricted form: instead they must be tied to the specific type to which they may refer. Figure 9 shows the use of links and pointers to define a type binary tree. Two data objects are to be stored at each node of this tree: an integer and a *pointer* to an object in an externally defined data base. *Links* are used to describe the way that the structure itself is constructed. The practical significance of this distinction between pointers and links is described in the next section. It can be summarized though by saying that the tree is all the objects that can be reached by starting at the link that defines it and following all accessible links. Objects that can be reached by pointers, on the other hand, are not part of the tree itself, although the tree contains references to them.

4.2.3 The Assignment Operator

In MIRROR, assignment is done with the ASSIGNQ operator. The statement

(ASSIGNQ A B)

always means copy B into A. This differentiates MIRROR from many languages.

When an assignment statement in any language is executed, something is copied from one place to another. Sometimes what is copied is the value of interest, and sometimes what is copied is a pointer to such a value. In languages, such as PASCAL, in which user programs manipulate pointers explicitly, it is generally clear which is happening. Data types that are implemented using structures with pointers to

```
(TYPE
  (BINTREE (LINK TO TREENODE)) (ANNOTE (TEXT Binary tree)))

(TYPE
  (TREENODE (RECORD
              (VALUE1 INTEGER (ANNOTE (TEXT First value)))
              (VALUE2 DB-POINTER (ANNOTE (TEXT Second value)))
              (LCHILD BINTREE (ANNOTE (TEXT Left child)))
              (RCHILD BINTREE (ANNOTE (TEXT Right child)))))
  (ANNOTE (TEXT Node in a binary tree)))

(TYPE
  (DB-POINTER (POINTER TO DB))
              (ANNOTE (TEXT Pointer to a database record)))
```

Figure 9: MIRROR Definition of the Type Binary Tree

them (such, as for example, a binary tree implemented as a linked structure) are not copied, while all other data types (e.g. integers) are. In languages, such as LISP, in which user programs do not explicitly manipulate pointers, it is much less clear whether objects or pointers are copied on assignment. In LISP, for example,

```
(SETQ A B)
```

results in copying if B has a number as its value but it does not if B has as its value a list.

Whether or not copying is done is crucial to an understanding of a program's behavior. This is particularly true since failure to copy creates aliasing, which must be considered when the question answerer tries to match the objects with which it is concerned against specific named objects in the code. The goal, in the design of MIRROR, is to make the copying/not copying distinction clear in two respects:

- It should be clear which is occurring from a static analysis of a program. This can be done easily since the types of all variables must be declared.
- The distinction between copying and not copying should be made on semantic rather than implementation grounds. Whether the assignment of one binary tree to another should involve copying should not be decided by whether the tree happens to be implemented as a linked structure or an array.

If A and B are declared to be of type binary tree (as defined in the last section), then the ASSIGNQ statement shown above will cause the tree structure to be copied using the following algorithm:

- To copy an object of any type other than link (e.g., integer, pointer), simply copy the value directly
- To copy an object of type link, recursively copy the object that is linked to, then, in place of the original link, insert a link to the newly copied object.

This means that all the tree structure, which is tied together with links, will be copied. But the data values stored in the external database will not be copied. Just the references to them (the pointers) will be copied.

If P and Q are declared to be of type pointer to bintree, then the statement

```
(ASSIGNQ P Q)
```

will copy the pointer contained in Q into P. But no tree structure will be copied. Thus we have made the key distinction between objects that are trees (even though they happen to be implemented with addresses) and objects that are indexes into trees (which are also implemented with addresses).

4.2.4 Augmented Types

Many functions that normally return values of one type may, on occasion, return special values that are not of that type. For example, a READ function that is expected to return an integer may also return the special value EOF. To support this, MIRROR allows the definition of *augmented types*. An augmented type's domain is equal to the domain of another, already defined type (called the base type) plus one or more special values. All of the operations defined for the base type are defined for all elements of the augmented type that are also present in the base type. The program must check to make sure that these operations are not called with any of the special values. Coercion from the augmented type to the base type is allowed subject to the same restriction.

Figure 10 shows an example of the use of an augmented type to allow a function to test for end-of-file. Note that although the type of IN is the augmented type FILE-INTEGER, the type of ONE is simply INTEGER. This is allowed since the code of PROCESS will prevent PROCESSONE from being called when IN takes on the special value EOF.

```
(TYPE
  (FILE-INTEGER (INTEGER ∧ EOF))
                (ANNOTE (Text Range of values from input file)))

(ACTING-ROUTINE
   (PROCESS
     (FORMALS)
     (LOCALS (IN FILE-INTEGER (ANNOTE (TEXT Next input from file))))
     (RESULTS)
     (ANNOTE (TEXT Read integers from file until done))
     (RULES (T LOOP
                  (INIT (ASSIGNQ IN (READ)))
                  (BODY (WHILE (NEQ IN EOF) DO
                          (PROCESSONE IN)
                          (ASSIGNQ IN (READ)))))))))


(ACTING-ROUTINE
    (PROCESSONE
     (FORMALS (ONE (INTEGER BY-VALUE)
                   (ANNOTE (TEXT Value to be processed))))
     (LOCALS)
     (RESULTS)
     (ANNOTE (TEXT Process one integer from file))
     (RULES (T ...))))
```

Figure 10: Using an Augmented Type

The use of "logical" augmented types is common in untyped languages such as LISP (in which the most common special value is NIL). By allowing the declaration of augmented types, MIRROR permits the flexibility that these types allow without sacrificing the static availability of type information.

4.2.5 Union Types

In many problem domains, the assignment of values to type classes is not best done using only a single level. Instead, it is sometimes logical (and thus easier to explain) to define a type as the union of two or more other types. Whenever a union type appears, it must be part of a structure that contains at least one other component that disambiguates the union type. Thus union types are similar to variant records in Pascal [Wirth 74].

An example of the use of a union type in the Scribe domain is the following. A user of Scribe may define an environment (such as letter body or enumerated list) by giving an environment name and a list of attribute value pairs that define what will happen inside the environment. Scribe needs an internal structure in which to represent this definition. The MIRROR declaration for this structure is shown in Figure 11 (*a*). It says that an environment-definition consists of a list of two elements. The first, called name, is of type string. The second, called attrs, is a list of objects, each of which is of type attribute-value-pair.

```
(TYPE
  (ENVIRONMENT-DEFINITION
    (RECORD
        (NAME STRING (ANNOTE (TEXT Environment name)))
        (ATTRS (LISTOF ATTRIBUTE-VALUE-PAIR)
               (ANNOTE (TEXT Modifications))))
    (ANNOTE (TEXT An environment definition))))
```

(a)

```
(TYPE
  (ATTRIBUTES
     (ENUMERATED LEFT-MARGIN RIGHT-MARGIN JUSTIFICATION FLUSHLEFT...)
     (ANNOTE (TEXT All attribute names))))

(TYPE
  (VALUES
    (UNION HORIZONTAL-DISTANCE VERTICAL-DISTANCE INTEGER BOOLEAN NIL)
    (ANNOTE (TEXT Possible value types))))

(TYPE
  (ATTRIBUTE-VALUE-PAIR
    (RECORD
        (ATTR ATTRIBUTES (ANNOTE (TEXT The attribute name)))
        (VALUE VALUES (ANNOTE (TEXT It's value))))
    (ANNOTE (TEXT An attribute value pair))))
```

(b)

Figure 11: The Use of a Union Type

But now a definition of the type attribute-value-pair is required. An attribute-value-pair consists of an attribute and a value. The attributes are all strings and must come from a fixed set prescribed by Scribe. But the values must depend on the attribute. Attributes such as LEFT-MARGIN have a value of type horizontal-distance, while JUSTIFICATION requires a boolean value and FLUSHLEFT takes no argument at all. To represent this, we must create a union type that includes all of these base types. Figure 11 (*b*) shows the definition of the type attribute-value-pair using this union type.

The only operation that is defined explicitly on a union type is assignment. All other operations must be

done by viewing the value as an element of the appropriate base type (such as horizontal-distance) and performing operations defined on that type. Coercion from the union type to the appropriate base type is allowed.

4.2.6 Size Definitions

Every structured object (such as a set) has, at each instant of run-time, a size. Sometimes this size, or at least a bound on it, is known prior to run-time. For example, there are always exactly 80 columns on a punch card, so the size of a structure representing such a card will be known at the time the program using it is written. Sometimes, though, the size of a structure cannot logically be known prior to run time. Continuing with the punch card example, for instance, consider the size of a list of cards, each representing one employee, to be read in and processed. The size of this structure is not known until the cards are read.

MIRROR allows sizes of structured objects to be specified but they need not be. If the size is known, it can be used as part of an explanation. But this is true only if the size is real and has not simply been included because an early binding of structure size is required by the language. When such a spurious binding is provided, it interferes with the operation of the question answerer in two ways. It makes what is really only a forced guess appear to have the status of a fact. And it forces the programmer to write housekeeping code to handle the situations in which the guess turns out to have been inaccurate. This extra housekeeping code clutters the program and so makes explanation of the program more difficult.

The ability to choose between a size specification provided at program-writing time and one that is left open until run time is an example of the crucial MIRROR idea of "logical" binding time.

4.2.7 Aliasing and Global Variables

Any procedure for answering questions from code must depend heavily on the ability to match code segments against specified effects. To do this, it is necessary to find the places in the code where a particular object is set and used. To do this efficiently, it is necessary to be able to reason from a static analysis of the code, with a minimum of simulation of the program's dynamic behavior. To do this, it is necessary to minimize aliasing.

One way to do this is to use global variables for structures that are truly global and to restrict, as much as possible, the use of global variables as parameters. For example, the Scribe system makes extensive use of a system state vector, several input files, and several text buffers. Almost all parts of the system refer to these structures. By making these structures global and referring to them throughout the program consistently by the same name, references to them can easily be discovered by the matcher. The use of global variables has been the source of some controversy [Wulf 73, Peterson 73, Liskov 81]. But most of that controversy has centered around the use of global variables for other reasons than because structures are genuinely global, for example, to simulate the ALGOL60 **own** capability. We are not suggesting that they be used for that.

Once the use of any nonlocal variables is admitted, the question of the number of levels of nonlocal reference to be allowed immediately arises. For many systems, of which Scribe is one, the necessary number of levels is one. There are global structures and there are local ones. But logically this is not necessary. For example, a system could consist of several phases, each with its own set of global structures. Or one large phase might consist of several subparts each of which uses its own global structures as well as structures global to the phase. So multiple levels may be necessary. But many fewer levels of nonlocal referencing are required than there are levels of procedural decomposition. Thus, in MIRROR there is a clear-cut distinction between blocks, which contain type definitions, global variable

definitions, and procedure definitions, as opposed to procedures, which contain only local variables and code. This leads to the definition of a MIRROR program shown in Figure 12.

<program> ::= <block>

<block> ::= <type definitions> <global variables> <body>

<body> ::= <block sequence> | <procedure sequence>

<block sequence> ::= <block> | <block> <block sequence>

<procedure sequence> ::= <procedure> | <procedure> <procedure sequence>

Figure 12: The Form of a MIRROR Program

4.2.8 Parameters

Aliasing cannot be avoided entirely by the use of global variables. It arises whenever a new pointer is created to an object. There are two circumstances in which this may occur:

1. Passing of a parameter by reference.
2. Execution of an ASSIGNQ statement of an existing pointer value to another pointer variable.[2]

Although the use of global variables cuts down on the need for parameters, parameters are necessary for procedures that do in fact operate on more than one program object. This occurs often for local variables. It occurs occasionally even for large global structures to which a standard utility routine such as sort must be applied. Thus the matching component of the question answerer must handle aliasing by tracing calling sequences to find places where the object in which it is interested is used. In MIRROR, parameters are typically passed by value, and so no aliasing arises. Parameters may be passed by reference, but they must be marked as such in the header of the called procedure. There are two reasons why a programmer might choose to pass a parameter by reference

- To avoid the overhead of copying, even though no nonlocal effect on the original object is intended.
- To make it possible to change the value of the object within the called procedure.

Since the MIRROR representation of a program is not intended to be the executable representation of the program, the first of these will not occur. Reference parameters will be used only when they are required for semantic reasons. When this happens, the question answerer must handle the multiple names that an object may have.

The fact that aliasing may also arise from the use of pointers and must also be handled properly by the question answerer has already been discussed.

---

[2] In MIRROR, ASSIGNQ is used for all assignments, including those done in other languages using other assignment statements (such as LISP RPLACA and RPLACD).

4.3 Summary of Procedures, Headers, Parameters, and Results

Procedures are central to the structure of MIRROR because they allow a program to be divided into pieces in such a way that only a small part of the program must be examined during the process of answering specific questions. Several things have already been mentioned about the way procedures are declared and used in MIRROR. This is because those things have related to many of the other issues in the language design. In this section, a complete description of MIRROR procedures, parameters, and results will be presented, even though some of the descriptions will duplicate statements that have already been made.

Each MIRROR procedure begins with a header that contains summary information about the objects with which the procedure deals. The header contains four mandatory fields. The four mandatory fields are: FORMALS, LOCALS, RESULTS, and ANNOTE. Each field contains a list of variables and their associated types (except RESULTS, all of whose variables must be mentioned in one of the other fields and whose type information can be found there; and ANNOTE, which contains something else). The purpose of the header information is to constrain the question-answerer's search to just those procedures that might act on the objects involved in the question to be answered.

There are two kinds of procedures in MIRROR, observing procedures and acting procedures. The former cause no side effects; they only compute values to be returned. The later may both cause side effects and create values. This distinction is marked explicitly in the procedure heading.

Procedures may return one or more values, representing objects created or discovered by the procedure. Some procedures do not need to return any values; they do all of their work by modifying existing objects, which may either be global variables or parameters to the procedure.

The FORMALS field of the procedure heading lists all of the formal parameters used in the procedure. Each parameter must be marked explicitly as being passed either by value or by reference.

The LOCALS field of the procedure heading lists the procedure's local variables. The scope of these variables is just the procedure in which they are declared. They can, however, be used in called procedures if they are passed as parameters.

The RESULTS field of the procedure heading lists the formal and local variables whose values are to be transmitted out of the procedure. The order in which multiple variables appear in the RESULTS field defines the order in which the values will be returned.

The ANNOTE field of the procedure heading provides the link between the procedure and the terms that users may employ to describe the operation that is performed by the procedure. These annotations are used both to find the appropriate procedure given a question and to generate appropriate English responses given a procedure. As described in Section 4.2, similar annotations are also associated with variables in MIRROR.

5 Implications for Source Languages and Programs

We know that the choice of a particular internal representation does not completely determine the design of the language in which people write programs. But it is recognized that some programming language features facilitate the transformation into certain kinds of internal representations while other features make the same transformation difficult. Thus we see languages such as FORTRAN and C, from which it is easy to generate good executable code and languages like ALPHARD [Wulf 76] and GYPSY [Ambler 76], from which it is relatively easy to generate good correctness proofs.

Some language structures are particularly important if programs written in them are to be easily translatable into MIRROR. These include:

- User-defined functions and procedures.
- Static type declarations.
- A rich collection of data types to build from and a good way of building from them.
- Global variables.
- Static scope of names.
- Parameters transmitted by reference and marked as such.

Of course, it is not sufficient for a program to be written in a language that provides the necessary structures; the program must use them properly. The following properties of a program facilitate that program's translation into MIRROR:

- Modularity.
- Explicit use of defined types for semantically significant classes of objects.
- Separation of observing procedures from acting procedures.
- Mutually exclusive guards in case statements or CONDs.

Even with the right language structures and an ideal program design, the transformation of a traditional program into MIRROR will not normally be straightforward because of the additional information (particularly the annotations) that are required. Two solutions to this problem are possible:

- Translate into MIRROR interactively and ask for additional information as it is needed. There is currently an effort underway to do this with programs written in INTERLISP.
- Provide a good programming environment that supports the construction of programs that are augmented with the additional information required for a MIRROR representation. This is certainly the preferred approach for new programs.

6 Conclusion

In this paper we have described a program-representation language that facilitates the automatic generation of explanations of program behavior. Although this language is not necessarily the language in which people initially write programs, its structure does impose constraints on the way people do write their programs and on the language in which they write them. In particular, the relationship between program structures and objects in the problem domain must be made clear by the programmer.

If the requirements imposed on programming by the need to translate into MIRROR were orthogonal to those imposed on programming by the other uses to which programs are put (as suggested in Section 2), we would be concerned and would be forced to question the usefulness of the MIRROR approach. But, in fact, they are not. In particular, the requirements of MIRROR are very similar to the requirements of both human maintenance and automatic verification. (Compare, for example, the programming principles outlined in [Gries 79].) This is not surprising. Many of the ideas discussed in this paper (such as logical binding time, augmented types, and the use of the most natural control structure) apply to those other aspects of program development as well.

There are, however, some differences between MIRROR and languages that were designed primarily to

support formal verification. The most important of these is the admission of the address types pointer and link. There is no question that the inclusion of these types in the language substantially complicates the language's semantics. But they have been intentionally included in MIRROR. The reason is the following. For any program, there is an abstraction continuum, with the code at one end and the original idea of what the program should do at the other. Somewhere in between, there should be an explicit statement (the specifications) that describe what the program should do. Verification systems must show the correspondence between the code and these explicit specifications. The designer of a verification system and a language to support that system is allowed to choose both of those points. So it makes sense to design a language that does not contain features that are difficult to handle; those same features can be omitted from the specification language. If those difficult features occur naturally in the original idea for the program, they can be ignored since the verification system does not concern itself with reasoning at that level or with mapping from that level to the specifications. In designing a program to reason in English, to users, about another program that operates in a particular problem domain, one does not have so much freedom. Now the system must map all the way from the code to the naturally occuring concepts in that problem domain. If the messy things are missing in the program, they will have to be reconstructed, which is more difficult even than dealing with them if they appear in the code.

As an example, consider the implementation of a binary tree. If pointers are not provided, a binary tree can be implemented as an array. If the formal specifications for the tree are written in terms that map easily to arrays, then verifying the correctness of a program that manipulates the tree will not be difficult. But explaining that program in terms of a tree will be hard. For question answering, on the other hand, it is not possible to hide the difficult transformations in the idea-to-specification step and then to simplify the specification-to-code correspondence. The whole span, from idea to code, must be accounted for. Thus MIRROR includes such things as address types, even though they complicate the semantics of the language, because they simplify the relationship between the code and the ideas behind the code.

7 Acknowledgements

We wish to thank Kim Korner, John Hartman, and Warren Hunt for their contributions to the design of MIRROR.

References

[Ambler 76] Ambler, A. L., D. I. Good, J. C. Browne, W. F. Burger, R. M. Cohen, C. G. Hoch & R. E. Wells.
Gypsy: A Language for Specification and Implementation of Verifiable Programs.
In *Proc. ACM Conference on Language Design for Reliable Software*. 1976.

[Dijkstra 78] Dijkstra, E. W.
Guarded Commands, Nondeterminacy, and Formal Derivation of Programs.
In *Programming Methodology*, Springer-Verlag, New York, 1978.

[Genesereth 78] Genesereth, Michael.
Automated Consultation for Complex Computer Systems.
PhD thesis, Harvard, 1978.

[Geschke 77] Geschke, C. M., J. H. Morris, Jr., & E. H. Satterthwaite.
Early Experience with Mesa.
CACM 20(8), Aug., 1977.

[Gries 79] Gries, D.
Current Ideas in Programming Methodology.
In P. Wegner (editor), *Research Directions in Software Technology*, MIT Press, Cambridge, Mass., 1979.

[Henderson 80] Henderson, P.
Functional Programming: Application and Implementation.
Prentice-Hall, Englewood Cliffs, N.J., 1980.

[Horowitz 76] Horowitz, E. & S. Sahni.
Fundamentals of Data Structures.
Computer Science Press, Potomac, Md., 1976.

[Ichbiah 80] Ichbiah, J. et al.
Reference Manual for the Ada Programming Language.
U.S. Dept. of Defense, 1980.

[Kehler 80] Kehler, T. P. & M. Barnes.
Alternatives for On-line Help Systems.
In *Proc. 8th ACM SIGUCC User Services Conference*. 1980.

[Liskov 81] Liskov, B. et al.
CLU Reference Manual, Lecture Notes in Computer Science.
Springer-Verlag, 1981.

[Novak 82] Novak, G.
GLISP: A High-Level Language for A.I. Programming.
In *Proc. AAAI-82*. 1982.

[Parnas 72] Parnas, D. L.
On the Criteria to be Used in Decomposing Systems into Modules.
Communications of the ACM 15, December, 1972.

[Peterson 73] Peterson, J. L.
Letter to the Editor.
SIGPLAN Notices 8(5), May, 1973.

[Reid 80] Reid, Brian.
Scribe: A Document Specification Language and its Compiler.
PhD thesis, Carnegie-Mellon, 1980.

[Rich 82] Rich, E. A.
Programs as Data for Their Help Systems.
In *Proc. National Computer Conference*, pages 481-485. 1982.

[Rich 83] Rich, E. A.
Artificial Intelligence.
McGraw-Hill, New York, 1983.

[Shapiro 75] Shapiro, S. C. & S. C. Kwasny.
Interactive Consulting via Natural Language.
Comm. of the ACM 18(8), August, 1975.

[Weiser 81] Weiser, M.
Program Slicing.
In *Proc. 5th International Conference on Software Engineering.* 1981.

[Wilensky 82] Wilensky, R.
Talking to UNIX in English: An Overview of UC.
In *Proc. AAAI 2.* 1982.

[Wirth 74] Wirth, N. & K. Jensen.
PASCAL User Manual and Report.
Springer-Verlag, New York, 1974.

[Wulf 73] Wulf, W. A. & M. Shaw.
Global Variable Considered Harmful.
SIGPLAN Notices , February, 1973.

[Wulf 76] Wulf, W. A., R. L. London, & M. Shaw.
An Introduction to the Construction and Verification of Alphard Programs.
IEEE Trans. on Software Engineering 2(4), 1976.