October 1970

QA4 WORKING PAPER

by

Johns F. Rulifson
Richard J. Waldinger
Jan A. Derksen

Artificial Intelligence Group
Technical Note 42
SRI Projects 8721, 8550, 8259

The research reported herein was sponsored by the Air Force Cambridge Research Laboratories under Contract F19628-70-C-0246, by the National Aeronautics and Space Administration under Contract NASW-2086, and by the Advanced Research Projects Agency and the National Aeronautics and Space Administration under Contract NAS12-2221.
I GENERAL GOALS OF THE LANGUAGE

A. The Language and Its Data Base

The QA4 language is an enhanced omega-order language\(^*\) embedded in a system of control statements. The declarative facets of the language include atomic symbols, tuples, unordered tuples, sets, function definitions, and applications; the imperative facets include (in addition to normal program control features) set iteration, backtracking, and parallelism. The language is intended to be a natural formalism for the description of problem-domain-oriented theorem-proving strategies. Moreover, the specification of problems to be solved by QA4 programs have a natural, compact formulation in the same language. That is, the statement of theorems to be proved or the specification of programs to be written is a task similar in nature to writing theorem provers or program synthesizers. For this reason, the data base for QA4 programs is QA4 expressions. A preliminary description of the QA4 syntax appears in Ref 2.

B. Properties of Expressions

In addition to the syntactic component that uniquely distinguishes it from all other QA4 expressions, every QA4 expression has a property list. This list stores arbitrary properties and their values,

\(^*\) References are listed at the end of this note.
the values being, in turn, QA4 expressions. The properties are used by QA4 programs both to store information for the interpreter, and to guide strategies and communicate information about the data on which the programs are working. These properties fall into three categories: interpreter bookkeeping, semantic, and pragmatic.

The standard semantic properties of an expression include its value, the set of expressions it is known to equal, the sets of expressions it may not equal. Rules for evaluation and simplification are also semantic properties. It is assumed that partial evaluation or simplification of expressions will be an important strategy in all QA4 problem solvers. The QA4 interpreter comes equipped with such a partial evaluator. It is, however, incomplete, but can be enhanced through the use of appropriate semantic properties. Finally, it is often useful to write a strategy in terms of a particular data structure, say a set. The programs may be clear and concise, making the strategy transparent and flexible. Yet, for reasons of efficiency it may be necessary to represent the set outside the standard QA4 framework, say with a LISP array. Such representation information is handled by the use of semantic properties.

Pragmatic properties are peculiar to each individual problem. The properties are used by strategy programs to communicate and note information about expressions. They take the flavor of statements such as "I've tried this before and it didn't work."
C. **Expression Manipulations**

Expression manipulation is accomplished by decomposition and construction. Decomposition, in QA4, means naming parts or components of an expression. The naming is done with pattern matching. Patterns may occur at many points in the language: in functional variable bindings, assignment statements, and conditional tests. Transformation of expressions is done through a complete set of constructors: add an element to a set, add onto tuples, or construct a lambda expression, to name a few. There is also a large set of primitive operators on the structural data forms, e.g., set union, arithmetic addition, and Boolean conjunction.

D. **Control Statements**

In order to solve large problems and carry out long proofs, it is necessary to have highly goal-directed search strategies. Moreover, many of the searches done in QA4 strategy programs simply do not have appropriate numerical means of guiding them. That is, the semantic-pragmatic search techniques are guided by programs making local decisions on current information. Any attempt to centralize the search or have uniform procedures cannot be done easily. For this reason, the QA4 language makes directly available, through statements in the language, many well-known search procedures. This means that each particular problem-domain-oriented strategy program can use appropriate search techniques at its own local level. Strategies may thus search in parallel,
grow search trees, or backtrack whenever such methods are appropriate. Accordingly, one can no longer characterize a QA4 program as doing a particular kind of search while it is problem solving; in most cases, many (if not all) kinds of search are being done.

The search-oriented statements of QA4 fall into three categories:

*Iteration over sets*—taking the form of selection through patterns and for each statements.

*Parallelism*—Appearing as coroutines, parallel strategy execution, and when statements.

*Backtracking*—Taking place in the program failure mechanism and the choice function (choices many times being made from possible matches to a pattern).

II ORGANIZATION OF THE INTERPRETER

A. User Interface

The QA4 programmer views the system as an interactive programming tool. He types commands in the form of QA4 expressions to a top-level function. These commands may input or modify expressions or values of properties of expressions; define, modify, or execute programs; or perform debugging tasks. Roughly speaking, the system is divided into three parts: input/output, editor, and interpreter.

The input/output system is an expression parser, which transforms QA4 infix syntax into prepolish or internal format. The parser uses the BIP package and has the advantage of being readily modified.
Similarly, an output function takes the internal expression form and outputs a corresponding infix output stream. Thus, the user always communicates with QA4 in an infix mathematical-style notation.

The editor is still conceptual. While we feel it is an essential part of a useful human-oriented system, it is yet to be specified.

The QA4 interpreter is an EVAL function resembling LISP EVAL. It accepts QA4 expressions and, with the aid of an extensive library of primitive functions, executes them. At this time we have no plans to make interpretations of expressions that do not have an immediate, obvious value (say, FORALL statements). We hope that experience with theorem-proving programs will show ways of automatically extending the basic EVAL.

B. **Expression Storage**

The storage and retrieval of expressions is fundamental to the QA4 system. That is, given a syntactic form for an expression, a fundamental operation is to look the form up and find the properties already assigned or known about the form. This is an extension of LISP's atom property feature to expressions in general. Internally, a QA4 expression is a property list consisting of a property EXPV, whose value contains the syntactic information about the expression, and whose remaining properties are semantic or pragmatic. When an expression is stored, a lookup is made to determine whether or not the expression has been stored
before. If so, the old expression is returned, and if not, a new expression is added to the general store. Thus, only one copy of each expression is retained by the system.

The storage mechanism is a discrimination net. To understand the workings of the net, suppose the system contained only the expressions, in internal format,

\[(\text{SET } A \text{ B}), (\text{TUPLE } A \text{ B}), (\text{TUPLE } C \text{ B})\].

The net automatically created for storing these expressions might be

```
  STYLE
   /\      \\
  SET \    /\  TUPLE  \\
     /\      /\      \\
    (SET A B) (A) (C) \\
          /\     \\
         /     \\
        (TUPLE A B)  (TUPLE C B)
```

The net is a tree. Each node of the tree contains

1. A function, which extracts an atomic piece of syntactic information, and

2. Either a terminal node or a list of branches. (A terminal node contains an expression, and a branch is a pair—an atom and another node).

A syntactic form is looked up in the net by applying the feature extraction at the top node, choosing the appropriate branch, and continuing until a terminal node is reached or there is no appropriate
branch. If no branch exists, then the expression does not occur in the net and a new terminal node may be added.

When a terminal node is reached, the input expression must be checked against the syntactic property on the expression at the terminal node. If they match, all is well and the property list for the form has been found. If they do not match, a new branching node must be created. To construct the feature selector the two expressions are compared in a structural depth-first manner until the first difference is noted. The results of this search are encoded into a list and installed as the feature selector of the new node. A terminal node for the new expression is constructed, the two new branches made up, and the net is transformed to hold the property list for the new form.

If two QA4 expressions are identical except for the names of their bound variables, they go into the same internal representation. Thus, bound variables may not be used as selector functions. Moreover, in order to store sets and bags in the net, an index is assigned to each element of a set or bag expression the first time it is stored. If the same set is then stored a second time (perhaps with some expressions permuted), the elements are first sorted by the index numbers and then discriminated upon syntactically. Thus, if a user types in the set \{A,B,C\}, the elements are assigned indices A ← 1, B ← 2, C ← 3. If the set \{C,B,A\} is entered, it is sorted into \{A,B,C\} and then found to
already occur. The net functions also maintain statistics concerning the number of references made to each expression and discrimination for future optimizations.

C. **Equality Partitions**

The efficient treatment of the equality predicate is crucial to the operation of any problem-solving system. Rather than axiomatize the equality rules, we have built them into the QA4 system by introducing equality partitions. Each expression in a context has (as its value property for that context name) the set of expressions known to be logically equal to it in that context. When two expressions are asserted or proved equal in a context, their "equality sets" are merged to form a new set for each. Moreover, each expression has (in context) a set of sets of expressions that are known to be unequal to the given expression. That is, each set in the "unequal set" contains a set of expressions known to be not all equal. Again, when a new equality assertion is made, these sets are updated correspondingly. Consequently, whenever an equality assertion causes a contradiction via the equality rules, it is immediately known. An additional advantage to maintaining the equality information is to be able to select the "best" expression equal to a given expression for a certain purpose.
III CONTEXTS

A. Intent and Uses

Variable bindings are implemented in the QA4 interpreter with a "context" mechanism. This method of storing all the changeable property values of expressions simplifies the execution of parallelism and backtracking in the interpreter. The same facilities, moreover, are made available to the users as a method of data manipulation in programs dealing with the frame-problem, conditional proofs, or variable bindings. The mechanism simulates a branching pushdown stack. Each node in the tree corresponds to a process or state of the world. When a process changes properties of an expression, the changes are only effective for the process and its descendants. The property values of the ancestors of the process are unchanged.

B. Example

1. Coroutines

For example, suppose a process P is being interpreted, and it creates two coroutine subprocesses P1 and P2. With each creation, the interpreter creates a new context, and each is an extension of P. We might represent this as:

```
    P
   /\  \\
  /   \  \\
P1    P2
```

2. **Backtracking**

Backtracking is slightly different. If P is terminal (that is, it has no subprocesses when a backtracking point is reached), then a new context is created; however, the new context is an extension of P. This is done so that further changes in variable values in P will not destroy the old values, and the state at the backtracking point can be readily restored:

$$P \Rightarrow \{ \}$$

If P already had subprocesses, then the new context is an extension of P, which interposes itself between the original P and the subprocesses:

$$P \Rightarrow \{ \}$$

C. **A Note of Caution**

When the interpreter and programs use the same data base, care must be taken by user programs during property list manipulation. These concerns come naturally to a LISP programmer who confronts the same problem when he uses properties of atoms. The usefulness of the feature, however, certainly makes it worthwhile. The problems of the interpreter and user programs are very similar, and mechanisms useful
for one are probably useful for the other. It is important, therefore, that QA4 programmers fully understand the context mechanism and exploit it in their programs to gain the full power of the language.

D. Implementation

A data item of type context is a list of numbers, say (5, 3, 1). Each number corresponds to a node in the graph representation of the process structure. For example, suppose the current process structure was

```
      P1
     /  \
   P2   P3
    /    /  \
   P4   P5   P5
```

then (5, 3, 1), (4, 3, 1), (2, 1), (3, 1) are all possible contexts. Process P3's context is (3, 1), while P4's context is (4, 3, 1). The extension of a context is handled by the function XCTX, which creates a new unique context number and puts it on the front of a context.

The values of properties of expressions are stored as property lists themselves, where the context numbers are property names. For example, an expression might look like:

```
(NET EXPV (TUPLE 1 2) P1 (CONTEXTLIST 5 Q 3 R)).
```

This internal representation means that the value of property P1 for the tuple (1, 2) was set to Q under a context headed by 5, say (5, 3, 1) and set to R under a context headed by 3, say (3, 1). In the sample above, P3 may have set the value to R, while P5 set it to Q.
E. **Lookup**

The lookup routine CTXGET takes an expression, a property name, and a context as arguments. If e were a pointer to the above expression, then (CTXGET e "Pl" "(3 1)"), would first get the LISP values of property \textit{Pl}, the list \text{(CONTEXTLIST 5 Q 3 R)}. It would then look for a value under context number 3, and if that fails under 1. In our example, it finds one under 3 and returns R.

F. **Changing Contexts**

Contexts are popped by the function POPCTX, properties are added with CTXPUT, and removed with CTXREM. The context functions note all current contexts and discard all else during garbage collection.

G. **Summary**

The whole notion of the discrimination net as a means of accessing expressions is a method of extending the LISP idea of property list from atoms to expressions in general. The inclusion of bound variable expressions and sets in the net causes some concern, but can be handled. The context mechanism is an extension in a similar vein. The values of properties can be with respect to a given state or binding level. LISP programs sometimes do this when the value of a property is treated as a pushdown stack. However, a simple stack is not enough for parallelism and backtracking. The context mechanism appears to be a concise, natural
method of extending the basic notions. It even carries along the
features of garbage collection, something which change lists and other
approaches have difficulty with.

H. Example

The QA4 theorem prover uses high-level rules of inference.
Thus, one QA4 proof step may represent many formal steps. QA4 rules
of inference may be very special-purpose: In any situation, we expect
the system to select, from a large collection, those rules that might be
advantageously applied.

We see the QA4 theorem prover working at the same level as a
human mathematician, and a finished QA4 proof should read like a proof
in a mathematical textbook. To illustrate this point we present a
fairly difficult theorem, and a protocol of the projected QA4 proof
procedure applied to this theorem. The following discussion presents
only the "correct" branch of the hypothetical QA4 solution. A problem
solving strategy that would generate this solution, among others, is
described in the next section of this note.

The theorem to be proved arises in a program-synthesis problem.
We are given a recursive program to compute the Fibonacci sequence
1,1,2,3,5,8, ... in which each term is the sum of the preceding two terms.
The program we are given is

\[ \text{fib}(x) = \begin{cases} 
1 & \text{if } x \leq 1 \\
\text{fib}(x - 1) + \text{fib}(x - 2) & \text{else} 
\end{cases} \]
This program is grossly inefficient, requiring many redundant recursive computations of the function on the same argument. We would like to construct an equivalent iterative program.

Of the many possible QA4 rules of inference, the following are useful in this problem.

(1) Induction (Going-Up Iterative⁴): To prove a theorem of the form \((\forall x)P(x)\), where \(x\) is a natural number, prove \(P(0)\) and prove \((\forall x)P(x) \Rightarrow P(x + 1)\).

(2) Resolution: The equivalent of Robinson's rule,⁵ but expressed in terms of QA4 expressions with quantifiers.

(3) Partial Evaluation: Take a function that is defined in the system, and expand it according to its definition. For example, replace \(\text{fib}(x + 2)\) by \(\text{fib}(x + 1) + \text{fib}(x)\). The rule especially applies to expressions of the form \(f(a)\) or \(f(x + a)\), where \(a\) is a constant.

(4) Conditional Split: Replace an expression of the form \(\text{if } P \text{ then } Q \text{ else } R\) by \((P \supset Q) \land (\neg P \supset R)\).

(5) Conditional Derivation: To prove a theorem of form \(P \supset Q\), assume \(P\) and prove \(Q\).
(6) \(\wedge\)-Split: To prove a theorem of form \(P \wedge Q\), prove
P and prove Q. When an assertion of form \(P \wedge Q\) is
made, assert P and assert Q.

(7) Functional Split: To prove a theorem of form \((\exists z)\)
\[z = f(t_1, \ldots, t_n),\] prove a theorem \((\exists z_1)\)
\[z_1 = t_1 \wedge \ldots \wedge (\exists z_n)\]
\[z_n = t_n.\]

(8) Equality: To prove a theorem of form \(t_1 = t_2\), where
the \(t_i\) are terms, replace the existentially quantified
variables of the \(t_i\) so that the two resulting
terms are identical.

(9) Change of Variables: Replace an expression of form
\((\forall x) [x \geq a \supset P(x)]\), where \(x\) is a natural number,
by \((\forall x) [P(x + a)]\) (replacing \(x\) by \(x - a\)).

(10) Simplification: Replace \(1 + 1\) by 2, \(0 \cdot X\) by 0,
and make other such improvements.

These rules are roughly stated; for example, the forms that
\(\wedge\)-split, conditional split, and the equality rule are applied to may have
certain quantifiers. In practice these rules would be separate, complex
programs in the QA4 language.

Now let us examine the behavior of the system when faced with
the program synthesis problem. We first assert

(11) Assert \(\text{fib} = \lambda x \; \text{if } x \leq 1 \; \text{then } 1 \; \text{else } \text{fib}(x - 1) + \text{fib}(x - 2)\).
(12) Assert ($\forall x \ (x \leq 1 \Rightarrow \text{fib}(x) = 1)$

and

(13) Assert ($\forall x \ (x \geq 2 \Rightarrow \text{fib}(x) = \text{fib}(x - 1) + \text{fib}(x - 2)$).

To produce (13) the system used the simplifier to replace $\neg (x \leq 1)$ by $x \geq 2$; we will not always mention the actions of the simplifier explicitly. We then give the system the goal

(14) Construct an iterative program that satisfies the input-output relation, $z = \text{fib}(x)$, where $x$ is the input and $z$ is the output, and \texttt{fib} is not taken to be "primitive."

The condition that \texttt{fib} not be primitive means that \texttt{fib} is not permitted to appear in the iterative program. This restriction is intended to prevent the system from producing the following iterative program.

![Diagram]

(This program is correct, iterative, and every bit as inefficient as the original recursive program.)
When the system is given this program-synthesis goal, it may transform it into a theorem-proving goal by using a standard technique. ¹

Thus, it produces the new goal

(15) Prove (∀x)(∃z) z = fib(x).

From its collection of inference rules, the system selects those that seem relevant to the proof of this theorem. These are induction, equality, and resolution (against 12 or 13). Induction is an expensive routine; we will defer trying it until we have explored the other possibilities. Equality tries to substitute fib(x) for z; however, the stipulation that fib is not primitive prevents that substitution from being made; otherwise, the proof would be concluded and the trivial program above would be produced. In this case, however, the equality rule fails. "Resolution" of (15), with (12) produces

(16) Prove (∀x) x ≥ 2 ⊃ (∃z) z = fib(x).

This goal is more attractive than the original goal (15) because it is a special case of (15); (16) is the consequent of (15). Therefore, the attention of the system is focussed on (16), and work on (15), including application of the induction rule, is delayed. The system then selected those rules that seem relevant to the proof of (16). The rules selected include change of variables (9), conditional derivation (5), and induction. Change of variables is applied before the other rules, producing a new goal
(17) \((\forall x)(\exists z) z = \text{fib}(x + 2)\).

The form of (17) suggests the immediate application of the partial evaluation rule (3). This produces (with simplification)

(18) Prove \((\forall x)(\exists z) z = \text{fib}(x - 1) + \text{fib}(x)\).

This goal is in the proper form for functional splitting (7).

The new goal,

(19) Prove \((\forall x)[(\exists z_1) z_1 = \text{fib}(x + 1) \land (\exists z_2) z_2 = \text{fib}(x)]\),

is produced. Although the form of this expression suggests \(\land\)-splitting, this tack quickly proves to be a dead end: of the two goals produced,

(20) Prove \((\forall x)(\exists z_1) z_1 = \text{fib}(x + 1)\) and

(21) Prove \((\forall x)(\exists z_2) z_2 = \text{fib}(x)\),

the second proves to be identical to the original goal (15). Since both these goals must be achieved in order that (19) be achieved, both (20) and (21) are discarded. Having exhausted the other possibilities, the system ventures to try induction on (19). The two new goals generated are:

(22) Prove \((\exists z_1) z_1 = \text{fib}(1) \land (\exists z_2) z_2 = \text{fib}(0)\), and

(23) Prove \((\forall x)[((\exists z_1) z_1 = \text{fib}(x + 1) \land (\exists z_2) z_2 = \text{fib}(x)) \lor

((\exists z'_1) z'_1 = \text{fib}(x + 2) \land (\exists z'_2) z'_2 = \text{fib}(x + 1))]\).

Both these goals must be achieved if the theorem is to be proved. The system considers the first goal first. The most appropriate rule to be applied is \(\land\)-split, which produces two new goals,
(24) Prove \((\exists z_1)\ z_1 = \text{fib}(1)\) and

(25) Prove \((\exists z_2)\ z_2 = \text{fib}(0)\),

both of which must be achieved. Partial evaluation applies to both goals, producing

(26) Prove \((\exists z_1)\ z_1 = 1\) and

(27) Prove \((\exists z_2)\ z_2 = 1\).

Then the equality rule is applied to each of these goals with success, so that (22) has been achieved. Attention now focusses on (23). Conditional derivation (5) allows us to make the assumption

(28) Assert \((\exists z_1)\ z_1 = \text{fib}(x + 1) \land (\exists z_2)\ z_2 = \text{fib}(x)\), and create the goal

(29) Prove \((\exists z'_1)\ z'_1 = \text{fib}(x + 2) \land (\exists z'_2)\ z'_2 = \text{fib}(x + 1)\).

The \(^\lor\)-split rule, applied to the assertion (28), produces two new statements,

(30) Assert \((\exists z'_1)\ z'_1 = \text{fib}(x + 1)\) and

(31) Assert \((\exists z'_2)\ z'_2 = \text{fib}(x)\).

The same rule, applied to the goal (29), results in the establishment of two other goals

(32) Prove \((\exists z'_1)\ z'_1 = \text{fib}(x + 2)\)

and

(33) Prove \((\exists z'_2)\ z'_2 = \text{fib}(x + 1)\),

both of which are to be achieved.
The resolution rule applies between goal (33) and assertion (30) resulting in a success. Partial evaluation, applied to goal (32) constructs:

(34) Prove \((\exists z_1)' z_1' = \text{fib}(x + 1) + \text{fib}(x)\).

As before, function splitting produces

(35) Prove \((\exists z_3) z_3 = \text{fib}(x + 1) \land (\exists z_4) z_4 = \text{fib}(x)\),

and \&-split produces

(36) Prove \((\exists z_3) z_3 = \text{fib}(x + 1)\) and

(37) Prove \((\exists z_4) z_4 = \text{fib}(x)\).

These goals resolve with assertions (30) and (31) respectively, completing the proof.

We have included mostly those steps in the search that actually did lead to the proof. The system would examine some of the false paths too, although it does not rely on blind search and discontinues a line of reasoning when another appears more profitable.

Program synthesis techniques allow us to produce the program illustrated in Figure 1, from the proof. This program turns out to be far more efficient than the original recursive program.

In this section we have discussed the behavior of a problem solver without specifying a mechanism that exhibits this behavior. In the next section we outline a system capable of carrying out such reasoning.
IV The QA4 PROBLEM SOLVER

This section gives an overview of the goals, overall structure, and flow of control of the QA4 problem solver.

A. Goals

- The problem solver should be easy to guide with intuitive knowledge about various forms of problem solving. If we run a proof, for example, and we see the problem solver doing an obviously stupid thing, then it should be possible to modify the proof strategy or give additional information in an easy way so that the system does not make
the same errors in a second run of the problem. Thus, the problem solver should also be easily modifiable.

- A large body of pragmatic information in the system
- A natural and compact formulation not only of goal statements but also of strategies in a unified language.

For example, we would not write the theorems to be proved in first-order predicate calculus while writing strategies in LISP.

B. Statements

The system is given information with four sorts of statements:

- Goal statements: e.g., Prove $(\forall x)(\exists z) z = \text{fib}(x)$
- Assertions: e.g., $\text{factorial} = \lambda x \text{ if } x = 0 \text{ then } 1 \text{ else } x \cdot \text{factorial}(x - 1)$
- Eval rules: e.g., change of variables $(\forall x) x \geq a \Rightarrow P(x) \text{ transforms to } (\forall x) P(x + a)$
- Strategies: e.g., a linear equation solver.

The goal statements and assertions are analogous to the theorems and axioms of a resolution-type theorem prover. The eval rules and strategies are expression transformation rules.

An eval rule is a single-expression transformation rule. It takes an input expression, matching a pattern given in the first half
of the eval rule, and transforms it under given conditions (when a predicate is true) into an output expression according to the second half of the eval rule.

A strategy is a program made up of control statements, eval rules, and other strategies. The program tells how to apply several transformations, sequentially or in parallel, for example.

C. Basic Method

The system is goal-directed. A problem entered in the system is the first goal statement. The system tries to find eval rules and strategies that may aid in achieving the goal. From these rules it constructs a single strategy associated with the goal. This strategy is applied to the goal; if this strategy does not succeed at once, the system may create one or more subgoals. In the same way, subgoals are given associated strategies, which control their processing.

The eval rules and strategies relevant to a given goal or assertion are selected by the "filter."

D. The Filter

The filter is a program that analyzes expressions and the associated semantic and pragmatic information kept on the expression's property list. The filter's main task is to find in an efficient way all eval rules, strategies, and typed-in pragmatics applicable to
(matching with) a given expression. After it has found the relevant information, a combined strategy is put together, put on the property list of the expression, and given to the interpreter.

E. How Statements are Processed

Let us see how the system processes each sort of QA4 input statement. First, consider the case of an assertion given to the filter. An assertion must be entered in the data base of the problem solver. It is possible that whenever an assertion of a certain form (matching a given pattern) is made, other assertions also should be made. We can give a great number of this sort of rules in the form of eval rules. An example is the conditional-split rule, which is applicable to the assertion fib = \lambda x_1 \cdot \text{if} \ldots \text{in the example of Section IV. Two additional assertions must be made according to this rule. Matching rules are found by the filter. A strategy is made up and interpreted that puts the initial assertion and the assertions discovered by the filter in the data base.}

In the case of a goal statement, an expression is given to the filter together with advice. For example, the goal statement "Prove \( z = \text{fib}(x) \)" is given to the filter, together with constraints and advice, such as: "the given expression is an input/output relation, this is a program-writing problem, write an iterative program." The filter tries to find the relevant eval rules and strategies with the information
residing in the filter. It will do some pattern matching to find relevant expression transformation rules, and use the constraints and advice given, together with the goal statement, in the search for the right rules. In the example, the filter puts the strategy "try the theorem-proving approach" together. This strategy creates the new subgoal "Prove (∀x)(∃z) z = fib(x)." The strategy gives the subgoal, together with the advice "try only techniques that give iterative solutions," to the filter. Now the whole procedure will be repeated until success is achieved and the goal can be proved true.

The filter is changed by entering new eval rules and strategies. The front end of an eval rule (a pattern) will get its proper place
among the already collected patterns in the filter; e.g., the eval rule change of variables will cause the filter to be updated with the pattern \((\forall x) x \in a \supset P(x)\). When an expression of that form is passed through the filter, the change-of-variables rule will be selected.

F. How Problems are Solved

All strategies, eval rules, the filter, a simple monitor, and other high-level programs of the problem solver are written in the QA4 language. For this language, a simple LISP-like EVAL is being written.

The flow of control in the system is governed by strategies, interpreted by a simple monitor. Strategies are put on property lists of expressions according to certain conventions. The task of the monitor is to interpret strategies under a set of conventions. The monitor also hands expressions to the filter and utility functions; for example, a function that puts typed-in information about a problem statement on the property list of this expression. The monitor interprets the control functions and in general connects the complex of strategies and system functions. The task of the monitor is, however, a mechanical task: All "cleverness" resides in the strategies.

The situation of a strategy creating one subgoal can get more complex when more eval rules or strategies are applicable; e.g., in the example of the fib function: Try partial evaluation, resolution, or induction. Now the system can work on one subgoal, but should not give
up on the other subgoals. It could work for a time on the goal generated by the partial evaluation but then decide that the goals are getting worse (compared with the original) and try the induction step.

To be able to work in such a fashion, a set of functions for controlling strategies are available. They will be all realized with a simple coroutine mechanism that makes use of the contexts as described in Section III.

G. Control Functions

To give the flavor of the control functions, some are described below. A strategy can create two or more goals and ask the problem solver to prove them all. An example is induction, in which two subgoals (the zero case and the step case) must be proved true. The system uses for this purpose the AND statement (AND set strategy). All the strategies in the set are run in parallel, and the relative speed of each program is controlled by the strategy. Sometimes it is necessary for a program in the set to communicate with the controlling strategy. For example, the program sees its progress is poor and wants to give this information to the controlling strategy of the AND, so that another program in the set can be given a turn or other action can be taken. For this purpose a program (strategy) can use the WAIT statement (WAIT x). The value of x is given to the strategy associated with the AND and the calling program is suspended. The OR statement (OR set strategy) operates in a similar
way. For example, in the Fibonacci problem, three alternative rules are proposed for the goal \((\forall x)(\exists z) \, z = \text{fib}(x)\): induction, resolution, and the equality rule. These rules are combined by an OR statement and equality is tried first, but fails. Now resolution is selected by the strategy associated with the OR statement. Induction is only tried when the resolution strategy fails or produces poor results, in which case a return to the OR statement is made. In the case of the Fibonacci example the resolution was successful.

H. Advice to the System During a Proof

The problem solver is able to take advice during a proof. A natural point to do this is whenever a strategy calls the filter and gives a new goal (or new goals) to be analyzed. We can envision among others two ways of giving advice:

(1) Changing a strategy, mainly strategies controlling AND and OR statements; and

(2) Supplying a new strategy in the set of an AND or OR, which gives rise to a new subgoal.
REFERENCES


