April 1971

A LANGUAGE FOR WRITING PROBLEM-SOLVING PROGRAMS

by

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

Paper accepted for presentation at IFIP Congress '71,
Ljubljana, Yugoslavia, August 23-28, 1971

Artificial Intelligence Group

Technical Note 48

SRI Projects 8259, 8721, 8550

The research reported herein was sponsored by the Advanced Research Projects Agency and the National Aeronautics and Space Administration under Contracts NAS12-2221, and NASW-2086, and by Air Force Cambridge Research Laboratories under Contract F19628-70-C-2046.

A LANGUAGE FOR WRITING PROBLEM-SOLVING PROGRAMS

JOHNS F. RULIFSON
RICHARD J. WALDINGER
JAN A. DERKSEN
Stanford Research Institute,
Menlo Park, California

This paper describes a language for constructing problem-solving programs. The language can manipulate several data structures, including ordered and unordered sets. Pattern matching facilities may be used in various ways, including the binding of variables. Implicit backtracking facilitates the compact representation of search procedures. Expressions are treated analogously to atoms in LISP. A "context" device is used to implement variable bindings, to effect conditional proofs, and to solve the "frame" problem in robot planning.

I BACKGROUND

In order to design a deductive problem-solving program, we are constructing a new formal language that can express complex inferential mechanisms concisely. This language, called the QA4 language, is being used to build a proposed intelligent system, that will be able to organize and use a large body of specialized knowledge. The selection of three specific applications--automatic program synthesis, automaton planning, and theorem proving--permits a concentration of effort within a framework of generality. All three applications, however, share a common basis that encompasses natural language dialogue, question answering, and inference, as well as many other areas of Artificial Intelligence.

There is strong motivation for the development of a new language for problem-solving programs. Earlier systems have been constrained by fixed inference mechanisms, built-in strategies, awkward languages for problem statement, and rigid overall structure. They relied on one or two rules of inference that could not be changed. To modify a strategy required a complete reprogramming of the system. It was sometimes harder to express a program-synthesis problem in a language the system could understand than it was to write the program oneself. Systems were limited to the use of a single paradigm that might be applicable to some types of problems and inappropriate for others. Theorem-proving strategies have used syntactic properties of the expressions being manipulated, but have been unable to use semantic knowledge or pragmatic, intuitive information. They have been unable to employ the sort of pattern recognition the human problem solver relies on so heavily.

The basic approach of the QA4 project is to develop natural, intuitive representations of problems and problem-solving programs. The specification for a computer program to be synthesized, for example, is a blend of procedural and declarative information that includes explicit instructions, intuitive advice, and semantic definitions. The problem-solving programs are a similar blend of many special purpose inference mechanisms and heuristic strategies. A QA4 interpreter will execute programs in the transparent but precise language we have chosen for these representations; and the interpreter, together with an initial collection of QA4 "bootstrapping" programs, will constitute the basic QA4 system. The system will attempt to assimilate new advice facts and attempt to solve problems with continually increasing agility.

The project has revolved around the construction and reworking of hand simulations of a proposed final QA4 system. Each simulation includes a problem statement, relevant definitions and advice, and a protocol for the solution. These simulations have provided a focus for the language development and promoted explorations into theoretical foundations of the problem areas [1]. Table 1 summarizes the scope of the simulations and indicates the problem areas attacked.

The QA4 language is derived from more conventional programming languages and mathematical languages, and yet differs from both in many ways. The basic data structures include sets, sets with repeated elements ("bags"), ordered bags ("tuples"), and lambda expressions. Data may also be represented implicitly; for example, the set of even integers, although infinite, may still be described and manipulated in the system. Every expression in the language may have a variety of properties associated with it. This feature serves as the basis for describing the role of each expression and thus directing the processes that operate on the expression. WHEN and GOAL statements, using pattern matching, may direct program control. Various strategies, for example, might be selected for use by matching the pattern of their formal argument to the expression under consideration,

rather than by the conventional method of activating a subroutine by mentioning its name. Ambiguous patterns lead to nondeterministic programs and the need for automatic backtracking [3]. Other control features include parallel search and iteration through sets. Space does not permit a full presentation of our current, preliminary version of the QA4 syntax. Table 2 lists some features of the language.

Table 1

PROBLEM AREAS

Program Synthesis

- Generate recursive and iterative programs from declarative axiomatic specifications (problems taken from a LISP primer).
- Generate an iterative program from a recursive procedural definition (see Fibonacci example in [2]).
- Verify the correctness of programs with respect to input/output relations. These relations may be defined in terms of executable programs, as well as in terms of declarative axioms.

Automaton Planning

- Generate plans for simple robot problems.

Theorem Proving

- Prove simple algebraic identities over the integers.
- Derive simple algebraic laws from Peano's axioms.
- Prove properties of axiomatically defined groups.
- Accommodate general rules of interference, applicable to any logical system.

The system changes continuously as it is used. The programmer types commands in the form of QA4 expressions to a top-level function. The commands may input or modify expressions or properties of expressions; define, modify, or execute programs; or perform debugging tasks.

The input system of QA4 is a parser that transforms QA4 infix expressions into internal prefix format. The parser uses the input translater BIP[4], and has the advantage of being readily modified. Similarly, an output function takes the internal expression form and produces a corresponding infix output stream. Thus the user always communicates with QA4 in an infix mathematical-style notation.

Table 2

SOME LANGUAGE FEATURES

Data Manipulation

- Arithmetic and Boolean operations
- Set, bag and tuple operations
- Expression decomposition and construction

Pattern Matching

- Actual argument decomposition
- Data base queries
- Monitoring expression properties
- Invoking of strategies and inference rules

Control

- Standard serial and conditional statements (prog's, labels, go's and if's)
- Iterative forms for sets, bags, and tuples
- Automatic backtracking
- Strategy controlled parallel interpretation

The QA4 interpreter is a function resembling LISP EVAL [5]. It accepts QA4 expressions and, with the aid of an extensive library of primitive functions, executes them. Unlike LISP programs, QA4 expressions may succeed or fail and do not necessarily have values. The interpreter performs its task in small steps, and may, between any two steps, redirect its attention to other parallel processes or search programs.

II DATA CONTROL STRUCTURES

Every operator in QA4 has a single operand. The data type of each primitive operator has been chosen to eliminate a proliferation or rules governing algebraic properties, such as associativity, commutativity, and transitivity. The Boolean connective "and," for example, has a set as its operand. The infix expression A & B & C is translated into the internal representation AND{A, B, C}

(where braces denote the data type "set"). Since sets are independent of the order of their elements, this representation makes the statement of the commutativity law unnecessary.

Bags in QA4 are unordered tuples or, equivalently, sets with repeated elements. They play an important role in the definition of arithmetic operators, such as addition. The operand of PLUS cannot be a set, because the set {1, 1} is equal to {1}, but we would not want PLUS{1, 1} to equal PLUS{1}. Instead, the infix expression X + Y + Z becomes PLUS[X, Y, Z] internally, where [X, Y, Z] is a bag. Bags are evaluated by first evaluating their members. The resulting values are collected together into a bag. Thus, if X, Y and Z all had value 1, our expression would equal PLUS[1, 1, 1], and its value would be 3.

. Some expressions, infinite sets, for example, cannot always be explicitly evaluated. Finite sets may also be inconvenient to evaluate: A program may wish to search the Cartesian product of two sets, even when the entire set is too large to generate. The interpreter can perform the search by indexing through the original two sets. In cases such as this expressions are said to have implicit values.

QA4 has iterative, parallel, and backtracking control structures. The iterative statement forms of the language operate over sets, bags, and tuples. With each iteration statement may be associated an independent preference strategy that controls the order of iteration. During resolution theorem proving experiments, for example, pairs of logical expressions are analyzed in an order specified both in terms of syntactic properties, such as length, and of pragmatic properties, such as frequency of use. Parallel structures, in the form of coroutines and WHEN statements, are used in the construction of problem solving strategies. For example, in order to prove a theorem of the form A ∨ B, we may wish to establish two processes, one to prove A and the other to prove B. If either terminates successfully, the proof is complete. Nondeterministic programs give rise to backtracking. If a point of indeterminacy occurs, a choice determined by a prespecified strategy is made. If the program later fails, control is reestablished at the choice point, and a different selection is made.

. Without explaining all the notations we use, we illustrate the power of the QA4 language with a program that sorts bags:

```
SORT = CASES
     (λ [ ], ( ),

      λ X•B ↑ MIN(X,B), X•SORT(B));    .
```

When this function is applied to a bag, say B′, it first checks to see whether B′ is empty ([ ]); if so, it returns the empty tuple (( )). If the bag is not empty, it finds an element X of B′ such that X is less than or equal to all the elements of B, the bag remaining when X is deleted from B′. Then it sorts B (recursively) and adjoins X to the front of the resulting tuple. Thus the bound variables of a lambda expression may be patterns, and variable binding must then be done by pattern matching. With each iteration statement may be associated an independent preference strategy that controls the order of iteration.

III EXPRESSIONS

The data base for QA4 is made up of QA4 expressions. An expression is represented internally by a list of properties, one of which is the syntactic component that uniquely distinguishes it from all other QA4 expressions. This list stores arbitrary properties, and each property is, in turn, a QA4 expression. These properties fall into three categories: syntactic, semantic, and pragmatic. Table 3 is a brief example of an expression. Table 4 lists commonly used properties.

Table 3

A SAMPLE EXPRESSION

Syntactic component	⟨X, Y⟩
Value	⟨3, 4⟩
Length	2
Result when the function F is applied to the expression	27

Expression manipulation is accomplished by decomposition and construction. In QA4 decomposition means naming parts or components of an expression. The naming is done by the pattern matcher. Patterns may occur at many points in the language: in formal arguments of functions, in assignment statements, and in conditional tests. Table 5 illustrates some of the more useful facets of the pattern matching notation. Transformation of expressions is done through a complete set of constructors [7], such as: add an element to a set, add onto tuples, or construct a lambda expression.

Given the syntactic component for an expression, a fundamental operation is to retrieve the entire expression so as to find the properties already assigned or known about it. In this way, LISP's atom property feature is extended to expressions in general. When an expression is stored, whether the expression has been stored before is determined. If it has been, the old expression is returned; if not, the new expression is retained by the system.

Table 4

EXPRESSION PROPERTIES

Syntactic

- The form
- The logical type (e.g., a function mapping numbers into truth values)
- The data type (e.g., a set of 3-tuples)
- Frequently used information (e.g., the length)

Semantic

- The value
- An implicit value (e.g., a coroutine[6] that generates the value)
- A set of expressions equal to this one
- Constraints (e.g., a range or interval for the value)

Pragmatic

- Historical information
- Intuitive evaluation advice
- Success/failure indicators

The storage mechanism is a discrimination net. Each node of the net consists of a feature selector and a set of labeled branches. A syntactic component is retrieved by applying the topmost selector, choosing a branch based on the outcome of the selector, and repeating the process until either a terminal node is reached or there is no appropriate branch. When conflicts occur at a terminal node, a new selector is automatically generated and installed at the new node. The next time the same syntactic component is retrieved, the expression that has just been added will be returned. The net also serves as a pruning device for the pattern matcher.

It two QA4 expressions are identical except for the names of their bound variables, they have the same internal representation. Thus bound variables are not used as discrimination features. 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 the expression is stored. If the same set is then stored a second time (perhaps with some elements permuted), the elements are first ordered by their index numbers and then discriminated upon syntactically. If a user types in the set {A, B, C}, the elements might be 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 occur already. The storage and retrieval functions also maintain extensive statistics concerning the number of references made to each expression for use in future optimization.

Table 5

SOME PATTERN MATCHER FEATURES

- Transparent template notation

  ⟨X,4,3⟩ matches ⟨5,4,3⟩
  with X = 5

- Matching of internal or external notation

  ⟨X,4⟩ matches (TUPLE 3 4)
  with X = 3

- Fragment variables

  ⟨2, *Y⟩ matches ⟨2,3,4,5⟩
  with Y = ⟨3,4,5⟩

- Type constraints on variables

  X/INTEGER matches 3
  with X = 3

- Predicate constraints

  ⟨X ↑ X ≥4, 5, 6⟩ does not match
  ⟨2,5,6⟩

- "Occurs in" matching

  ·· + ·· matches 3 * 2 + 5

Variable bindings are implemented in the QA4 interpreter with a "context" mechanism--a method of storing all the changeable properties of expressions which simplifies backtracking and the execution of parallel processes. The same facilities, moreover, are made available to users and are especially useful in programs dealing with conditional proofs or robot planning programs confronted with the "frame" problem [8].

IV CONCLUSION

Certain structures and mechanisms have found repeated applieation in deductive problem solvers. It is our goal to give these concepts concise notations. We expect this effort to have several desirable consequences:

- Existing problem-solving techniques should become more easily representable and modifiable.

- A large store of special-purpose knowledge could be embodied in a program.

- Systems would be more likely to rely on strategies than on blind search if such strategies were easily expressed and incorporated.

A preliminary version of QA4 has been implemented. Extensions of this system will include enhanced pattern language and strategy operations, and more efficient evaluation.

V ACKNOWLEDGEMENT

The ideas presented in this paper were developed in collaboration with C. Cordell Green and Robert A. Yates. The development and use of the QA4 system has been supported at Stanford Research Institute by the Advanced Projects Agency and the National Aeronautics and Space Administration (NASA) under Contract NAS12-2221, by NASA under Contract NASW-2086, and by Air Force Cambridge Research Laboratories under Contract F19628-70-C-0246.

REFERENCES

[1] Z. Manna and R. J. Waldinger, Towards Automatic Program Synthesis, CACM, Vol. 14, No. 3, (March 1971) pp. 151-165.

[2] J. F. Rulifson, R. J. Waldinger, and J. Derksen, QA4 Working Paper, Artificial Intelligence Group Technical Note No. 42, Stanford Research Institute, Menlo Park, California (October 1970).

[3] L. Golomb and L. Baumert, Backtrack Programming, JACM, Vol. 12, No. 4 (October 1965) pp. 516-524.

[4] R. E. Fikes, A LISP Implementation of BIP, Artificial Intelligence Group Technical Note No. 22, Stanford Research Institute, Menlo Park, California (February 1970).

[5] J. McCarthy, et al., LISP 1.5 Programmer's Manual (M.I.T. Press, Cambridge, Massachusetts 1962).

[6] M. E. Conway, Design of a Separable Transition-Diagram Compiler, CACM, Vol. 6 (1963) pp. 396-408.

[7] P. J. Landin, The Mechanical Evaluation of Expressions, Computer J., Vol. 6, (1963/64) pp. 308-320.

[8] J. McCarthy and P. Hayes, Some Philosophical Problems from the Standpoint of AI, in Machine Intelligence 4, B. Meltzer and D. Michie, eds. (Edinburgh University Press, Edinburgh, Scotland 1969).