

April 1970

PRELIMINARY SPECIFICATION OF THE QA4 LANGUAGE*

by

Johns F. Rulifson

Artificial Intelligence Group

Technical Note 50

SRI Project 8259

The research reported here was sponsored by the Advanced Research Projects Agency and the National Aeronautics and Space Administration under Contract NAS12-2221.

*This note originally appeared as Section V of "Research and Applications--Artificial Intelligence," Interim Scientific Report, Contract NAS12-2221.

PRELIMINARY SPECIFICATION OF THE QA4 LANGUAGE

A. Introduction

The long-term problem-solving effort of the Stanford Research Institute Artificial Intelligence project has been involved with the design and implementation of a general-purpose, formal problem-solving system. The current system, termed QA4, is based upon mechanized theorem proving in higher-order logic and emphasizes the role of semantic information and flexible control strategies. Two major applications of such a system are in the field of automatic program writing and in robot planning and problem solving.

The QA4 system represents a significant extension and modification to the ideas incorporated in a resolution-based first-order theorem prover such as QA3.5. First the system is intended to be semantically rather than syntactically oriented. The methods or procedures used by the system to solve a particular problem should be related to the semantics of that problem and not applied uniformly to all problems. Secondly, QA4 is embedded in the formal framework of ω -order predicate calculus. This provides a much richer language than first order and permits the syntax and semantics of any portion of the system to be expressed in its own language. Moreover the addition of sets to the language provides a powerful interface between logical operations and computational operations (such as set enumeration etc.). Finally, the QA4 strategies will be user-defined and expressed directly in the QA4 language rather than being "frozen into" the problem-solver's code.

The following discussion is a description of QA4 at its present state of development.

B. The Logic Language

The current QA4 language is an extension of higher-order logic as defined by Robinson.^{1*} Set, bag, and tuple expressions and operations have been added, and the bound variable occurring in LAMBDA expressions has been significantly modified.

1. The Class of Expressions

A QA4 expression falls in one of seven syntactic categories: identifiers, numbers, applications, set expressions, tuple expressions, bag expressions, and bound-variable expressions.

Identifiers. An identifier is an individual symbol such as X, Y, MAX, etc. The identifiers are the function symbols, predicate symbols, and variables of the language. Certain identifiers such as AND, OR, UNION, etc. are called special in that they have predetermined, built-in meanings. All other identifiers are called nonspecial and may generally be used for variables, defined functions, etc.

Applications. An application is an expression of the form $F(A)$ [alternatively (FA) or FA] where F is an expression denoting a function and A is any expression. All QA4 functions take one argument; however, the argument can be a tuple $\langle A_1, A_2, A_3, \dots \rangle$ so there is no loss of generality. The meaning of an application $F(A)$ is its natural one--namely the result of applying the function (denoted by) F to the argument (denoted by) A . QA4 has an infix language that will be used in the remainder of the discussion. If one writes $A \Rightarrow B$, the symbol \Rightarrow is an infix operator and the expression is translated into the application $\text{IMPLIES} \langle A, B \rangle$ where $\langle A, B \rangle$ is the single argument of IMPLIES . Similarly $3 + X + Y$ is translated into $\text{PLUS}[3, X, Y]$ where $[3, X, Y]$ is a bag (discussed below). Thus the infix expression should be understood as an abbreviation of a corresponding application.

*References are listed at the end of this technical note.

Sets. A set expression is an expression of the form

$\{E_1, \dots, E_n\}$ (the QA4 parser uses [: for {, and :] for }),

where E_1, \dots, E_n are any expressions. The meaning of the set expression $\{E_1, \dots, E_n\}$ is the set of objects denoted by E_1, \dots, E_n . Since the order of the elements E_1, \dots, E_n in a set is immaterial--as well as multiple occurrences of elements--the sets

$\{A, B, C\}$ $\{C, A, B\}$ and $\{C, A, A, B, C\}$

are treated as identical expressions.

Tuples. A tuple expression is an expression of the form

$\langle E_1, \dots, E_n \rangle$,

where E_1, \dots, E_n are expressions. Its meaning is the n-tuple $\langle \bar{E}_1, \dots, \bar{E}_n \rangle$ of objects, where E_i denotes the object \bar{E}_i . Two tuples $\langle E_1, \dots, E_n \rangle$ and $\langle F_1, \dots, F_m \rangle$ are logically equal provided they have the same length ($n=m$) and each E_i is logically equal to F_i . So we have $\langle 3+1, 2-1 \rangle = \langle 4, 1 \rangle \neq \langle 1, 4 \rangle$.

Tuples are used as arguments to functions generally demanding more than one argument. Thus a function $f(x,y)$ in mathematics would be represented in QA1 as a function $F\langle X,Y \rangle$ where F takes the tuple $\langle X,Y \rangle$ as its single argument.

Bags. A bag expression is an expression of the form $[E_1, \dots, E_n]$, where E_1, \dots, E_n are arbitrary expressions. A bag is like a set except that elements may have multiple occurrences. For example, $[2, 3, 2] = [2, 2, 3] \neq [2, 3]$. Bags are used as arguments to functions such as + (plus) and * (times) that are commutative and associative. Thus $7 = +[2, 2, 3]$.

Numbers. A number is simply a positive or negative integer at present; e.g., 3, 0, -2 are numbers.

Bound Variable Expressions. A bound variable expression is an expression of the form (keyword BV E), where E is any expression, BV is a bound variable, and keyword is one of the following: LAMBDA, FORALL, EXISTS, CHOICE, ... So all LAMBDA expressions and quantified expressions are bound variable expressions. The bound variable, BV, has its own syntax--and semantics that extend the usual definition. It is more akin to a variable declaration. Basically, the purpose of a bound variable is to assign values to one or more variables for a temporary duration--namely for the evaluation or analysis of the expression E, the body of a bound variable expression.

A complete bound variable is a triple (name structure predicate) where name is an identifier, structure is a tuple, bag, or set of BVs (or a decomposition structure), and predicate is any truth-valued expression. When the bound variable is bound to an expression, the name is set equal to the expression, the expression is decomposed if possible according to the structure, and the predicate is tested on the results of the decomposition.

For example, (X $\langle Y,Z \rangle Y+Z<5$) is a bound variable having X as its name, $\langle Y,Z \rangle$ as its structure, and $Y+Z<5$ as its predicate. If we attempt to bind this bound variable to the expression $\langle 3,1 \rangle$, we set $X = \langle 3,1 \rangle$ and then bind $\langle Y,Z \rangle$ to $\langle 3,1 \rangle$ by setting $Y=3$ and $Z=1$. Then $Y+Z<5$ is tested and found true, so the binding succeeds.

More precisely, after assigning the name to the expression, the structure (if present) is examined. If the structure is a tuple (of bound variables), then the expression must itself be (or denote) a tuple of the same length. Then the elements of this structure are recursively bound to the corresponding elements of the expression.

If the structure is a set or bag of bound variables, then again the expression must be a set or bag respectively and the bound variables of the set (or bag) structure are bound to elements of the expression.

Here the situation is more complex since there is more than one possible assignment. The predicate is used to restrict the possible assignments to the one desired.

For example, to bind $(S \{X,Y,Z\} X<Y \ \& \ Y<Z)$ to the set $\{3,1,2\}$ the only possible assignment is $X=1, Y=2, Z=3, S=\{3,1,2\}$.

One further structure is the structure decomposition that takes the form $BV1 \cdot BV2$ or $BV1 :: BV2$. The dot and double colon are termed "cons" and "append" decompositions respectively. To illustrate how the binding takes place in this case we give an example. Bind $(T \ X \cdot T1)$ to $\langle 3,2,4 \rangle$. X is set to 3, $T1$ is set to $\langle 2,4 \rangle$, and T to $\langle 3,2,4 \rangle$. Thus $X \cdot T1$ decomposes the tuple $\langle 3,2,4 \rangle$ into its first element 3 and the remainder $\langle 2,4 \rangle$. (The absence of the predicate means simply that no predicate test is made.)

An example of the append decomposition is BIND

$(X \ Y :: Z \ \text{length} \ (Y) = 2)$ to $\langle 3,2,4 \rangle$ assigns $X = \langle 3,2,4 \rangle$, $Y = \langle 3,2 \rangle$, and $Z = \langle 4 \rangle$. The elements Y and Z must both be tuples, Y of length 2 and Z arbitrary; Y and Z appended together must yield the bound expression.

LAMBDA Expressions. A LAMBDA expression is a bound variable expression of the form $(\text{LAMBDA } BV \ E)$. A LAMBDA expression denotes a function whose value at an argument A is the value of E with the variables of E set to the result of binding BV to A . The main use of LAMBDA expressions is in defining new functions. An example is $(\text{LAMBDA } \langle X,Y \rangle$, where $\langle X,Y \rangle$ is a function that revises a tuple of length 2. $\langle X,Y \rangle$ is the BV , and $\langle Y,X \rangle$ is the expression body.

Quantified Expressions. A quantified expression is an expression of the form $(\text{quantifier } BV \ E)$, where E is truth-valued and quantifier is either the universal quantifier FORALL or the existential EXISTS. The mathematical statement $\forall x \forall y \forall z \ P(xyz)$ is expressed in QAD as $(\text{FORALL } \langle X,Y,Z \rangle \ P\langle X,Y,Z \rangle)$. (Similarly for EXISTS.)

2. Primitive Operations

The primitive QA4 operations can be broadly separated into three categories: logical operations, set and tuple operations, and arithmetic operations. The following list gives most of the basic operators.

Logical Operators

$\text{AND}\{P_1, \dots, P_n\}$

The operator AND takes a set of truth values and returns true if the set consists of the singleton {true} and false otherwise. Thus $\text{AND}\{P_1, \dots, P_n\}$ is true provided all the expressions P_1, \dots, P_n denote true. In the infix language we could write $P_1 \ \& \ P_2 \ \& \ \dots \ \& \ P_n$.

$\text{OR}\{P_1, \dots, P_n\}$

OR is analogous to AND except that it returns true if true is a member of the set and false otherwise. In the infix language we have $P_1 \ \vee \ P_2 \ \vee \ \dots \ \vee \ P_n$.

Since AND and OR are both commutative and associative, they have been made into a set as an argument rather than a tuple.

$\text{EQUAL}\{E_1, \dots, E_n\}$

EQUAL asserts that all of the members of the set are logically equal--and denote the same element. In the infix language we have $E_1 = E_2 = \dots = E_n$.

$\text{NOT}(P)$

NOT negates the truth value of its argument. In the infix language we have $\#P$.

IMPLIES $\langle P_1, P_2 \rangle$

IMPLIES asserts that P_1 implies P_2 .

Infix: $P_1 \Rightarrow P_2$.

Set, Bag and Tuple Operations

IN $\langle X, S \rangle$ asserts that X is an element of the set S .

Infix: $X \text{ IN } S$.

UNION $\{S_1, S_2, \dots, S_n\}$ is the set-theoretic union of the sets S_1, \dots, S_n .

INTERSECTION $\{S_1, \dots, S_n\}$ is the set-theoretic intersection of the sets S_1, \dots, S_n .

DIFFERENCE $\langle S_1, S_2 \rangle$ is the set-theoretic difference $S_1 \sim S_2$ of the sets S_1 and S_2 .

APPEND $\langle T_1, \dots, T_n \rangle$ adjoins the tuples (or bags) T_1, \dots, T_n so if $T_1 = \langle e_1^1, \dots, e_{m_1}^1 \rangle, \dots, T_n = \langle e_1^n, \dots, e_{m_n}^n \rangle$ then

APPEND $\langle T_1, \dots, T_n \rangle = \langle e_1^1, \dots, e_{m_1}^1, e_1^2, \dots, \dots, e_{m_n}^n \rangle$.

Infix: $T_1 :: T_2 :: \dots :: T_n$.

ADD $\langle X, T \rangle$ adjoins the element X to the tuple (or bag) T . If

$T = \langle X_1, \dots, X_n \rangle$ then $\text{ADD} \langle X, T \rangle = \langle X, X_1, \dots, X_n \rangle$.

NTH $\langle T, n \rangle$ extracts the n^{th} element from a tuple T . If

$T = \langle X_1, \dots, X_n, \dots, X_m \rangle$ $m \geq n$, then $\text{NTH} \langle T, n \rangle = X_n$.

Arithmetic Operations

PLUS $[n_1, \dots, n_m]$ forms the sum of the elements n_1, \dots, n_m in the bag argument.

Infix: $n_1 + n_2 + \dots + n_m$.

TIMES $[n_1, \dots, n_m]$ forms the product $n_1 \cdot n_2 \cdot \dots \cdot n_m$.

Infix: $n_1 * n_2 * \dots * n_m$.

GT(m,n) asserts that m is greater than n.

Infix: $m > n$.

LT(m,n) asserts that m is less than n.

Infix: $m < n$.

GTQ(m,n) asserts m is greater than or equal to n.

Infix: $m \geq n$.

LTQ(m,n) asserts m is less than or equal to n.

Infix: $m \leq n$.

C. Current Implementation

Although the design of the QA4 system is not complete, an initial implementation has been started to allow feedback and experimentation with many of the QA4 ideas. The implementation is being done in the LISP system on the PDP-10 and can be separated into three parts:

- (1) Input Output. An expression parser to take the QA4 infix syntax into a prepolish or internal format has been written. The parser uses the BIP (Appendix E)² package and has the advantage of being very easily modifiable. Similarly, an output function to take the internal expression form and output a corresponding infix version has been written. Finally, a top-level command language was designed to allow the user to enter and fetch expressions and properties of expressions from the data base and to do a variety of other simple tasks. This top-level function essentially interfaces the input output functions with the expression manipulation package.

- (2) Expression Storage and Manipulation

The Discrimination Net. In order to allow arbitrary semantic properties to be assigned to expressions and to allow expressions to be retrieved by these properties, a

discrimination net has been implemented to store all QA4 expressions. 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. When an expression is stored in the net, a discrimination on the syntactic properties of the expression is made to determine whether or not the expression has already been stored in the net. If so, the old net expression is returned, and, if not, the new expression is added to the net. Thus the net contains only one copy of each expression stored in it. Moreover, a check for expression equivalence up to bound variable names is being added so that two QA4 expressions that are identical except for the names of their bound variables go into the same internal net expression. 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 optimization.

Contexts. For the purpose of doing proofs by contradiction and conditional proofs (i.e., to prove $A \Rightarrow B$ assign A true and prove B), a context scheme was required in which a given expression in the net has a value relative to a context. Thus, in the above example, if B is proved true, it is true only in

the context in which A was assigned true. In order to have this ability, each expression in the net has a value property that consists of a context name (LISP atom) and a corresponding value for that name. A context c is an ordered list (a_1, \dots, a_m) of such names. If the value of an expression in context c is desired, the list (a_1, \dots, a_n) is examined in order until the first a_i is encountered for which the expression has a value--which is declared to be the value in that context. Thus to prove $A \Rightarrow B$ in a context $c = (a_1, \dots, a_m)$ we can create a new context name a_0 , assign A true in a_0 , and prove B in the context $c^1 = (a_0, a_1, \dots, a_n)$. If B is found to be true in c^1 , then $A \Rightarrow B$ is assigned true in context c. The context mechanism is also useful for bound variable expressions: to perform some operation on the body of the expression in the context where the identifiers in the bound variable have been assigned certain values.

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 correspondingly updated. 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.

Primitive Functions. A variety of primitive QA4 set and bag functions have been implemented. In particular, the functions UNION, DIFFERENCE, INTERSECTION have been written for finite sets and bags, and these functions are used as the basis for the equality partition code.

Simplification. A simplification package has been written to simplify algebraic and logical expressions. The simplification function expands products, collects like terms, deletes zeros and zero factors, and so on.

3. Strategy Control. The strategy control primitives involve operations such as evaluating a set of expressions and returning when one of the evaluations succeeds, or when all the evaluations succeed, or when "progress" has been made on any one of the set. Ideally, one would like the evaluations to proceed in parallel, be able to communicate results to each other, and possibly remain in suspended animation for a period of time until invoked to proceed. To achieve this type of flow of control, a co-routine package has been implemented within the LISP system. Unlike the LISP flow of control, which must return from any started evaluation, the co-routines permit processes to be started, interrupted for the purpose of invoking any other process, and returned to by a completely different route. We plan to use the co-routine facility to monitor and control QA4 strategies and effort allocations.

REFERENCES

1. J. A. Robinson, "Mechanizing Higher-Order Logic," in Machine Intelligence 4, Meltzer and Michie, eds. (Edinburgh University Press, Scotland, 1969).
2. R. E. Fikes, "A LISP Implementation of BIP," Technical Note 22, Artificial Intelligence Group, Stanford Research Institute, Menlo Park, California (February 1970).