A Logic Programming and Verification System for Recursive Quantificational Logic

Frank M. Brown
Dept. of Computer Science
The University of Kansas
Lawrence, Kansas 00045

Peiya Liu
Dept. of Computer Sciences
The University of Texas
Austin, Tx 78713

ABSTRACT

In this paper, we describe a logic programming and program verification system which is based on quantifier elimination techniques and axiomatization rather than on more common method of doing logic programming using the Herbrand-Prawitz-Robinson unification algorithm without occur-check.

This system is shown to have interesting properties for logic programming and includes a number of advanced features. Among these features are user-defined data objects, user-defined recursive relations and functions, either of which may involve quantifiers in the body of their definitions, and automatic termination and consistency checking for recursively defined concept. In addition, it has a correct implementation of negation in contrast to PROLOG implementation of negation as failure, a smooth interaction between LISP-like functions and PROLOG-like relations, and a smooth interaction between specifications and programs. Finally, it provides a method of mathematical induction applicable to recursive definitions involving quantifiers.

I. INTRODUCTION

Quantified Computational Logic(QCL) is a programming and program verification language for programs written in recursive quantificational logic. The language can be used both for logic programming and program specification.

A QCL program consists of sequences of recursively defined data objects declarations and recursive quantificational logic function definitions. Programs written in QCL are both executed and verified by a single automatic deduction system called the symbolic evaluator using a system of axioms and rules called the symmetric logic. The essence of the symmetric logic technique is to push quantifiers to the lowest scope possible in the hope of finding a way to eliminate them. The symbolic evaluator and the symmetric logic are described in detail [Brown 83].

A prototype system for QCL has been implemented in Interlisp.

II. QCL PROGRAMMING

A. Recursively Defined Data Objects

Although many logic programming languages provide system-defined data objects, they provide only a poor capability for user-defined data objects. QCL adopts the shell principle(Boyer&Moore 79] as a method to recursively define some useful user-defined data objects. Since the shell principle was invented, not much attention has been paid to it in logic programming. However, it is very valuable in modelling logic programs and verifying logic programs. Although many of the axioms produced by the shell principle of QCL are similar to those described in[Boyer&Moore 79], some are quite different in order to satisfy the fundamental deduction principle described in [Brown 83]. This principle requires that almost all steps in the automatic deduction should take place by a method of replacing expressions by an equivalent but simpler expressions.

The syntax of user-defined data object is: (SHELLCREATE constructor bottom recogniser selectors types defaults). The constructor is a name of new function which constructs objects of the new types. The bottom is the bottom object or T if there is no bottom object. The recognizer is a name of new function which recognizes objects of the new type. The selectors are a list of functions which are accessors of the data structure. The types are a list of type restrictions on each shell. Each restriction is an arbitrary formula in QCL consisting of symbols defined at that time. The defaults are a list of the default values for each shell. Data objects are easily declared. For examples, Lists are produced by the declaration (SHELLCREATE 'CONS 'T 'LLSTP '(CAR CDR) '(T T) '(NIL NIL)) and Trees are produced by the declaration (SHELLCREATE TREE 'BTM 'TREEP '(LEFT ROOT RIGHT) '(T T T) '(0 0 0)).

B. A Safe Definition Principle

Most existing logic programming languages have syntax checking but few perform logic checking for properties such as termination and consistency. Suppose P is a logic program and R is a goal we wish to solve. Then, executing P to solve R is the same as deducing R from P in logic program execution. Therefore, any answer will be correct if the logic program P consists of inconsistent logic statements. But the user might get one of the answers because of deduction strategy. Thus user forces his program to produce some answer. QCL on the other hand has logic protection capability to detect inconsistent or non-terminating statements in logic programs. In addition, it stores vital verification information during the checking.

S is a **term** if it is a variable, a sequence of a function symbol of n arguments followed by n terms, or a sequence of a universal quantifier ALL or existential quantifier EX of m+1 arguments followed by a variable and m terms. A variable is **free** in the term if at least an occurrence of it is not within the scope of a quantifier employing the variable. Note that our definition of term is not standard in classical first order logic. QCL intends to blur the distinction between the predicates and functions. And a term t **governs** an occurrence of term s if either there is a subterm (IF t p q) and the occurrence of s is in p, or (IF t' p q) and the occurrence of s in q, where t is (NOT t'). (ALL_LIST (x_1 ... x_n) p) is an abbreviation for (ALL x_1(ALL x_2(... (ALL x_n p)))), (EX_LIST(x_1 ... x_n) p) for (EX x_1(EX x_2(...(EX x_n p)))) and (ALL_EX$_i$ (x_1 ... x_n) p) for a specific n mixed quantifiers over p, its negated form (EX_ALL$_i$(x_1 ... x_n)(NOT p)).

Formally speaking, a safe definition principle is:
(DEFLQ (EQUAL(f x_1 ... x_n) body)), where

(A) f is a new function symbol of n arguments.

(B) $x_1, ..., x_n$ are distinct variables.

(C) body is a term and only mentions free variable symbols in $x_1 ... x_n$.

(D) There is a well-founded relation r and a measure function m of n arguments.

(E) For each occurrence of a subterm of the form (f y_1 ... y_n) in the body, the governing terms t_1 ... t_k and the governing variables which are the free variables z_1 ... z_j in the governing terms and (f y_1 ... y_n) but not in x_1 ... x_n, it is a theorem that

```
(ALL_LIST (x1 ... xn)(ALL_LIST (z1 ... zj)
              (IMPLIES(AND t1 ... tk)
                      (r (m y1 ... yn)(m x1 ... xn))))).
```

The definition principle is shown by proving that the axiom constitutes a recursive definition of some concept. An axiom of the form:(f x_1 ... x_n)==body can be shown to be recursive if according to some measure M the complexity of the arguments of any occurrence of f in the body, assuming the hypotheses governing f in the body, is less than the complexity of x_1 ... x_n.

Some examples:

```
(DEFLQ                     (DEFLQ
(EQUAL(AND X Y)              (EQUAL(NOT X)
    (IF X                   (IF X
      Y                       NIL
      NIL)))                  T))
```

NIL is considered as false and T denotes true. The function IF is a primitive operator. Informally speaking, if X is NIL, then (IF X Y Z) is equal to Z. and if X is not NIL, then (IF X Y Z) is equal to Y.

```
(DEFLQ                        (DEFLQ
(EQUAL(ADJ X Y)                  (EQUAL(ADJDL X Z)
     (IF(LISTP X)                 (IF(EQUAL X Z)
       (IF(ADJW(CAR X))                T
         (EQUAL Y(CDR X))              (EX Y(IF(ADJ X Y)
         NIL)                         (ADJDL Y Z)
       NIL)))                          NIL)))))
```

The (ADJDL X Z) says that two things X and Y make an adjective difference list if two things are identical or there is an adjective (ADJ X Y) followed by an adjective difference list (ADJDL Y Z). The defining equation for ADJDL is added to our theory by the following instantiation of definition principle: f is the function symbol ADJDL; n is 2; x_1 is X and x_2 is Z; body is the term (IF(EQUAL X Z) T(EX Y(IF(ADJ X Y)(ADJDL Y Z) NIL))); r is PLESSP; m is the function symbol LENGTH1, where (LENGTH1 X Y) is defined to be (LENGTH X); governing terms are (NOT(EQUAL X Z)) and (ADJ X Y); a governing variable is Y. One theorem required by (E) is: (ALL X(ALL Z(ALL Y(IMPLIES(AND(NOT(EQUAL X Z))(ADJ X Y)) (PLESSP(LENGTH1 Y Z)(LENGTH1 X Z))))))). The above theorem can be proved from definitions ADJ and LENGTH1.

```
(DEFLQ
(EQUAL(DIFFL X Z A)
    (IF(EQUAL X Z)
      (EQUAL A NIL)
      (IF(LISTP X)
        (EX B(IF (EQUAL A(CONS(CAR X) B))
               (DIFFL (CDR X) Z B)
               NIL))
          NIL)))
)
```

(DIFFL X Z A) follows the recursive definition principle because (ALL X(ALL Z(ALL A(ALL B(IMPLIES (AND(NOT(EQUAL X Z))(LISTP X)) (PLESSP(LENGTH2 (CDR X) Z B)(LENGTH2 X Z A))))))) is a theorem. (LENGTH2 X Y Z) is defined to be (LENGTH X).

```
(DEFLQ                        (DEFLQ
 (EQUAL(ADJL A)                  (EQUAL(ADJDIFFL X Z)
     (IF(LISTP A)                  (EX A(IF(DIFFL X Z A)
       (IF(ADJW(CAR A))                  (ADJL A)
         (ADJL(CDR A))                 NIL))))
         NIL)
       (EQUAL A NIL))))
```

(ADJL A) is recursive because (ALL A(IMPLIES(LISTP A)(PLESSP(LENGTH (CDR A))(LENGTH A)) is a theorem.

The principle of definition for this advanced programming language extends previous research[Boyer&Moore 79] in the following ways: The first way is that <u>existential and universal quantifiers</u> are allowed in the body of the defining equation. The second way is that the measured arguments of the recursive calls of a function are not necessarily some function of the formal parameters, but may simply be related to them by an <u>arbitrary relation</u>. For example, ADJDL is recursive because the ADJ relation makes Y smaller than X. In this case there is no obvious selector function car or cdr which, when applied to (ADJDL X Z), results in (ADJDL Y Z).

Theorem D There is a unique total function satisfying the recursive defining equation (f x_1 ... x_n)==body.

Proof: The details are shown in [Brown&Liu 84].

Thus, system guarantees consistency and termination for each new defined function.

C. Relationship to Horn Clauses

In Prolog and some other logic programming languages, Horn clauses are the main type of logic statements. Based on the closed world assumption[Reiter 78], QCL can easily translate these clauses into the quantified IF-form formulas. Thus Horn clauses are essentially a subset of QCL. For example, the following Horn clause style definition of ADJL is automatically translated to the above definition of ADJL.

```
(ADJL NIL) <-
(ADJL (CONS A1 A)) <- (ADJW A1)(ADJL A).
```

II. FEATURES OF QCL PROGRAMMING SYSTEM

A. Combining Programming with Verification System

QCL mixes programming and verification system in the natural manner. The output form of the command (PV r), which might **compute** or **prove** relations and functions, should be equivalent to the original expression. The meaning of (PV r)==output_form is (ALL_LIST (x_1 ... x_n) r==output_form) if r has n free variables. Otherwise, r== output_form.

The command (PV r) asks the system to generate any n-tuples satisfying the relationship r, where r contains n free variables. The command (PV r) may also be used to ask system to make deduction from the relation in the case that r has no free variables. The command simply tries to prove r. Usually, the output form is true, false, its equivalent but simplified form or a conjunctive form of some equalities and inequalities about remaining arguments and their binding values.

The deduction techniques of PV are based on the fundamental deduction principle, which makes it possible that QCL logic programming interpreter is the same as its verification system interpreter.

B. Explanation Capability

It is worthwhile embedding an explanation capability inside an automated reasoning system. Currently, the system can produce all tracing information whenever a rule, axiom, or definition is applied. This information consists of three parts: an input expression I which is being evaluated, an intermediate expression M and a name of rule which produces M from 1, an output expression O obtained by recursively evaluating the M expressions. Thus a trace will generally be of the form:

```
Il:exp
   by use of: a name of rule, axiom or definition.
MI :exp
  I2:exp
    by use of: ...
  M2:exp

  02:exp
Ol:exp
```

where the number immediately following 1, M, or O is the level at which the application of an rule, axiom, or definition takes place. At a given level number i, Oi and Mi are always associated with the preceding Ii.

The debugging principle is this: if Ii does not equal to Mi, then the definition, axiom or rule used in that application is incorrect, if Mi does not equal Oi then one must recursively examine the i+l level(ie. 1 M_{i+1} O_{i+1}) to find the error.

C. Negation Is Not A Failure

The negation-as-failure rule is an operational connection between negative and positive terms. The soundness and completeness of a restricted form has been shown in [Clark 78j[Jaffar 83]. There are still some fundamental limitations on that rule if there is no logical connection between them.

The following program uses negation-as-failure to define a term (NDIFFL X Y Z), which is supposed to be equivalent to (NOT(DIFFL X Y Z)). The cut symbol -/■ makes a commitment for all subgoals since the parent goal. Hence any attempt to resatisfy any previous subgoal will fail. NIL is a predicate defined in such a way that as a goal it always fails and causes backtracking to take place. But when NIL is encountered after cut, the normal backtracking behavior will be altered by the cut and will cause the effect that whenever (DIFFL X Y Z) is successful, (NDIFFL X Y Z) is failed.

```
(DIFFL XX NIL) <-
(DIFFL(CONS XI X)Z(CONS XI A))<-(DIFFL X Z A)
(NDIFFL X Y Z)<-(DIFFL X Y Z)/NIL
(NDIFFL XYZ)<-
```

A correct answer can be deduced by failing with NIL or rather false on the query (NDIFFL '(LARGE MANY) '(MANY) '(LARGE))). But it also returns a failure on the following question which is true for any Z not equal to MANY: (NDIFFL '(LARGE MANY) Z '(LARGE)). This relation involves a variable Z in the NDIFFL. In general, the rule is not complete and has to restrict its queries and either deduction strategy or program statements in order to work[Clark 78]. The negative query is often restricted to the ground term. Thus, much negative information simply can't be queried.

Even worse, it should be noted that the above set of clauses are inconsistent if (NDIFFL X Y Z) really is (NOT(DIFFL X Y Z)). QCL takes a logically correct approach towards handling negative information. Any negative term (NOT tm) is defined as the term (IF tm NIL T) by system. If DIFFL is defined as before, (PV (NOT(I)IFFL'(LARGE MANY) Z '(LARGE))) will result (IF (EQUAL Z '(MANY)) NIL T). This means that answer is any Z which is not equal to '(MANY). The details of computing this relation are given as follows:

```
Example A:
(PV (NOT(DIFFL '(LARGE MANY) Z '(LARGE))))
The expression to be recursively simplified is:
(NOT (DIFFL (QUOTE (LARGE MANY))
        Z
        (QUOTE (LARGE))))
  Il:(DIFFL (QUOTE (LARGE MANY))
        Z
        (QUOTE (LARGE)))
    by use of: DIFFL
  MI:(IF (EQUAL (QUOTE (LARGE MANY))
           Z)
       (EQUAL (QUOTE (LARGE))
           NIL)
       (IF (LISTP (QUOTE (LARGE MANY)))
          (EX *1 (IF (EQUAL (QUOTE (LARGE))
                  (CONS (CAR (QUOTE (LARGE MANY)))
                        *1))
                  (DIFFL (CDR (QUOTE (LARGE MANY)))
                      Z*l)
                  NIL))
          NIL))
   I2:(DIFFL (QUOTE (MANY))
         Z NIL)
     by use of: DIFFL
   M2:(IF (EQUAL (QUOTE (MANY))
            Z)
        (EQUAL NIL NIL)
        (IF (LISTP (QUOTE (MANY)))
           (EX *2 (IF (EQUAL NIL
                   (CONS (CAR (QUOTE (MANY)))
                        *2))
                   (DIFFL (CDR (QUOTE (MANY)))
                       Z*2)
                   NIL))
           NIL))
   02:(EQUAL (QUOTE (MANY))
         Z)
  01:(EQUAL (QUOTE (MANY))
         Z)

The result of recursive simplification is:
(IF (EQUAL (QUOTE (MANY))
        Z)
   NIL T)
end of deduction
```

Input 12 is due to variable *1 in Ml being replaced by NIL under the if-condition (EQUAL '(LARGE)(CONS(CAK '(LARGE MANY)) *1)).

Example B:
(PV (DIFFL '(LARGE MANY) Z '(LARGE)))
The expression to be recursively simplified is:

```
(DIFFL (QUOTE (LARGE MANY))
       Z
       (QUOTE (LARGE)))
  11:(D1FFL (QUOTE (LARGE MANY))
            Z
            (QUOTE (LARGE)))
    by use of: DIFFL
  MI:(IF (EQUAL (QUOTE (LARGE MANY))
                Z)
         (EQUAL (QUOTE (LARGE))
                NIL)
         (IF (LISTP (QUOTE (LARGE MANY)))
             (EX *I (IF (EQUAL (QUOTE (LARGE))
                        (CONS (CAR (QUOTE (LARGE MANY')))
                              *!))
                        (DIFFL (CDR (QUOTE (LARGE MANY)))
                               Z *I)
                        NIL))
             NIL))
  I2:(DIFFL (QUOTE (MANY))
            Z NIL)
     by use of: DIFFL
  M2:(IF (EQUAL (QUOTE (MANY))
                Z)
         (EQUAL NIL NIL)
         (IF (LISTP (QUOTE (MANY)))
             (EX *2 (IF (EQUAL NIL
                        (CONS (CAR (QUOTE (MANY)))
                              *2))
                        (DIFFL (CDR (QUOTE (MANY)))
                               Z *2)
                        NIL))
             NIL))
  02:(EQUAL (QUOTE (MANY))
            Z)
  OI:(EQUAL (QUOTE (MANY))
            Z)
```

The result of recursive simplification is:

```
(EQUAL (QUOTE (MANY))
       Z)
```

end of deduction

D. Relational vs. Functional

There are currently two really good ways of programming based on formal logic, namely: (I) programs based on recursive functions, such as the LISP. (2) programs based on NON-NEGATIVE recursive relations, such as HCPRVR and PROLOG.

QCL takes a different approach from LOGLISP[Robinson 82], QUTE[Sato 83] and TABLOG[Malachi 84]towards combining these two formalisms. Our concern focuses on the unique formalism and a smooth interaction between relations and functions.

Each type of programming system, of course, has many additional features (eg. assignment statements, goto's etc.), but it is the logical features: recursive functions or recursive relations which make it so easy to express, understand, and debug programs written in these systems. Some programs arc more easily expressed, understood, and debugged as functions than relations and vice-versa. In particular combinations of deterministic programs returning unique outputs should be written as functions. For example, to APPEND the REVERSE of a list A onto the result of APPENDing the REVERSE of a list B onto the REVERSE of a list C is written as functions as: (APPEND (REVERSE A) (APPEND (REVERSE B) (REVERSE C))) whereas it is rewritten as relations with many extra symbols as:

```
(EX XI(EX X2(EX X3(EX X4(AND(REVERSE A XI)
                            (REVERSE B X2)
                            (REVERSE C X3)
                            (APPEND X2 X3 X4)
                        (APPEND XI X4 ANSW))))))
```

XI,X2,and X3 are respectively the results of reversing A, B, and C, X4 is the result of appending X2 to X3, and the answer: ANSW is the result of appending XI to X4. In such n case the lack of nesting in relational notation makes it resemble assembler notation (where the variables are registers and the assembler operations are REVERSE and APPEND) rather than a high level language.

On the other hand, combinations of non-deterministic programs are more naturally expressed as relations. For example, a program which parses English(and translates it into another language) is naturally made up of a number of non-deterministic programs. The definition of a declarative tran sitive sentence is easily written in relational notation as:

```
(EX XI(EX X2(EX X3(AND(NP 11 12 XI)
                      (VG 12 13 X2)
                      (NP 13 14 X3)
                      (COMBINE XI X2 X3 ANSW)))))
```

where It is the input text, 12 is the rest of the text after a noun phrase is parsed, 13 is the rest of the text of 12 after a verb group is parsed, 14 is any remaining unparsed text, XI is the translation of the first noun phrase, X2 is the translation of the verb group, X3 is the translation of the second noun phrase, and ANSW is the translation of the entire sentence.

Such a program can not be rewritten in functional notation as: (COMBINE(NP 11 I2)(VG 12 I3)(NP 13 14)) because the NP and VG relations are not deterministic on their third arguments. For example, there may be many different parses of the the first noun phrase each giving a different answer. Furthermore, 12 is another non-deterministic output of (NP 11 12) which must be passed an input to (VG 12 13). Likewise, if insisting on functional notation, an assembly-like program can not be avoided.

These considerations also apply to advanced database technology. Data can be represented relationally as in relational databases [Codd 72] or functionally. For example, a request for the salary of the president of the university of the largest state in the united states could be written as:

```
(EX XI(EX X2(EX X3
(AND(LARGEST STATE OF(UNITED STATES) XI)
    (UNIVERSITY OF XI X2)
    (PRESIDENT OF X2 X3)
    (SALARY OF X3 ANSW)))))
```

However, it can be written in the simpler functional notation as: (SALARY OF (PRESIDENT OF (UNIVERSITY OF (LARGEST STATE OF (UNITED STATES))))) because each of these predicates is deterministic on its last argument. On the other hand, in an example with non-deterministic outputs, the relational representation is better.

Two distinct programming systems have been developer! based on formal logical notationfrecursive functions, recursive relations) and each is especially useful for representing a particular type of programs(deterministic and non-deterministic). Furthermore, the more facilities of formal logic that are provided in a programming system, the less one needs to use constructs such as GOTO and assignment which are harder to use, understand and debug. Therefore, it follows that a significantly better programming system could be achieved by amalgamating these forma' notations into a single formal system provided that effici .. interpreted and/or compiled inference systems could be designed.

E. Specification VS. Program

In QCL, specifications and programs are considered as different styles to express logic expressions for their own purposes within the same framework. Specifications are descriptively useful logic forms and their computationally useful forms are programs.

We believe that automatic derivation, verification and synthesis of logic programs are made easier, provided that specifications and programs have a smooth interaction. Two important relationships between logic programs and their specifications are correctness and completeness. Correctness is a property that for any n-tuple x, if x is computed as a solution satisfying a relation R, then x belongs to the relation specified by S. It can be formalized as Vx(S|— R(x)<=P|— R(x)). Completeness is a property that for all members of the specified relation R are computable from program P. It can be represented as Vx(S|— R(x)=>P—R(x)). One way to show these relationships is to use meta-level deduction to prove those formulas [Brown 78]. Another way is that, if finding a good specification S such that S—P or P|— S, we get same correctness proof or completeness proof by exploiting the transitivity of logical implication |—.

The second approach is the cornerstone of logic program derivation, verification and synthesis investigated in[Clark 77] [Hogger 81]. One advantage in this approach towards verification is that the object-level deduction from S to P or from P to S is sufficient to show their relationships without meta-level deduction and concerning goal relation R(x). Another advantage is that for proving any property of P, it-may be much easier to show that S has that property and P is complete with respect to S. In the Hogger study, specifications expressed in "if-and-only-if" and "first-order recursive" styles are crucial in this approach. Since QCL intend to bring specifications and programs together, it provides a good opportunity to investigate good specifications and program styles in order to automate the derivation, synthesis, and verification of logic programs. For example,

```
(DEFLQ
(EQI)AL(PICK.S U V Z)
    (IF(MEMBER.S U Z)
      (IF(MEMBER.S V Z)
        (PLESSP U V)
        NIL)
      NIL))
)
```

Informally stated, (P1CK.S U V Z)-(MEMBER.S U Z)A (MEMBER.S V Z)A(PLESSP U V).

```
(DEFLQ
(EQUAL(MEMBER.S U L)
(EX X(EX Y(IF(EQUAL L (CONS X Y)))
          (IF(EQUAL U X)
             T
             (MEMBER.S U Y))
          NIL))))
```

Informally, (MEMBER.S U L)-3x3y(L=(CONS x y)A (U=xV(MEMBER.S U y))).

The specification S:{PICK.S MEMBER.S} is for the following program P:{PICK.P MEMBER.P}. It said to pick any two numbers U and V from a list Z of numbers, not necessarily distinct, such that U is less than V. The relationships can be shown by object-level deduction, that is, *S*\— P and P|~S.

```
(DEFLQ
(EQUALfPICK.P U V Z)
    (IF(LISTP Z))
      (IFfEQUAL U(CAR Z))
        (IF(MEMBER.P V (CDR Z))
          (PLESSP U V)
          NIL)
        (IF(EQUAL V(CAR Z))
          (IF(MEMBER.P U (CDR Z))
            (PLESSP U V)
            NIL)
          (PICK U V(CDDR Z))))
      NIL)
)
(DEFLQ
(EQUAL(MEMBER.P X L)
    (IF(L1STP L)
      (IFfEQUAL X (CAR L))
          T
        (MEMBER.P X (CDR L)))
      NIL))
)
```

IV. QCL VERIFICATION

A. A Generalised Noetherian Induction Principle

In logic programming verification, we often need induction to rearrange the conjecture into a systematic way to prove. The induction principle we use is a generalized version of Noetherian induction[Burstall 69][Boyer&Moore 79]. Existential and universal quantifiers are allowed both in the base case and in the induction steps.

Formally stated, the induction principle is:

Suppose:

(A) p is a term with t's distinct free variables x_1, $\dots,x_n,x_{n+1},\dots,x_t$;
(B) r is a well-founded relation;
(C) m is a measure function of n arguments;
(D) $b_1, \dots, b_k$ are non-negative integers;
(E) For each i $1\leq i\leq k$, $z_{i,1}, \dots, z_{i,b_i}$ are distinct bound variables;
(F) $q_1, \dots, q_k$ are terms;
(G) $h_1, \dots, h_k$ are positive integers; and
(H) For $1\leq i\leq k$ and $1\leq j\leq h_i$, $s_{i,j}$ is a substitution and it is a theorem that

(ALL_LIST ($x_1 \dots x_t$)(ALL_LIST($z_{i,1} \dots z_{i,b_i}$)
  (IMPLIES q_i
    (r(m $x_1 \dots x_n$)/$s_{i,j}$(m $x_1 \dots x_n$))))).

Then

(I). (ALL_LIST (x_1 ... x_t) p) is a theorem if

for the base case,
(ALL_LIST(x_1 ... x_t)
(IMPLIES(AND(NOT(ALL_EX$_1$($z_{1,1}$ ... z_{1,b_1})q_1))
...,
(NOT(ALL_EX$_k$($z_{k,1}$ ... z_{k,b_k})q_k)))
p))
is a theorem **and**

for each $1 \leq i \leq k$ induction step,
(ALL_LIST(x_1 ... x_t)
(IMPLIES (ALL_EX$_i$($z_{i,1}$ ... z_{i,h_i})
(AND q_i p/$s_{i,1}$ ... p/s_{i,h_i}))
p))
is a theorem.

(II). (EX_LIST (x_1 ... x_t) p) is a theorem if

for the base case,
(EX_LIST(x_1 ... x_t)
(AND(AND(NOT(ALL_EX$_1$($z_{1,1}$ ... z_{1,b_1})q_1))
...,
(NOT(ALL_EX$_k$($z_{k,1}$ ... z_{k,b_k})q_k)))
p))
is a theorem **or**

for some $1 \leq i \leq k$ induction step,
(EX_LIST(x_1 ... x_t)
(AND (ALL_EX$_i$($z_{i,1}$ ... z_{i,b_i})
(AND q_i (NOT p)/$s_{i,1}$ ...(NOT p)/s_{i,h_i}))
p))
is a theorem.

We adopted a Boyer&Moore's definition here before proving soundness of this induction principle. <X1 ... Xn> is *RM-smaller* than <Y1 ... Yn> if there is a measure function M and a well-founded relation R such that (R (M X1 ... Xn)(M Y1 ... Yn)).

Theorem I. Soundness for Universal Quantifiers Case

Proof:

Supposed there is a t-tuples <X1, ...,Xt> such that (P X1 ... Xt)=NIL where (P X1 ... Xt) stands for p/s, s is substitution {<x1, X1> ... <xt, Xt>}. Let it be a RM-minimal such t-tuples.

Case I. Suppose
(NOT(ALL_EX$_1$($z_{1,1}$ ... z_{1,b_1})(Q1 X1 ... Xt)))≠NIL
...,
and
(NOT(ALL_EX$_k$($z_{k,1}$ ... z_{k,b_k})(Qk X1 ... Xt)))≠NIL,
where (Qi X1 ... Xt) is abbreviated for q_i/s, and s is a substitution {<x1, X1> ... <xt, Xt>}.

Then by base case, (P X1 ... Xt)≠NIL contradicting the assumption that (P X1 ... Xt)=NIL.

Case II. Suppose

Some i (NOT(ALL_EX$_i$($z_{i,1}$ ... z_{i,b_i})(Qi X1 ... Xt)))=NIL,
where (Qi X1 ... Xt) abbreviated for q_i/s, s is a substitution {<x1, X1> ...<xt, Xt>}.

Then (ALL_EX$_i$($z_{i,1}$ ... z_{i,b_i})(Qi X1 ... Xt))≠NIL.

Since <X1, ...,Xt> is an RM-minimal t-tuples such that (P X1 ... Xt)=NIL and condition (II), we have (ALL_EX$_i$($z_{i,1}$ ... z_{i,b_i}) (AND(Qi X1 ... Xt) (P $d_{i,1,1}$ .... $d_{i,1,t}$) ... (P $d_{i,h_i,1}$ ... $d_{i,h_i,t}$)))≠NIL, where for each $1 \leq v \leq t$, the substitution s_{i,h_i} replaces Xv with some $d_{i,h_i,v}$. But we know the ith induction step is a theorem. Hence, we get (P X1 ... Xt)≠NIL, contradicting the assumption that (P X1 ... Xt)=NIL.

Q.E.D.

Theorem II. Soundness for Existential Quantifiers Case

Proof: Theorem II is only a symmetric case of Theorem I. Its-validity is obvious after proving Theorem I.

Q.E.D.

An application of this induction principle is illustrated below. In the proof (PV (ALL X(ALL Z(EQUAL(AI)JI)L X Z)(EX A(AND(DIFFL X Z A) (ADJ A))))))), an induction is obtained by instantiation of the principle as follows: p is (EQUAL(ADJDL X Z) (EX A(AND(DIFFL X Z A)(ADJL A)))); r is PLESSP; in is LENGTH; n is 2; k is 1; b_1 is 1; $Z_{1.1}$ is Y; h_1 is 1; q_1, is the term (AND (NC)T(EQUAL X Z))(AI)J X Y)); the theorem required by (II) is: (ALL X(ALL Z(ALL Y(IMPLIES(AND (NOT(EQUAL X Z))(APJ X Y))(PLESSP(LENCTH1 Y Z)(LENGTIII X Z))))))). The induction is:

```
Base case :
(ALL X(ALL Z(IMPLIES(OR(EQUAL X Z)
                        (NOT (EX Y(ADJ X Y))))
                    (PXZ))))
Induction step:
(ALL X(ALL Z(IMPLIES(AND(NOT(EQUAL X Z))
                        (EX Y(AND(ADJ X Y)
                                 (P Y Z))))
                    (P X Z))))
```

B. Induction Schemes and A Proof Example

The recursive firsts-order function definition suggests plausible induction schemes. Thus, the induction schemes for each of these recursive functions can now be produced. Suppose we try to prove two notions of adjective different list art-equivalent, (PV (ALL X(ALL Z(EQUAL(ADJDL X Z)(ADJDIFFL X Z)))).

The mechanical proof of this conjecture begins by unraveling the non-recursive function ADJDIFFL and then examining the induction schemes for each recursive function.

(ALL X(ALL Z(EQUAL(ADJDL X Z) (EX A(AND(DIFFL X Z A)(ADJL A)))))) (by opening up the function ADJDIFFL)

The system examines the induction scheme for each recursive function.

The scheme for (ADJL A) is:

```
(EQUALfALL A(p A))
(AND(ALL A(IMPLIES(NOT(LISTP A)(p A)))
   (ALL A(IMPLIES(AND(LISTP A)(p (CDR A)))(p A)))))
```

The scheme for (DIFFL X Z A) is:

```
(EQUAL(ALL XfALL Z(ALL A(p X Z A))))
(ANI)(ALL X(ALL Z(ALL A(IMPLIES(OR(NOT(LISTP X))
                      (NOT(EX BfEQUAL A(CONS(CAR X) B)))))
                         (P X Z A)))))
   (ALL X(ALL Z(ALL A(IMPLIES(AND(LISTP X)
                     (EX B(AND(EQUAL A(CONS(CAR X) B))
                           (p (CDR X) Z B))))
                         (P X Z A)))))))
```

The scheme for (ADJDL X Z) is:

```
(EQUAL(ALL X(ALL Z(p X Z)))
(ANDfALL X(ALL Z(IMPLIES(OR(EQUAL X Z)
                        (NOT(EX Y(ADJ X Y))))
                     (P X Z))))
   (ALL X(ALL Z(IMPLIES(AND(NOT(EQUAL X Z))
                      (EX Y(AND(ADJ X Y)(p Y Z)))
                     (P X Z)))))))
```

The induction scheme for (ADJL A) is ignored since it has a measured argument which is a bound variable. The induction scheme for (DIFFL X Z A) is subsumed by the induction scheme for (ADJDL X Z); so the ADJDL induction scheme is used. The system uses heuristics and follows the induction principle to split the original conjecture into two cases as above. The details of the mechanical proof are generated in [Brown&Liu 84].

V, CONCLUSIONS

A new logic programming language has been developed which is based on quantifier elimination techniques and axiomatization. Unlike other logic program systems, it does not involve any unification algorithm. This system handles a number of problems better than unification-based logic programming system. For example, this system allows the proper axiomatization of negation, a smooth interaction between functions and relations, and an ability to write specifications and to verify the correctness of programs using a generalized Noetherian induction rule. Also, it extends previous work in program verification to the problem verifying recursively defined relations whose definition bodies contain quantifiers.

ACKNOWLEDGEMENTS

The work described here has been supported in part by the Army Research Office under Grant DAAG 29-83-K-0103 and in part by the National Science Foundation under Grant DCR-8320340.

The authors would like to thank Prof. Norman Martin and Ruey-Juin Ghang for their helpful comments to earlier drafts of this paper. Our thanks also go to the referees for their useful suggestions. Finally, we thank Ching-Hua Ghow and Chin-Laung Lei for their tips on using text editor.

REFERENCES

1. Boyer, R.S., and J S. Moore, *A Computational Logic,* New York, Academic Press, 1979.
2. Brown, F.M., "Semantic theory for Logic Programming", *Coloquia Mathematica Society Janos Rolyai, 26 Mathematical Logic in Computer Science ,* Salgotarjan, Hungary, 1978.
3. Brown, F.M., "Experimental Logic and the Automatic Analysis of Algorithms", *Proceedings of the Army Conference on Application of AI to Battlefield Information Management,* Maryland, 1983, pp 217-281. (To appear AI Journal).
4. Brown, F.M., and P. Liu, "Foundations of QOL Programming System" The University of Texas at Austin, Computer Science Department, Technical Report, May 1984.
5. Burstall, R. M., "Proving Properties of Programs by Structural Induction", *Computer Journal,* vol. 12, no. 1, February 1969.
6. Clark, K.L., "Negation as Failure" in *Logic and Database*,* eds. Gallarire, H.J. and J. Minker, Plenum Press, 1978.
7. Clark, K.L. and S. Stickel "Predicate Logic: A Calculus for Deriving Programs", *UCA1-77.*
8. Codd, E.F. "Relational Completeness of Data Base Sublanguages", *Data Base Systems* ed. R. Rustin, Prentice-Hall, 1972.
9. Hogger, C.J. "Derivation of Logic Programs", *J ACM* Vol.28, No.2, April 1981.
10. Jaffar, J. Lassez, and J. Lloyd, "Completeness of the Negation as Failure Rule", *IJCA1-88,* pp 500-506.
11. Malachi, Y., Manna, Z., and Waldinger, R., "TABLOG: The Deductive-Tableau Programming Language", *Proceedings of 1984 Lisp and Functional programming Conference,* Austin, Texas, pp 323-330.
12. Reiter, R., "On Closed World Data Bases" in *Logic and Databases,* eds. Gallarire, H.J. and J. Minker, Plenum Press, 1978.
13. Robinson, J.A. and E.E. Sibert, "LOGLISP: An Alternative to Prolog", *Machine Intelligence 10,* 1982.
14. Sato, M. and T. Sakurai, "QUTE: A Prolog/Lisp Type Language for Logic Programming", *IJCAI-88,* pp 508-513.