LOGIC PROGRAMMING

Logic programming is programming by description. The programmer describes the application area and lets the program choose specific operations. Logic programs are easier to create and enable machines to explain their results and actions.

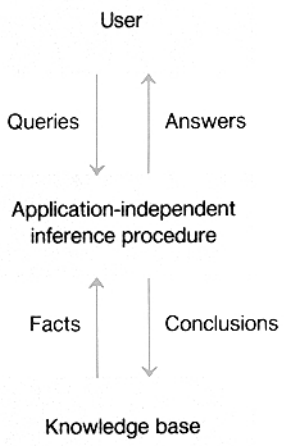
MICHAEL R. GENESERETH and MATTHEW L. GINSBERG

The key idea underlying logic programming is *programming by description.* In traditional software engineering, one builds a program by specifying the operations to be performed in solving a problem, that is, by saying *how* the problem is to be solved. The assumptions on which the program is based are usually left implicit. In logic programming, one constructs a program by describing its application area, that is, by saying *what* is true. The assumptions are explicit, but the choice of operations is implicit.

A description of this sort becomes a program when it is combined with an application-independent inference procedure. Applying such a procedure to a description of an application area makes it possible for a machine to draw conclusions about the application area and to answer questions even though these answers are not explicitly recorded in the description. This capability is the basis for the technology of logic programming. Figure 1 illustrates the configuration of a typical logic programming system. At the heart of the program is an application-independent inference procedure, which accepts queries from users, accesses the facts in its knowledge base (the description), and draws appropriate conclusions. It is thus able to answer users' questions and, in some cases, to record its conclusions in its knowledge base.

Because the inference procedure used by a logic program is independent of the knowledge base it accesses, program development amounts to the development of an appropriate knowledge base—finding a suitable description of the application area. There are several advantages to this. Chief among these is incremental development. As new information about an application area is discovered (or perhaps just discovered to be important to the problem the program is designed to solve), that information can be added to the program's knowledge base and so incorporated into the program itself. There is no need for algorithm development or revision.

A second advantage is explanation. With the piecemeal nature of automated reasoning, it is easy to save a record of the steps taken in solving a problem. By presenting this record to the user, it is possible for a program to explain how it solves each problem and, therefore, why it believes the result to be correct. Explana-



An application-independent inference procedure is at the center of any logic program. When interrogated by the user, the inference procedure replies after drawing conclusions from the facts in the knowledge base. In some instances, the conclusions drawn are stored in the knowledge base for later use.

FIGURE 1. A Logic Programming System

tions of this sort are especially valuable to programmers for debugging logic programs.

DESCRIPTION

The process of describing an application area begins with a conceptualization of the objects presumed or hypothesized to exist in the application area and the relations satisfied by those objects. Facts about these objects and relations as sentences are then expressed in a suitable formal language.

Our notion of object is quite broad: They can be concrete (e.g., a specific circuit, a specific person, this paper) or abstract (e.g., the number 2, the set of all integers, the concept of justice); they can be primitive or composite (e.g., a circuit that consists of many components); they can even be fictional (e.g., a unicorn, Peer Gynt, Miss Right).

A relation is a quality or attribute of a group of objects with respect to each other. Of course, a single relation may hold for more than one group of objects (e.g., the "father of" relation holds for many different pairs of individuals), and a single group of objects may satisfy more than one relation (e.g., one individual may be the "father of" another and also a "neighbor").

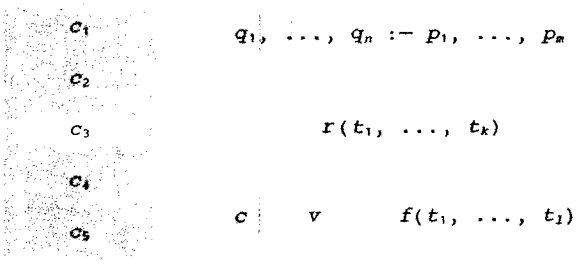
The essential characteristic of a logic programming language is *declarative semantics*. There must be a simple way of determining the truth or falsity of each statement, given an interpretation of the symbols of the language. The meaning of each statement must be independent of the intended use of that statement in solving any particular problem.

Most logic programming systems use *clausal form*, which is a variant of the predicate calculus (see Figure 2). Clausal form has the requisite declarative semantics, although it is not the only language with this property. Declarative semantics can also be defined for such disparate languages as tables, graphs, and charts. Clausal form is also extremely expressive in that the existence of logical operators and variables makes it possible to encode partial information about an application area. In more restricted languages such as frames, procedures must be written to encode partial information.

The simplest kind of expression in clausal form is the *atom*. An atom states that a specific relation holds for a specific group of objects. For example, one can express the fact that Art is the father of Bob by taking a relation symbol like `F` and object symbols `Art` and `Bob` and combining them:

```
F(Art,Bob).
```

More complex facts can be written using the operator :—. For instance, q :— p indicates that q is true if p is true; in other words, p implies q. The "reverse" implication operator is often used in logic programming languages in place of the usual "forward" operator ⇒, to



A *logic program* is an arbitrary set of expressions known as clauses. A *clause* is an expression of the form shown on the upper right. A sentence of this form says that one of the q_i must be true if all of the p_i are true. The expressions to the left and right of the :— operator in a clause must be *atoms*, that is, expressions of the form shown on the middle right, where r is a relation constant and where each t_i is a term. A *term* is either an object constant, a variable, or an expression of the form shown on the lower right, where f is a function constant and each t_i is a term. A *constant* is a contiguous sequence of lowercase characters and numbers, and a *variable* is a contiguous sequence of uppercase characters and digits beginning with an uppercase character.

FIGURE 2. The General Form of a Logic Program

call attention to the conclusion of the implication. The following sentence states that Art is Bob's parent if Art is the father of Bob:

```
P(Art,Bob) :- F(Art,Bob)
```

If more than one sentence is written to the right of this operator, then the conclusion is guaranteed to be true only if *all* of the conditions are true, that is, if the conditions are conjoined. The following sentence states that Art is Cap's grandparent if Art is Bob's parent and Bob is Cap's parent:

```
G(Art,Cap) :- P(Art,Bob), P(Bob,Cap)
```

If there are no sentences to the right of the operator, then the sentence to the left is true under all conditions. Thus, atoms can be written as clauses in which the right-hand side is empty.

If more than one sentence is written to the left of the implication operator, then *one* of the conclusions is guaranteed to be true, that is, the conclusions are disjoined. The following sentence states that Art is either Bob's mother or Bob's father if Art is Bob's parent:

```
F(Art,Bob), M(Art,Bob) :- P(Art,Bob)
```

Although the conclusions in this case are mutually exclusive and jointly exhaustive, this need not generally be the case.

A clause with no sentences to the left of the operator is a statement that the conditions to the right are inconsistent, that is, that at least one of them is false. Thus, one can express the negation of an atom by writing it as a clause in which the left-hand side is empty.

More general facts can be expressed using variables (lowercase letters) instead of constants to refer to all objects in the programmer's conceptualization of the world. The following sentence states that x is the parent of y if any person x is the father of any person y, that is, that the connection between the "father of" relation and the "parent of" relation is true for everyone:

```
P(x,y) :- F(x,y)
```

A logic program is simply a set of sentences written in this language. The sentences below constitute a logic program for kinship relations. The objects are people in this case, although individual names are not known initially. There are two unary relations: "male" (written `Male`) and "female" (written `Female`); and four binary relations: "father of" (`F`), "mother of" (`M`), "parent of" (`P`), and "grandparent of" (`G`).

```
A1:  P(x,y) :- F(x,y)
A2:  P(x,y) :- M(x,y)
A3:  G(x,z) :- P(x,y), P(y,z)
A4:  Male(x) :- F(x,y)
A5:  Female(x) :- M(x,y)
```

The next set of sentences captures the kinship relationships among six individuals, Art, Amy, Bob, Bud, Cap, and Coe. In some situations detailed information of this sort is available at the outset and can be incorporated into the program itself; in others, it becomes available only at run time.

```
B1:  F(Art,Bob) :-
B2:  F(Art,Bud) :-
B3:  M(Amy,Bob) :-
B4:  M(Amy,Bud) :-
B5:  F(Bob,Cap) :-
B6:  F(Bud,Coe) :-
```

The next example shows how a numerical concept like the factorial function can be defined. The first clause states that the factorial of 0 is 1, and the second clause handles the recursive case. A number n is the factorial of a number k if $k - 1$ is l, the factorial of l is m, and $k * m$ is n.

```
C1:  Fact(0,1) :-
C2:  Fact(k,n) :- k-1=l, Fact(l,m),
                  k*m=n
```

Functions can also be defined on lists. A list in clausal form is an expression of the form $[x_1, x_2, \ldots, x_n]$. This expression can also be written $x_1.[x_2, \ldots, x_n]$. For example, the "append" function on two lists can be defined by the following:

```
D1:  Append([],m,m) :-
D2:  Append(x.l,m,x.n) :- Append(l,m,n)
```

DEDUCTION

Logic is the study of the relationship between beliefs and conclusions. For example, if we believe that Art is the father of Bob and that fathers are parents, then we can conclude that Art is the parent of Bob. The first two sentences *logically imply* the conclusion. In logic programming the programmer encodes a set of beliefs about the application area, and the machine derives conclusions that are logically implied by those beliefs.

Research in mathematical logic and artificial intelligence has led to the development of many application-independent inference procedures. Most of these procedures can be described as the application of one or more *rules of inference* to known facts for the purpose of deriving new conclusions. Subsequent applications to other facts and to the conclusions allow a program to derive further conclusions. And so forth.

Of the rules that have been invented, resolution is the most extensively studied. Given a clause with an atom p on its left-hand side and another clause with an atom p on its right-hand side, it is possible to create a new clause in which the left-hand side is the union of the left-hand sides of the two original clauses with p deleted, and the right-hand side is the union of the right-hand sides of the two clauses with p deleted. In the following example, the expression `P(Art,Bob)` appears on the left-hand side of one rule and on the right-hand side of the other. Consequently, we can apply resolution to these two clauses to produce the conclusion shown. Note that the net effect is to replace the

expression P(Art,Bob) in the second clause with the expression F(Art,Bob).

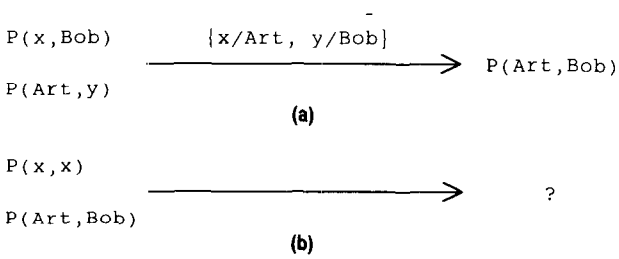
```
Given:
  P(Art,Bob) :- F(Art,Bob)
  G(Art,Cap) :- P(Art,Bob), P(Bob,Cap)
Conclude:
  G(Art,Cap) :- F(Art,Bob), P(Bob,Cap)
```

This example is very simple in that the atoms resolved upon are identical. The resolution rule is somewhat more general in that it can be used even when two atoms are not identical, so long as they can be made to look alike by appropriate instantiations of their variables. The process of finding such instantiations is called *unification* (see Figure 3).

The combination of unification with the resolution rule permits more complicated conclusions to be drawn (see Figure 4). For example, in the following deduction the expression P(x,Bob) in the first clause unifies with the expression P(Art,y) in the second by replacing the variable x with the constant Art and the variable y with the constant Bob. In the conclusion, note that the occurrence of x in F(x,Bob) has been replaced by Art and that the occurrence of y in P(y,z) has been replaced by Bob.

```
Given:
  P(x,Bob) :- F(x,Bob)
  G(Art,z) :- P(Art,y), P(y,z)
Conclude:
  G(Art,z) :- F(Art,Bob), P(Bob,z)
```

The resolution rule can be used as the basis for a number of different inference procedures. Resolution refutation is an especially important case. It is *sound* in that any conclusion it draws is guaranteed to be correct



Unification is the process of finding a set of bindings for the variables in two expressions that makes the two expressions look alike. The expressions in (a) can be unified by the substitution {x/Art, y/Bob}. The expressions in (b) cannot be unified since there is no single binding for the variable x that will make the expressions look alike.

FIGURE 3. Unification

```
P(x,Bob) :- F(x,Bob)

G(Art,z) :- P(Art,y) , P(y,z)

G(Art,z) :- F(Art,Bob), P(Bob,z)
```

Whenever an atom on the left-hand side of one clause unifies with one on the right-hand side of another, a new clause can be deduced, as shown here. The left- and right-hand sides of the new clause are the unions of the left- and right-hand sides of the original clauses, with the unified expressions deleted and the unifying substitution applied to the remaining expressions. In this example, the expression P(x,Bob) on the left-hand side of the first clause unifies with the expression P(Art,y) on the right-hand side of the second clause. This allows us to apply resolution to these two clauses to produce a conclusion. Note that the occurrence of x in F(x,Bob) in the conclusion has been replaced by Art and that y in P(y,z) has been replaced by Bob.

FIGURE 4. Resolution

so long as its premises are correct. It is also *complete* in that it can derive any logical implication from a given set of premises.

In a resolution refutation we assume the negation of the goal we are trying to prove and attempt to derive the *empty clause*, that is, a clause with no conditions and no conclusions. For example, given the preceding kinship facts, we can use resolution refutation to prove that Art is Bob's grandparent. The trace-of-rule application shown below is an example of a *proof* of this conclusion. The labels on the right indicate the clauses resolved to produce the conclusion on that line.

```
E1:  :- G(Art,Cap)
E2:  :- P(Art,y), P(y,Cap)     E1  A3
E3:  :- F(Art,y), P(y,Cap)     E2  A1
E4:  :- P(Bob,Cap)             E3  B2
E5:  :- F(Bob,Cap)             E4  A1
E6:  :-                        E5  B5
```

In this case the final result of the deduction is simply the determination that the goal is logically implied by the facts in the knowledge base. When a goal contains variables, however, we can get further information. For example, suppose that our goal is to determine whether or not there is a person of whom Art is a grandparent. This goal is false if and only if for every person Art is *not* his grandparent. Therefore, the negation of this goal is the clause :- G(Art,z). If we can find a binding for the variable z from which the empty clause can be derived, then we have proved the goal and, furthermore, found an object that makes it true. The upshot of this is that resolution can be used for computing answers other than "true" and "false."

To see how this is done, consider the following resolution proof. In addition to the justification information, this proof contains information about the bindings of all variables, in particular the value Cap for the grandchild variable z.

```
F1:  :- G(Art,z)
F2:  :- P(Art,y), P(y,z)          F1  A3
F3:  :- F(Art,y), P(y,z)          F2  A1
F4:  :- P(Bob,z)  y=Bob           F3  B1
F5:  :- F(Bob,z)  y=Bob           F4  A1
F6:  :-           y=Bob z=Cap     F5  B5
```

A particularly noteworthy feature of this approach to computation is that there can be more than one answer to a question of this sort. Other answers can be determined through other lines of reasoning. In this case, Coe is also one of Art's grandchildren, as demonstrated by the following proof:

```
G1:  :- G(Art,z)
G2:  :- P(Art,y), P(y,z)          G1  A3
G3:  :- F(Art,y), P(y,z)          G2  A1
G4:  :- P(Bud,z)  y=Bud           G3  B2
G5:  :- F(Bud,z)  y=Bud           G4  A1
G6:  :-           y=Bud z=Coe     G5  B6
```

Another noteworthy feature is that we can get answers to other questions by using variables in different positions. For example, by using a variable as the first argument in a grandparent question and a constant as the second argument, we can compute the grandparents of the individual specified as the first argument:

```
H1:  :- G(x,Cap)
H2:  :- P(x,y), P(y,Cap)          H1  A3
H3:  :- F(x,y), P(y,Cap)          H2  A1
H4:  :- P(Bob,Cap)  x=Art y=Bob   H3  B1
H5:  :- F(Bob,Cap)  x=Art y=Bob   H4  A1
H6:  :-             x=Art y=Bob   H5  B5
```

Of course, we can also use variables in all positions and compute all grandparent-grandchildren pairs. The derivation of one such pair would be the following:

```
I1:  :- G(x,z)
I2:  :- P(x,y), P(y,z)                 I1  A3
I3:  :- F(x,y), P(y,z)                 I2  A1
I4:  :- P(Bob,z)  x=Art y=Bob          I3  B1
I5:  :- F(Bob,z)  x=Art y=Bob          I4  A1
I6:  :-           x=Art y=Bob z=Cap    I5  B5
```

Finally, note that nonatomic clauses can be resolved with other nonatomic clauses to produce new clauses that are more efficient to use than original clauses.

```
J1:  GF(x,z) :- G(x,z), Male(x)
J2:  G(x,z)  :- P(x,y), P(y,z)
J3:  F(x,y)  :- P(x,y), Male(x)
J4:  GF(x,z) :- P(x,y), P(y,z), Male(x)   J1  J2
J5:  GF(x,z) :- F(x,y), P(y,z)            J4  J3
```

The lesson to be learned from these examples is that deduction is a very flexible "interpreter" for logic programs. Deduction can be used to compute a single answer to a question, even when there are many answers; it can be used "backwards" to compute answers to other questions; and it can be used on actual logic programs to produce new logic programs that are simpler and/or run more efficiently. Finally, as depicted in Figure 5, the trace of proof can be used as the basis for explanations of the system's conclusions.

CONTROL

Automated reasoning with resolution is a combinatoric process. There are usually several inferences that can be drawn from an initial knowledge base; once an inference has been drawn, new inferential opportunities arise, and so on. In order to achieve efficient computation, it is important for one to choose the best possible opportunity for exploitation at each point in time. This section summarizes several techniques for controlling deduction.

The most straightforward approach is to perform deductions in a fixed but arbitrary order. The PROLOG

```
=> := G(Art,Cap)?
Yes

=> Why?
Z1: G(Art,Cap) :- because
        Z2: P(Art,Bob) :-
        Z3: P(Bob,Cap) :-
        A3:  G(x,y) :- P(x,y), P(y,z)

=> Why Z2?
Z2: P(Art,Bob) :- because
        B1:  F(Art,Bob) :-
        A1:  P(x,y) :- F(x,y)

=> Where R1?
Z2: P(Art,Bob) :- because
        B1:  F(Art,Bob) :-
        A1:  P(x,y) :- F(x,y)

Z3: P(Bob,Cap) :- because
        B5:  F(Bob,Cap) :-
        A1:  P(x,y) :- F(x,y)
```

One of the key advantages of logic programming is that it enables a machine to explain its results and its actions. In this case, the conclusion that Art is the grandparent of Cap is explained by citing the facts that Art is the parent of Bob, that Bob is the parent of Cap, and that the parent of a parent is a grandparent. The conclusion that Art is Bob's parent is explained by the fact that Art is Bob's father and that fathers are parents. The final interaction shows that it is also possible for the machine to show which conclusions depend on which premises.

FIGURE 5. Explanation

interpreter is a good example of this approach. Starting with a clause of the form :- $q_1, \ldots, q_n$, PROLOG finds the first clause of the form q :- $p_1, \ldots, p_m$, where q unifies with q_1, and then resolves the two. This process is then applied recursively with the resulting subgoal until the empty clause is produced. If this recursion fails to produce the empty clause, the interpreter backs up, finds the next applicable clause, and tries again. This process repeats until no further clauses can be found. Note that every resolution involves one clause of the form :- $q_1, \ldots, q_n$ and that no resolution involves a conjunct other than the first.

The following proof shows all of the deductions resulting from this procedure, not just the key steps in the successful proof. The nesting illustrates the subgoal–supergoal relationship among the various conclusions.

```
K1:   :- G(Art,Coe)
K2:        :- P(Art,y), P(y,Coe)
K3:             :- M(Art,y), P(y,Coe)
K4:             :- F(Art,y), P(y,Coe)
K5:                  :- P(Bob,Coe)
K6:                       :- M(Bob,Coe)
K7:                       :- F(Bob,Coe)
K8:                  :- P(Bud,Coe)
K9:                       :- M(Bud,Coe)
K10:                      :- F(Bud,Coe)
K11:                           :-
```

The disadvantage of this approach is potential inefficiency. If we pay no attention to how the clauses in a logic program are going to be used, we may end up writing them in a way that makes the derivation of desired conclusions computationally complex. There is a basic trade-off between cognitive cost for the programmer and computational cost for the machine. In light of this trade-off, many programmers compromise the methodological purity of logic programming to take details about the interpreter into account. In particular, most PROLOG programmers are careful to order their clauses with respect to each other and to order the conjuncts within each clause.

As an example, consider the ordering of the conditions on the right-hand side of rule `A3` (p. 935). In processing a query of the form `G(Art,z)`, PROLOG enumerates Art's children and tries to find a child for each. If the conditions had been written in the opposite order, PROLOG would enumerate all parent–child pairs and check to see whether Art is a parent for each. Since the number of parent–child pairs is likely to be very much larger than the number of Art's children, inverting the order would be disastrous.

Unfortunately, this ordering is not always best. For example, in using `A3` to answer the query `G(x,Coe)`, the situation is reversed. PROLOG would enumerate all parent–child pairs and check the child in each pair to see whether he or she is Coe's parent. For this query the opposite order would be computationally superior.

To resolve this dilemma, we need an interpreter that can determine the order of its deductions in a situation-specific way. The MRS interpreter is a good example of this approach. MRS differs from PROLOG in that it provides a vocabulary for expressing facts about the process of problem solving and not just about the content. In addition to *content* clauses that describe the application area of a program, we can write *control* clauses in MRS that prescribe how those content clauses are to be used (see Figure 6).

The basic cycle of the MRS interpreter is similar to the instruction fetch and execute cycle of a digital computer. The interpreter applies a PROLOG-like deductive procedure to the control clauses in a program to "fetch" the ideal deduction to perform on the content clauses. This deduction is then "executed," and the cycle repeats.

In writing control clauses, we use a conceptualization different from that of the application area. The objects include expressions such as variables, constants, atoms, and clauses. There are also actions such as the act of resolving a clause `p` with a clause `q` to produce a new clause `r` (written `R(p,q,r)`), relations on expressions such as "variable" (written `Var`) and "constant" (written `Const`), and relations on actions such as the binary relation "before" (written `Before`).

As an example of the control of deduction in MRS, consider once again the use of rule `A3`. The following control clauses state that it is better to work on a "parent of" goal in which one of the arguments is known than to work on a "parent of" goal in which both arguments are variable. Using this control clause, MRS would automatically choose the correct conjunct to work on, independent of the order of conjuncts speci-

```
Before(R(p1,q,r1),R(p2,q,r2)) :- Better(p1,p2)
Better(R1,R2)
Better(r1,r2) :- Length(r1,m), Length(r2,n), m<n
```

Automated reasoning with resolution is a nondeterministic process—it usually requires a search of multiple inference paths before a desired result can be proved. In some applications it is desirable to eliminate portions of the search space and to order the exploration of the remaining portions. Different logic programming systems provide different ways for the user to exercise this sort of control. PROLOG allows for implicit control via clause and conjunct ordering and explicit control via syntactic features like the ! operator. MRS emphasizes explicit control by providing a general metalevel control language. The first clause in this example states that any resolution involving clause `p1` should be done before any resolution involving clause `p2` if `p1` is "better than" `p2`. The second clause states that clause `R1` is better than clause `R2`. The third clause states the more general rule that short clauses are better than long clauses.

FIGURE 6. Control

fied by the "grandparent of" rule and of the position of the variable in the query:

```
L1:  Before(R(p1,q1,r1),R(p2,q2,r2)) :-
        Better(p1,p2)
L2:  Better(P(u,v),P(x,y)) :-
        Const(u), Var(v), Var(x), Var(y)
L3:  Better(P(u,v),P(x,y)) :-
        Var(u), Const(v), Var(x), Var(y)
```

The use of control clauses like these can substantially reduce the run time of a logic program. Unfortunately, the overhead of interpreting the control clauses offsets this improvement: For every step of a content-level deduction, the interpreter must complete an entire control-level deduction. In many applications the improved performance is clearly worth the overhead. In others, the overhead swamps the gains, and a simpler interpreter like PROLOG's is preferable.

EXAMPLE

Recent advances in design methodology and fabrication technology have made digital hardware of unprecedented complexity possible. The disadvantage of this complexity is that it substantially increases the difficulty of reasoning about designs. Logic programming is a powerful way of writing programs to assist designers in simulating, diagnosing, and generating tests for their designs.

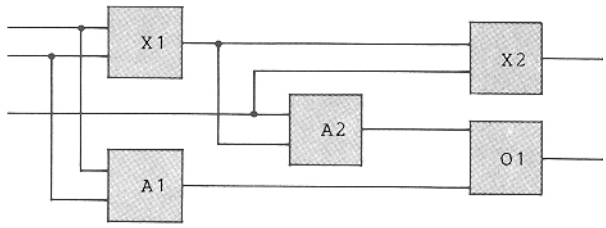
The "full adder" is a good example of a digital circuit. A full adder is essentially a one-bit adder with carry in and carry out, and it is usually used as one of n elements in an n-bit adder.

A graphical representation of the structure of the full adder F1 is shown in Figure 7. The structure of this circuit is described in clausal form below. Each part is designated by an object constant (e.g., `X1`). The structural type of each part is specified using type relations (e.g., `Xorg`); and the inputs and outputs (i.e., the ports) of each device are named using the functions `In` and `Out`. Connections are made between ports of devices.

```
M1:  Xorg(X1) :-
M2:  Xorg(X2) :-
M3:  Andg(A1) :-
M4:  Andg(A2) :-
M5:  Org(O1)  :-

M6:  Conn(In(1,F1),  In(1,X1))  :-
M7:  Conn(In(2,F1),  In(2,X1))  :-
M8:  Conn(In(1,F1),  In(1,A1))  :-
M9:  Conn(In(2,F1),  In(2,A1))  :-
M10: Conn(In(3,F1),  In(2,X2))  :-
M11: Conn(In(3,F1),  In(1,A2))  :-
M12: Conn(Out(1,X1), In(1,X2))  :-
M13: Conn(Out(1,X1), In(2,A2))  :-
M14: Conn(Out(1,A2), In(1,O1))  :-
M15: Conn(Out(1,A1), In(2,O1))  :-
M16: Conn(Out(1,X2), Out(1,F1)) :-
M17: Conn(Out(1,O1), Out(2,F1)) :-
```

The behavior of each of the components can also be described in clausal form. A proposition of the form



The full adder consists of two "xor" gates (`X1` and `X2`), two "and" gates (`A1` and `A2`), and an "or" gate (`O1`). The solid lines between the gates depict their interconnections. The device as a whole has three inputs and two outputs.

FIGURE 7. The Full Adder Digital Circuit F1

`I(n,y,z)` states that the value on the nth input of device `y` is `z`. A proposition of the form `O(n,y,z)` states that the value on the nth output of device `y` is `z`. The first six clauses characterize the behavior of "and," "or," and "xor" gates, and the final three characterize the behavior of connections.

```
N1:  O(1,x,1) :-
        Andg(x), I(1,x,1), I(2,x,1)
N2:  O(1,x,0) :-
        Andg(x), I(n,x,0)

N3:  O(1,x,1) :-
        Org(x), I(n,x,1)
N4:  O(1,x,0) :-
        Org(x), I(1,x,0), I(2,x,0)

N5:  O(1,x,1) :-
        Xorg(x), I(1,x,y), I(2,x,z), y#z
N6:  O(1,x,0) :-
        Xorg(x), I(1,x,z), I(2,x,z)

N7:  I(n,y,z) :-
        Conn(In(m,x),In(n,y)), I(m,x,z)
N8:  I(n,y,z) :-
        Conn(Out(m,x),In(n,y)), O(m,x,z)
N9:  O(n,y,z) :-
        Conn(Out(m,x),Out(n,y)), O(m,x,z)
```

The advantage of describing the circuit in this form is that we can use the description to reason about the circuit in a variety of ways. For example, we can simulate the behavior of the circuit by adding information about the values of the inputs to the knowledge base and proving facts about the outputs:

```
O1:  I(1,F1,1) :-
O2:  I(2,F1,0) :-
O3:  I(3,F1,1) :-
O4:  I(1,X1,1) :-
O5:  I(2,X1,0) :-
O6:  O(1,X1,1) :- I(1,X1,x), I(2,X1,y),
                  x#y
```

```
O7:  O(1,X1,1) :- I(2,X1,y), 1#y
O8:  O(1,X1,1) :- 1#0
O9:  O(1,X1,1) :-
O10: I(1,X2,1) :-
O11: I(2,X2,1) :-
O12: O(1,X2,0) :- I(1,X2,z), I(2,X2,z)
O13: O(1,X2,0) :- I(2,X2,1)
O14: O(1,X2,0) :-
O15: O(1,F1,z) :- O(1,X2,z)
O16: O(1,F1,0) :-
```

We can also diagnose faults in an instance of the circuit. In this case, let us suppose that the first output of the circuit is 1 instead of 0. Something must be wrong. Either a gate is not working correctly or a connection is bad. For simplicity, assume that all connections are guaranteed to be OK. In order to avoid contradictions, the type statements about the components must be removed from the knowledge base. By starting with a statement of the symptom (the negation of the expected behavior), we can deduce a set of suspect components. Recall that in a statement of the form $:- p, q$ is equivalent to saying that either p is false or q is false or both. Therefore, P17 asserts that either X2 is not acting as an "xor" gate or X1 is not acting as an "xor" gate, that is, that at least one of the two is broken.

```
P1:  :- O(1,F1,0)
P2:  :- Conn(Out(1,X2),Out(1,F1)),
        O(1,X2,0)
P3:  :- O(1,X2,0)
P4:  :- Xorg(X2), I(1,X2,z), I(2,X2,z)
P5:  :- Xorg(X2),
        Conn(Out(1,X1),In(1,X2)),
        O(1,X1,z), I(2,X2,z)
P6:  :- Xorg(X2), O(1,X1,z), I(2,X2,z)
P7:  :- Xorg(X2), Xorg(X1), I(1,X1,u),
        I(2,X1,v), u#v, I(2,X2,1)
P8:  :- Xorg(X2), Xorg(X1),
        Conn(In(1,F1),In(1,X1)),
        I(1,F1,u), I(2,X1,v), u#v,
        I(2,X2,1)
P9:  :- Xorg(X2), Xorg(X1), I(1,F1,u),
        I(2,X1,v), u#v, I(2,X2,1)
P10: :- Xorg(X2), Xorg(X1), I(2,X1,v),
        1#v, I(2,X2,1)
P11: :- Xorg(X2), Xorg(X1),
        Conn(In(2,F2),In(2,X1)),
        I(2,F1,v), 1#v, I(2,X2,1)
P12: :- Xorg(X2), Xorg(X1), I(2,F1,v)
        1#v, I(2,X2,1)
P13: :- Xorg(X2), Xorg(X1), 1#0,
        I(2,X2,1)
P14: :- Xorg(X2), Xorg(X1), I(2,X2,1)
P15: :- Xorg(X2), Xorg(X1),
        Conn(In(3,F1),In(2,X2)),
        I(3,F1,1)
P16: :- Xorg(X2), Xorg(X1), I(3,F1,1)
P17: :- Xorg(X2), Xorg(X1)
```

In diagnosing digital hardware, it is common to make the assumption that a device has at most one malfunctioning component at any one time. The clauses shown below provide a simple but verbose way of encoding this assumption. Clauses Q1–Q4 together state that either X1 is a working "xor" gate or the other devices are OK. Clauses Q1 and Q5–Q7 state the same for X2, and so forth. The single fault assumption can be stated more succinctly as a single axiom, but the encoding is somewhat more complex.

```
Q1:  Xorg(X1), Xorg(X2) :-
Q2:  Xorg(X1), Andg(A1) :-
Q3:  Xorg(X1), Andg(A2) :-
Q4:  Xorg(X1), Org(O1)  :-
Q5:  Xorg(X2), Andg(A1) :-
Q6:  Xorg(X2), Andg(A2) :-
Q7:  Xorg(X2), Org(O1)  :-
Q8:  Andg(A1), Andg(A2) :-
Q9:  Andg(A1), Org(O1)  :-
Q10: Andg(A2), Org(O1)  :-
```

Using the single fault assumption and the fact that a fault is guaranteed to be in some subset of parts, we can exonerate the parts not in that subset. For example, if we know that either X1 or X2 is broken, as in the example above, we can prove that components A1, A2, and O1 are OK. The following proof makes the case:

```
R1:  :- Xorg(X1), Xorg(X2)
R2:  Andg(A1) :- Xorg(X2)      R1  Q2
R3:  Andg(A1) :-               R2  Q5
R4:  Andg(A2) :- Xorg(X2)      R1  Q3
R5:  Andg(A2) :-               R4  Q6
R6:  Org(O1)  :- Xorg(X2)      R1  Q4
R7:  Org(O1)  :-               R6  Q7
```

Finally, we can devise tests to discriminate possible suspects. Starting with a behavioral rule for one of the suspect components, we can derive a behavioral expectation for the overall device that implicates a subset of the suspects. For example, clause S18 below states that using the same inputs as in the previous example when the output is not 1 implies that Xorg(X1) is false.

```
S1:  O(1,X1,1) :-
        Xorg(X1), I(1,X1,y), I(2,X1,z),
        y#z
S2:  O(1,X1,1) :-
        Xorg(X1), I(1,X1,1), I(2,X1,0)
S3:  O(1,X1,1) :-
        Xorg(X1),
        Conn(In(1,F1),In(1,X1)),
        I(1,F1,1), I(2,X1,0)
S4:  O(1,X1,1) :-
        Xorg(X1), I(1,F1,1), I(2,X1,0)
S5:  O(1,X1,1) :-
        Xorg(X1), I(1,F1,1),
        Conn(In(2,F1),In(2,X1)),
        I(2,F1,0)
S6:  O(1,X1,1) :-
        Xorg(X1), I(1,F1,1), I(2,F1,0)
```

```
S7:  I(2,A2,1) :-
        Xorg(X1), I(1,F1,1), I(2,F1,0),
        Conn(Out(1,X1),In(2,A2))
S8:  I(2,A2,1) :-
        Xorg(X1), I(1,F1,1), I(2,F1,0)
S9:  O(1,A2,1) :-
        Xorg(X1), I(1,F1,1), I(2,F1,0),
        Andg(A1), I(1,A2,1)
S10: O(1,A2,1) :-
        Xorg(X1), I(1,F1,1), I(2,F1,0),
        I(1,A2,1)
S11: O(1,A2,1) :-
        Xorg(X1), I(1,F1,1), I(2,F1,0),
        Conn(In(3,F1),In(1,A2)),
        I(3,F1,1)
S12: O(1,A2,1) :-
        Xorg(X1), I(1,F1,1), I(2,F1,0),
        I(3,F1,1)
S13: I(1,O1,1) :-
        Xorg(X1), I(1,F1,1), I(2,F1,0),
        I(3,F1,1),
        Conn(Out(1,A2),In(1,O1))
S14: I(1,O1,1) :-
        Xorg(X1), I(1,F1,1), I(2,F1,0),
        I(3,F1,1)
S15: O(1,O1,1) :-
        Xorg(X1), I(1,F1,1), I(2,F1,0),
        I(3,F1,1), Org(O1)
S16: O(1,O1,1) :-
        Xorg(X1), I(1,F1,1), I(2,F1,0),
        I(3,F1,1)
S17: O(2,F1,1) :-
        Xorg(X1), I(1,F1,1), I(2,F1,0),
        I(3,F1,1),
        Conn(Out(1,O1),Out(2,F1))
S18: O(2,F1,1) :-
        Xorg(X1), I(1,F1,1), I(2,F1,0),
        I(3,F1,1)
```

The use of logic programming in this application area has several important advantages. The most obvious is that a single design description can be used for multiple purposes. We can simulate a circuit, diagnose it, and generate tests all from one description. A second advantage is that the expressive power of the language allows higher level design descriptions to be written and used for these purposes. By working with more abstract design descriptions, these tasks can be performed far more efficiently than at the gate level. Finally, the flexibility of the language and deductive techniques allow these tasks to be performed even with incomplete information about the structure or behavior of a design.

CONCLUSION

The practicality of logic programming as a software-engineering methodology depends to a large degree on the underlying technology. Unfortunately, there are several areas where technological progress is currently needed.

One of the key limitations of most logic programming systems is that their deductive methods are inadequate. Resolution is complete in that it can prove any conclusion logically implied by its knowledge base. However, a good logic programming system should be able to draw conclusions from uncertain data, reason analogically, and generalize its knowledge appropriately if it is to be really effective.

Another problem with current logic programming systems is the inefficiency caused by the absence of user-specified control. The development of smarter interpreters and compilers should relieve the logic programmer of much of the burden of control.

Finally, there is a need for considerable improvement in the usability of logic programming systems. This situation could be remedied by the design of more perspicuous languages for expressing knowledge and by the development of better tools for manipulating and debugging logic programs.

Although the technology of logic programming is already adequate for supporting the methodology in a wide variety of situations, advances in the functionality, efficiency, and usability of logic programming systems should substantially broaden the range of applicability. Although we do not expect that logic programming will completely supplant traditional software engineering, its advantages and range of applicability suggest that it may become the dominant programming methodology in the next century.

FURTHER READINGS

1. Clocksin, W.F., and Mellish, C.S. *Programming in PROLOG*. Springer-Verlag, New York, 1981.
2. Genesereth, M.R. Partial programs. HPP-84-1, Heuristic Programming Project, Stanford University, Calif., 1984.
3. Genesereth, M.R. The role of design descriptions in automated diagnosis. *Artif. Intell. 24*, 1-3 (Dec. 1984), 411-436.
4. Genesereth, M.R., Greiner, R., and Smith, D.E. MRS—A meta-level representation system. HPP-83-27, Heuristic Programming Project, Stanford University, Calif., 1983.
5. Hayes, P. Computation and deduction. In *Proceedings of the 2nd MFCS Symposium*. Czechoslovak Academy of Sciences, 1973, pp. 105-118.
6. Kowalski, R. Algorithm = logic + control. *Commun. ACM 22*, 7 (July 1979), 424-436.
7. Kowalski, R. *Logic for Problem Solving*. North-Holland, Amsterdam, 1979.
8. McCarthy, J. Programs with common sense. In *Semantic Information Processing*, M. Minsky, Ed. MIT Press, Cambridge, Mass., 1968, pp. 403-410.
9. Moran, T. Efficient PROLOG pushes AI into wider market. *Mini-Micro Syst.* (Feb. 1985).
10. Smith, D.E., and Genesereth, M.R. Ordering conjunctive queries. *Artif. Intell. 26*, 2 (May 1985), 171-215.

CR Categories and Subject Descriptors: D.2.m [**Software Engineering**]: Miscellaneous—*rapid prototyping*; D.3.2 [**Programming Languages**]: Language Classifications—*very high-level languages*; I.2.3 [**Artificial Intelligence**]: Deduction and Theorem Proving—*answer/reason extraction, deduction, logic programming, metatheory*

General Terms: Languages

Additional Key Words and Phrases: MRS, PROLOG

Authors' Present Address: Michael R. Genesereth and Matthew L. Ginsberg, Computer Science Dept., Stanford University, Stanford, CA 94305.