LOGIC PROGRAMMING

To design an entity (a machine or a program) capable of behaving intelligently in some environment, it is necessary to supply this entity with sufficient knowledge about this environment. To achieve this, computer scientists have developed a collection of programming languages that serve as means of communication with the machines. It is customary to distinguish between two types of knowledge: (1) *procedural* ("knowing how") and (2) *declarative* ("knowing that"). This difference led to classifying paradigms for programming languages into two distinct types, *imperative* and *declarative.* The imperative languages, like Pascal and C, specify how a computation is performed by sequences of changes to the computer's store. The declarative languages are more concerned with specifying what is to be computed. Logic programming belongs to the declarative programming paradigm, which strives to reduce a substantial part of a programming process to the description of objects comprising the domain of interest and relations between these objects.

The software development process in this paradigm starts with a natural language description of the domain, that, after a necessary analysis and elaboration, is translated into a collection of logical axioms of an unambiguous logical language containing information from this description which is relevant to the problem (or problems) at hand. Such a collection of axioms can be viewed as a declarative program. Programs whose axioms are "logical rules", that is, statements of the form A if B_1 and . . . and B_n where $0 \leq n$, are called *logic programs.* The language of relational databases and various functional languages also have substantial declarative components which allow only more restrictive forms of axioms. A logic program can be executed by providing it with a problem, formalized as a logical statement to be proved, called a goal statement (or a query). The execution is an attempt to solve a problem, that is, to prove the goal statement, given the axioms of the logic program. The proof provided by a program should be constructive. This means that if the goal statement is existentially quantified, that is, it states that there is some object satisfying some property, then the proof provides identity of this unknown object. In summary: *a logic program is a collection of axioms; computation is a constructive proof of a goal statement from the program.*

SYNTAX OF PURE PROLOG

These ideas can be illustrated by writing a program in a logic programming language called Pure Prolog. We start with describing syntax of our language suitable for formalization of a particular domain. The syntax will contain constants, that will be used to name objects of the domain, functions and relations between these objects, and names for variables over the objects. A collection of these symbols is called a *signature.* Names of relations of a signature σ are often called *predicate symbols.* In what follows, constants will be denoted by strings of letters and digits that start with a lower-case letter. Sequences of the same type that start with capital letters denote variables. The underscore is also used to make the names more readable. To define sentences of a language $\mathscr{L}$ over signature σ an auxiliary notion of *term,* is needed, defined as follows:

1. Constants and variables of σ are terms;
2. If f is a function symbol and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.
3. Nothing else is a term.

Terms not containing variables are called *ground.* They are used to name objects of the program domain. If $t_1, \ldots, t_n$ are ground terms and p is a predicate symbol, then a string $p(t_1, \ldots, t_n)$ is read as "objects denoted by $t_1, \ldots, t_n$ satisfy property p" and is called an *atom.*

The above vocabulary provides the basis for construction of all logic programming languages. A particular logic programming language can be characterized by the type of statements which can serve as axioms of its programs. Pure Prolog allows two types of such statements: facts and rules. *Facts* are atoms. *Rules* are statements of the form:

1. p_0 :- $p_1, \ldots, p_n$

where $p_0, \ldots, p_n$ are atoms. The sequence $p_1, \ldots, p_n$ is called the *body* of the rule and p_0 is called its *head.* In what follows we identify atoms with rules with the empty bodies. The symbol ":-" in rule (1) can be viewed as a form of implication, "," stands for the logical conjunction $\wedge$, and variables are assumed to be universally quantified over the objects of the program domain. If $X_1, \ldots, X_m$ are variables occurring in rule (1) then the rule (1) is read declaratively as "Any $X_1, \ldots, X_m$ satisfying conditions $p_1, \ldots, p_n$ satisfy condition p_0." In addition to the declarative reading of rule (1), it can also be read as follows: to solve (execute) p_0, solve (execute) p_1 and p_2 and . . . p_n. This procedural reading of rules, first formulated by Kowalski (1), serves as the basis of proof procedure implemented in Prolog interpreters and compilers. Now a logic program can be defined in Pure Prolog (with some underlying signature σ) simply as a collection of rules. A set of rules of a program whose heads are atoms formed by the same relation r is sometimes called a definition of r. So a program can be viewed as a collection of definitions of relations between the objects of the program domain. For simplicity it can be assumed that queries in Pure Prolog are atoms. (More complex queries are allowed in practice.)

Now assume that it is necessary to construct a logic program containing information about a small computer science department. Assume that the department has three professors, Smith, Jones, and Domingez; that this summer it offers classes in Prolog (cs1), Pascal (cs2), and Data Structures (cs3), taught by Smith, Jones, and Domingez, respectively. This information can be expressed by the following atomic sentences of Pure Prolog:

1a. *course*(*cs*1,*prolog*).
1b. *course*(*cs*2,*pascal*).
1c. *course*(*cs*3,*data_structures*).
2a. *is_prof*(*smith*,*cs*).
2b. *is_prof*(*jones*,*cs*).
2c. *is_prof*(*domingez*,*cs*).
3a. *teaches*(*smith*,*cs*1).
3b. *teaches*(*jones*,*cs*2).
3c. *teaches*(*domingez*,*cs*3).

So far, communication with the program occurred in the "teaching" mode, that is, the above facts were simply stored in a file. To query the program we need to switch into the "querying" mode. The Prolog interpreter will load the program and respond by prompting us with a ?, indicating that it is ready for questioning. We start with a simple query

? teaches(smith,cs1)

interpreted as "Does Smith teaches cs1 ?". The program will answer "Yes" and prompt us for the next question. Asked

? teaches(jones,cs1)

the program will answer "No." The answer can be interpreted in two different ways. It may mean "No, I have not been able to prove that Jones teaches cs1." It may also mean "No, Jones does not teach cs1." The second interpretation is valid only if our summer schedule is complete. In this case, inability to prove that Jones teaches cs1 is equivalent to this statement being false. The assumption of completeness of information about the program domain encoded by axioms of the program is called the closed world assumption (2). This assumption has been proven useful for formalization of various domains and, as a result, is embodied in the semantics of Pure Prolog. Thanks to this assumption we have a powerful language which does not contain negation. Later we show how Pure Prolog can be extended to deal with incomplete information.

The queries we have asked so far did not contain variables. For programs consisting entirely of facts, such queries are answered by a simple table lookup. The situation becomes more complicated for queries with variables. Suppose we want to find out which of the professors is teaching *cs*1. To do that we need to use a variable. The corresponding query

? teaches(X,cs1)

is a request to constructively prove the statement $\exists X teaches(X,cs1)$. Procedurally, it can be read as "Find X such that *teaches*(X,*cs*1) is true". The query will be answered by

$X = smith$

If we want to find out which class is taught by Jones we issue a query:

? teaches(jones,X)

which will be answered by

$X = cs2$

In these examples the answer is obtained by matching our queries against the facts of the program. The matching process attempts to make the query identical to a fact by a process of substituting terms of the language for variables in the corresponding sentences. In our simple case such a substitution is easily found and reported as answer to a query. In general, however, the situation is much more complex. The matching is performed by nontrivial unification algorithm (3,4) which we describe shortly. Meanwhile, let us go back to the teaching mode and communicate to the program more knowledge about the department. Suppose we are interested in the relation *subject_taught*(S,P), which is true iff subject S is taught by a professor P. To define this relation for a computer we may use a rule:

```
4a. subject_taught(S,P) :-
  teaches(P,C),
  course(C,S)
```

Now if we want our program to tell who is teaching a class in Pascal we can ask a query

? subject_taught(pascal,P)

which will be answered by

$P = jones$

The Prolog interpreter will answer this query by: finding the rule (4*a*) whose head matches the query by substituting *pascal* for S; asking query *teaches*(P,C) and answering it with $P = smith$ and $C = cs1$; asking query *course*(*cs*1,*pascal*) and answering it with "No"; backtracking to the query *teaches*(P,C) and answering it with a new answer $P = jones$ and $C = cs2$; checking that *course*(*cs*2,*pascal*) is true, succeeding, and returning $P = jones$. This is, of course, the only answer to the query which can be obtained from our program. So if we ask the interpreter to find another answer (which, on most systems can be done by simply typing a ";"), the interpreter will respond with "No." In general, however, a query q with variables may allow more than one answer. If the set of answers to q is finite we can ask for and get all the answers. In case of infinite collection of answers we can get one answer at a time. We hope that this example gives the reader a flavor of programming in logic.

Before we go to more precise mathematical treatment of Pure Prolog and to extensions of this language we would like to demonstrate one more interesting feature common to all logic programming language—the ability to define relations recursively. Suppose we want to inform our program that the CS department in question belongs to the engineering college of the small university known as "the school." This can be done by giving the program the following rules:

```
5a. belongs_to(cs,engr)
5b. belongs_to(engr,the_school)
```

This, however, will not allow us to conclude that our CS department belongs to (or is part of) the school. This information is, of course, implicit in the informal description of the domain and should therefore be made known to the program. To achieve this we could simply add *belongs_to*(*cs*,*the_school*) but this solution obviously would not be sufficiently general. It will not be, for instance, feasible for large hierarchies. Instead, we will define a new relation "part_of(X,Y)" defined as the transitive closure of "belongs_to." This can be done by expanding the program by the rules:

```
6a. part-of(X,Y) :-
              belongs_to(X,Y)
6b. part_of(X,Y) :-
              belongs_to(X,Z)
              part_of(Z,Y)
```

Notice that the last rule has occurrences of the same predicate symbol in the head and in the body. Rules satisfying this property are called *recursive;* such rules are needed to define transitive closures and other useful relations and, to a large degree, are responsible for the great expressive power of Pure Prolog. It can be formally shown that neither a standard relational database query language SQL nor the first-order logical languages commonly used for formalization of knowledge in artificial intelligence are capable of expressing a notion of transitive closure of a binary relation. As always, there is a trade-off between expressivity and efficiency of the language, and recursive rules can be a source of inefficiency and even nontermination of logic programs. The attentive reader probably noticed that we did not really give a good justification for the introduction of the relation "part_of" in the language of our program. The same information could have been communicated by simply adding a rule:

```
belongs_to(X,Y) :-
               belongs_to(X,Z)
               belongs_to(Z,Y)
```

Even though this rule can be used in some logic programming systems (5) it is unacceptable in Prolog. The reason is that, in the presence of this rule, the Prolog interpreter may not terminate on some simple queries, such as, q = "? belongs_to(X,Y)." This, of course, follows from the procedural interpretation of Prolog rules: to answer the above query the interpreter will need to answer the query "? belongs_to(X,Z)," which is essentially the same as q and hence causes the interpreter to loop. (Recall that both queries are read as "find a pair of objects satisfying relation "belongs_to.")

It is also worth noticing that transitivity of the relation *part_of* has not been explicitly stated in its informal description. It is rather a "commonsensical" property of the relation, something "everyone knows." Discovering such properties of various relations and giving them to a program constitutes an important part of the art of declarative programming. Here is another such rule, undoubtedly understandable to humans but not yet known to the program.

```
2d. is_prof(X,P) :-
              part_of(Q,P)
              is_prof(X,Q)
```

The rule [and the new definition of the relation "is_prof," consisting of rules (2*a*)–(2*d*)] is obviously recursive. Going back to the querying mode we can ask a program if "the school" has a professor called Jones. The corresponding query will have a form

```
? is_prof(jones,the_school)
```

and will be answered by "Yes." Notice that this answer requires more reasoning then previous ones. We will reason, for instance, that using rules (5) and (6) one can show that the CS department is a part of the school; that by fact (2b) Jones is a professor in this department and, therefore, by rule (2d), Jones is the professor in the school. In general, it may be difficult to write logic programs without a good understanding of the semantics and the inference mechanism used by a particular language. Now we give a mathematical treatment of the semantics and the underlying inference mechanism of Pure Prolog.

INFERENCE IN PURE PROLOG

By pure logic programs we mean programs of Pure Prolog with some underlying signature σ. By *ground*(Π) we denote the set of all rules obtained from program Π by replacing variables in the rules by the ground terms of σ. To give the semantics of pure logic programs we define what ground atoms of σ are "consequences" rules of the program. In doing that we treat atomic sentences of a program as axioms and its nonatomic rules as the inference rules. This suggests the following definitions:

Let Π be a ground program, that is, a program not containing variables. We say that a set of ground atoms S is closed under Π if for every rule (1) in Π, $p_0 \in S$ whenever $\{p_1, . . ., p_n\} \subseteq S$. The *set of consequences* of Π is the smallest set of ground atoms of σ closed under the rules of Π. It is not difficult to show that such a set always exists. The set of consequences of a pure logic program Π which contain variables is defined as the set of consequences of *ground*(Π). We denote this set by $Cn(\Pi)$ and write $\Pi \models q$ if $q \in Cn(\Pi)$. A conjunction $q_1 \wedge . . . \wedge q_n$ of ground atoms is true in a set of atoms if all the q's are true in this set. It is false otherwise. A *query of Pure Prolog* is a conjunction of atoms. Let Q be such a query with variables $X_1, . . ., X_n$. A sequence $t_1, . . ., t_n$ of ground terms is an *answer* to query Q if $Q(t_1, . . ., t_n)$ is true in $Cn(\Pi)$. [If $n = 0$ and Q is true in $Cn(\Pi)$ then the answer to Q is "Yes"]. If no such sequence exists then the answer to Q is "No". All the answers returned to our queries by the example program above are indeed the answers according to this definition.

The consequence relation of Pure Prolog has several nice properties. It is monotone, that is, if $\Pi_1 \subseteq \Pi_2$ then $Cn(\Pi_1) \subseteq Cn(\Pi_2)$. It is compact, that is, every consequence of Π is a consequence of a finite subset of Π. $Cn(\Pi)$ can be characterized as the least fixpoint of the function T_Π defined on the sets of ground atoms of σ such that $T_\Pi(S)$ is the set of heads of the rules of *ground*(Π) whose bodies are subsets of S. Thus $T_\Pi(S)$ is the set of ground atoms which can be derived from S "in one step" using the rules of *ground*(Π). Obviously, T_Π is monotone and hence, according to the general fixpoint theory, has the least fixpoint. Moreover, this fixpoint is equal to $Cn(\Pi)$ (6). By the same theory, the union of the sets obtained by iterating T_Π on the empty set $\emptyset$ is a subset of the least fixpoint of T_Π. For this particular function, the union happens to be equal to this fixpoint, that is,

$$Cn(\Pi) = \cup_{n \geq 0} T_\Pi^n(\emptyset)$$

This observation suggests the method of bottom-up evaluation of logic program which is sometimes used for answering queries in Datalog—a logic programming query language which can be viewed as Pure Prolog without function symbols and a finite collection of constants. These two conditions guarantee that every Datalog query has a finite set of answers. In database applications we are usually interested in obtaining all the answers to a query which makes this property especially important. In its simplest form the method consists in grounding Π and applying T_Π operator until it reaches the fixpoint. Various optimization techniques (7) allow us to use the goal or the class of goals to avoid the grounding of complete programs, to speed up the evaluation of recursive queries, and so forth. A detailed description of these methods can be found in (7). In the next section we describe a more general inference mechanism which is implemented in Prolog interpreters and compilers. It is based on the resolution-style proof of Robinson (4) and is based on a body of work in mathematical logic and automated theorem proving which can be traced back to Herbrand's work in the 1930s (3). Resolution-style proof systems are defined for logical languages whose expressive power substantially exceeds that of Pure Prolog. We define this system for so called *clausal theories*—collections of universally quantified formulas of the form

2. $l_1 \vee \ldots \vee l_n$

where l's are literals, that is, atoms and their negations and $\vee$ is a logical *or*. (Negation of atom p will be denoted by $\neg p$.) It is convenient to identify formula of the form (2) with a set of literals $\{l_1, \ldots, l_n\}$. Let S be a set of ground atoms. Ground atom p is *true* in S if $p \in S$; ground literal $\neg p$ is *true* in S if $p \notin S$; a ground clause C is *true* in S if at least one literal of C is true in S. Let $\mathcal{C}$ be a clausal theory. A set S of ground atoms is called a *model* of $\mathcal{C}$ if all clauses of $\mathcal{C}$ are true in S; $\mathcal{C}$ is called *unsatisfiable* if it has no model. Notice that the empty clause, normally denoted by $\square$, has no model and hence any theory containing $\square$ is unsatisfiable. We say that a conjunction Q of literals is a consequence of a clausal theory $\mathcal{C}$ if Q is true in all models of $\mathcal{C}$. Let $Q = l_1 \wedge \ldots \wedge l_n$. It is easy to see that Q is a consequence of $\mathcal{C}$ iff the set $\mathcal{C} \cup \neg Q$ (where $\neg Q = \{\neg l_1, \ldots, \neg l_n\}$) is unsatisfiable. (Here and below we identify $\neg\neg p$ with p.) The resolution proof system uses this observation to reduce the question of derivability of a query Q from $\mathcal{C}$ to the question of unsatisfiability of $\mathcal{C} \cup \neg Q$. It is based on the unification algorithm performing matching between atoms of the language. To describe the algorithm we need some preliminary definitions.

Let E be a finite set of equations of the form $X_1 = t_1, \ldots, X_n = t_n$ where X's are distinct variables, t's are terms and for any i, X_i is different from t_i. By an expression we mean a term, a literal or a set of literals. A *substitution* α (defined by E) is a mapping that maps an expression e into the expression $\alpha_E(e)$ obtained by simultaneously replacing each occurrence of $X_1, \ldots, X_n$ in e by the corresponding term. α is called a *unifier* of expressions p, q if $\alpha(p) = \alpha(q)$. α is called a *most general unifier* (mgu) of p and q if:

1. $\alpha(p) = \alpha(q)$.
2. For any unifier β of p and q there is a unifier γ such that for every expression e, $\beta(e) = \gamma[\alpha(e)]$.

The substitution $X = g(Z)$, $Y = b$, $U = a$ is an mgu of atoms $p[f(X, Y), a]$ and $p(f(g(Z), b), U)$. Atoms $p(f(X))$ and $p(a)$ are not unifiable. We sketch an algorithm which for any two atoms A and B produces their mgu if they are unifiable and otherwise reports nonexistence of a unifier. The particular version of the algorithm presented below is due to Martelli and Montanari (8).

If A and B are formed by different predicate symbols then stop with failure. Otherwise replace atoms $p(t_1, \ldots, t_n)$ and $p(s_1, \ldots, s_n)$ by the set of equations $S_0 = \{t_1 = s_1, \ldots t_n = s_n\}$, nondeterministically choose an equation from S_0 and perform the action from the corresponding entry in the table below.

Equation	Action
(1) $f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$	Replace by $t_1 = s_1, \ldots, t_n = s_n$
(2) $f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$	Stop with failure
(3) $X = X$	Delete the equation
(4) $t = X$ where t is not a variable	Replace by $X = t$
(5) $X = t$ where X is different from t and X has another occurrence in the set of equations	If X occurs in t then stop with failure, else replace occurrences of X by t in every other equation

The algorithm stops with failure or returns a collection of equations of the form $X_1 = t_1, \ldots, X_n = t_n$ which define an mgu of A and B. To complete definition of resolution we need more terminology. Two clauses C_1 and C_2 are called complementary if there exist atoms p_1 and p_2 such that

1. $p_1 \in C_1$
2. $\neg p_2 \in C_2$
3. p_1 and p_2 are unifiable

Literals p_1 and $\neg p_2$ are called resolving literals. Let C_1 and C_2 be two clauses and let C be the result of replacing variables of C_1 by new variables not occurring in C_2. If C_1 and C_2 are complementary with resolving literals l_1 and l_2 and a corresponding mgu α then the clause $C = \alpha((C\backslash\{l_1\}) \cup (C_2\backslash\{l_2\}))$ is called a *resolvent* of C_1 and C_2.

If a clause C contains literals l_1 and l_n unifiable by an mgu α then the clause $\alpha(C\backslash\{l_1\})$ is called a *factor* of C.

A sequence $C_1, \ldots, C_n$ of clauses is called a *resolution derivation of* C_n from a set of clauses $\mathscr{C}$ ($\mathscr{C} \vdash C_n$) if for every $i \in [1..n]$ $C_i \in \mathscr{C}$ or C_i is a resolvent or a factor of some previous elements of the sequence.

Theorem. A set of clauses $\mathscr{C}$ is unsatisfiable iff there is a resolution derivation of the empty clause from $\mathscr{C}$ (4).

This implies that to check if a query Q is a consequence of clausal theory $\mathscr{C}$ it suffices to check if there is a resolution derivation of $\Box$ from $\mathscr{C} \cup \{\neg Q\}$. The following algorithm returns answer "true" for any unsatisfiable set of clauses $\mathscr{C}$. If $\mathscr{C}$ is satisfiable the algorithm returns "false" or goes into infinite loop. In what follows by $R(V)$ we denote V united with the set of all resolvents and factors of clauses from V.

function simple_resolution($\mathscr{C}$: clausal_theory) : boolean
var W, V : clausal_theory
$W := \mathscr{C}$
repeat
  $V := W$
  $W := R(V)$
until $(\Box \in W) \vee (V = W)$
if $(\Box \in W)$ then return(true) else return(false)

At least two aspects of this proof procedure can be substantially improved. First we can modify the procedure to expand the class of causal theories on which it terminates. It is known, however, that the consequence relation in clausal theories is undecidable, that is, there is no algorithm which terminates on any clausal theory $\mathscr{C}$ and query Q and returns true iff Q is a consequence of $\mathscr{C}$. This means that the above procedure is bound to go into infinite loop on some inputs. Second, the efficiency of the procedure can be substantially improved by the goal-dependent selection of resolvents and other refinements. Now we briefly describe how resolution method is used in Prolog.

First, we map a rule (1) of Pure Prolog into a clause $\{p_0, \neg p_1, \ldots, \neg p_n\}$. A program Π of Pure Prolog then becomes a collection of clauses $\mathscr{C}(\Pi)$. It is possible to show that a query Q is a consequence of Π iff it is a consequence of $\mathscr{C}(\Pi)$. Prolog interpreter answers the query $Q = q_1 \wedge \ldots \wedge q_n$ by converting it into a clause $G = \{\neg q_1, \ldots, \neg q_n\}$ and asking if $\mathscr{C}(\Pi) \cup G$ is unsatisfiable. To answer this question the interpreter will use a special form of resolution called *linear resolution.* A linear resolution proof of a clause C from a clausal theory $\mathscr{C}$ is a sequence of pairs $\{C_0, B_0\}, \ldots, \{C_n, B_n\}$, such that $C = C_n$ and

1. $C_0 \in \mathscr{C}$ and each B_i is element of $\mathscr{C}$ or equals some C_j with $j < i$.
2. Each C_{i+1}, $i \leq n$, is a resolvent of C_i and B_i.

A linear derivation of $\Box$ from $\mathscr{C}$ is called a *linear refutation* of $\mathscr{C}$.

The Prolog inference engine checks if $\Pi \cup G$ is unsatisfiable by looking for linear refutation of $\Pi \cup G$ with $C_0 = G$. In general, linear resolution is incomplete, that is, a set of clauses $\mathscr{C}$ may be unsatisfiable but there may be no linear refutation of $\mathscr{C}$. It can be shown, though, that for Π and G defined as above $\Pi \cup G$ is unsatisfiable iff there is a linear refutation of $\Pi \cup G$ which starts with G. To complete the description of the Prolog inference engine we need to specify how to select a clause B_i from Π and the resolving literal l from C_i. The latter can be done by ordering literals in C_i and defining a selection rule which chooses l. Natural order of literals is given by the form of rules in Π. The selection rule used in most implementations of Prolog is to always resolve on the first, that is, the leftmost, literal in C_i. The resulting clause C_{i+1} preserves the order of literals in C_i and B_i with the former positioned to the left of the latter. We call this *SLD-resolution.* This restriction preserves soundness and completeness of linear resolution. Completeness is, however, lost in the process of selecting a clause B_i from Π to resolve with C_i. Prolog normally does that by selecting the first clause in Π which is possible, in some cases causing the inference engine going into the loop. Consider, for instance, a program

```
p :- p
p
```

and a query p. According to the above strategy, the inference engine will use the first rule forever and never get to the second one. A similar thing happens with our recursive definition of relation *belongs_to* in the first example. As mentioned before, there are several logic programming systems that use better strategies. Still, fully avoiding these types of problems remains the responsibility of the programmer.

REPRESENTING INCOMPLETE INFORMATION

Recall that, since the semantics of Pure Prolog adopts the closed world assumption, no negation was allowed in its syntax. We introduce two extensions of Pure Prolog that allow negative statement and are more suitable for reasoning with incomplete information. Consider the following example: assume that the schedule of our CS department is represented by the table

Professor	Course
smith	cs1
jones	cs2
staff	cs3

Here *staff* is a so called *null* value (a vaguely defined databases term) which stands for an unknown professor (possibly different from Smith and Jones). A person looking at this table will conclude that Smith teaches cs1 and does not teach cs2, but, since the identity of "staff" is not known, will not be able to tell if Smith teaches cs3. It is easy to see that the Pure Prolog program

```
(f1) teaches(smith,cs1)
(f2) teaches(jones,cs2)
(f3) teaches(staff,cs3)
```

does not capture this reasoning. Indeed, the program answers "No" to both queries: teaches(smith,cs2) and teaches (smith,cs3). We need to answer the first one by "No" and the second one by "Unknown". To deal with the problem we expand the language of Pure Prolog by allowing rules of the form:

3. l_0 :- $l_1, \ldots, l_n$

where l's are literals over some signature σ and $0 \le n$. The semantics of the new language, called Basic Prolog, is similar to that of Pure Prolog. The set of consequences of a program Π of Basic Prolog is defined as the smallest set S of ground literals of σ which satisfies two conditions:

1. S is closed under the rules of $ground(\Pi)$.
2. If S contains an atom p and its negation $\neg p$, then S contains all ground literals of the language.

The second condition corresponds to the rule of classical logic which allows any formula to be entailed from a contradiction. Every program Π has a unique set of consequences. As before, we denote this set by $Cn(\Pi)$. A ground conjunction $Q = l_1 \wedge \ldots \wedge l_n$ is *true* in a set S of literals if $l_i \in S$ for every $1 \le i \le n$; Q is *false* in S if for some i, $\neg l_i \in S$; Q is *unknown* in S otherwise.

A *query of Basic Prolog* is a conjunction of literals. Let Q be such a query with variables $X_1, \ldots, X_n$. A sequence $t_1, \ldots, t_n$ of ground terms is an *answer* to a query Q if $Q(t_1, \ldots, t_n)$ is true in $Cn(\Pi)$; if for any such sequence $Q(t_1, \ldots, t_n)$ is false in $Cn(\Pi)$, then the answer to Q is “No”; otherwise the answer is *unknown*. Information from the table above can be represented by the program consisting of positive facts (f1)–(f3) and negative facts

(f4) $\neg teaches(smith, cs2)$
(f5) $\neg teaches(jones, cs1)$

Observe that the program properly answers our queries. It does, however, require an explicit representation of negative facts which make this method of representation impractical for large databases. This problem is solved by using another logic programming connective, *not,* called *negation as failure* or *default negation.*

NEGATION AS FAILURE

Intuitively, *not* l is an “epistemic” connective read as “there is no reason to believe that l is true.” Procedurally, a query *not* l succeeds if l is ground and all the attempts to prove l finitely fail. We give a precise semantics of *not* shortly, but first let us see how it can help with our example. Consider a program consisting of the facts (f1)–(f3) and rules

(r1) $\neg teaches(P,C)$:-
  not $teaches(P,C)$
  not $ab(r1,P,C)$
(r2) $ab(r1,P,C)$:-
  $teaches(staff,C)$

The first rule allows us to conclude by default that a given professor P does not teach a given class C. A symbol $r1$ is used to name this rule; the symbol ab stands for “abnormal”—a relation used for expressing exceptional status of objects to which the corresponding default is not applicable. Given a query, say, $q_1 = teaches(smith,cs1)$, the program will attempt to prove q_1 and $\neg q_1$; q_1 is proven by matching with (f1); attempt to prove $\neg q_1$ leads to a new query *not* q_1 (read as “cannot prove q_1”), which fails. Hence, the answer to q_1 is “Yes”. Suppose now that $q_2 = teaches(smith, cs2)$. The program attempts to prove q_2 and fails. Attempts to prove $\neg q_2$ leads to a query *not* q_2; q_2 fails and, hence, *not* q_2 succeeds; Similarly, for $ab(r1, smith, cs2)$; hence the answer to q_2 is “No.” Finally, consider $q_3 = teaches(smith, cs3)$. It is easy to see that the program can prove $ab(r1, smith, cs3)$. Hence, neither q_3 nor $\neg q_3$ can be proven and the answer to q_3 is “Unknown.”

Originally negation as failure *not* was introduced in logic programming as a purely procedural device. The first declarative semantics of *not* was given in the pioneering work of Clark (9). Some difficulties with this semantics led researchers to the development of several alternative semantics for negation as failure (10–13). We give a precise definition of *answer set semantics* for programs with negation as failure (14). A survey of different approaches to semantics of negation as failure can be found in (15). Let us introduce an extension of Basic Prolog called A-Prolog. Programs of A-Prolog are collections of rules of the form

6. $l_0 : -l_1, \ldots, l_n, not\ l_{n+1}, \ldots, not\ l_m$

where l's are literals over some signature σ and $0 \le n$. A program Π of A-Prolog can be viewed as a specification given to a rational agent for constructing beliefs about possible states of the world. Technically these beliefs are captured by the notion of answer set of a program Π.

Let Π be a program of A-Prolog without variables. For any set S of literals, let Π^S be the program obtained from Π by deleting

- Each rule that has an occurrence of *not* l in its body with $l \in S$
- All occurrences of *not* l in the bodies of the remaining rules

Clearly, Π^S doesn't contain *not* and hence can be viewed as a program of Basic Prolog with the set of consequences $Cn(\Pi^S)$. We say that S is an answer set of Π if

7. $S = Cn(\Pi^S)$

Let S be an answer set of Π. As before, literal l is true in S if $l \in S$; false in S if $\neg l \in S$. This is expanded to conjunctions and disjunctions of literals (and possibly other formulas) in a standard way. We say that formula Q is entailed by a program Π ($\Pi \models Q$) if Q is true in all answer sets of Π. Let query Q be a conjunction $l_1 \wedge \ldots \wedge l_n$ of ground literals. Π's answer to Q is “Yes” if $\Pi \models Q$; “No” if $\Pi \models \neg Q$ ($\neg Q = \neg l_1 \vee \ldots \vee \neg l_n$); “Unknown” otherwise.

Here are some examples. Assume that signature σ contains two object constants a and b. The program Π_1 consisting of the rules

$\neg p(X) : -not\ q(X)$
$q(a)$

has the unique answer set $S = \{q(a), \neg p(b)\}$. The program Π_2:

$p(a) : -not\ p(b)$
$p(b) : -not\ p(a)$

has two answer sets, $\{p(a)\}$ and $\{p(b)\}$. The program Π_3

$p(a) :- not\ p(a)$

has no answer sets.

Programs which have a consistent answer set are called *consistent.* It can be shown that if program is consistent then so are all of its answer sets.

It is easy to see that programs of A-Prolog are *nonmonotonic,* that is, addition of new facts or rules may force the program to withdraw its previous conclusion. This happens, for instance, if we expand the program Π_1 above by a new fact $q(b)$. The new program does not entail $\neg p(b)$ while Π_1 does. Nonmonotonicity of its entailment relation makes A-Prolog and other logic programming formalisms which include negation as failure suitable for formalization of commonsense reasoning which is inherently nonmonotonic: new information constantly forces us to withdraw previous conclusions. This contrasts sharply with classical logic which formalizes mathematical reasoning: a theorem remains proven even if the original set of axioms of the correspond mathematical theory is expanded by new axioms. To learn more about relevance of nonmonotonic reasoning to artificial intelligence and about advances in the development of mathematical theory of nonmonotonic logics, the reader can consult Refs. 16 and 17.

Programs of A-Prolog not containing the connective $\neg$ are called *general logic programs;* answer sets of a general logic program Π are called *stable models* (13) of Π. This class of programs and its subclasses were extensively studied in the last decade. We mention two of such subclasses: stratified and acyclic programs. Stratified programs are general logic programs which do not contain recursion through negation. To give a precise definition we need a notion of the dependency graph G_Π of a program Π. Vertices of G_Π correspond to the predicate symbols of Π. If p_i is a predicate symbol occurring in the head of a rule r from Π and p_j is a predicate symbol occurring in the body of r, then G_Π has an edge from p_i to p_j. This edge is labeled by $-$ if there is an occurrence of p_j in r which belongs to the scope of *not.* If there is an occurrence of p_i in r which does not belong to the scope of *not* then the corresponding edge is labeled by $+$. (Notice that an edge in G_Π can have two labels $+$ and $-$.) A cycle in G_Π is called negative if it contains at least one edge which has a negative label. A program is called *stratified* if its dependency graph has no negative cycles (18). As follows from (13,18), a stratified program has exactly one stable model. Stratified programs play an especially important role in deductive databases where they are used as the basis for a query language called Stratified Datalog. A modification of the bottom-up evaluation procedure described above can be naturally adopted to answer queries in this language (7). A top-down query answering method based on SLD resolution has also been adopted to work for general logic programs. The resulting inference engine is called SLDNF resolution. [For a detailed description, see (19,20)]. When an interpreter, implementing this engine, reaches a goal of the form *not* q, it checks if q contains uninstantiated variables. If it does then the interpreter *flounders.* In this case, no reasonable answer can be given to the original query. Otherwise the interpreter starts an attempt to prove q. If the attempt (finitely) fails then the goal *not* q succeeds. Otherwise, it fails. A notion of *mode* (21) which indicates what parameters of a relation should be instantiated to guarantee the correct behavior of the interpreter greatly facilitates the process of writing programs that avoid floundering. (Because of efficiency considerations actual implementations of SLDNF frequently do not contain the check for floundering, which makes the use of modes even more important.) The above inference is sound, with respect to the stable model semantics, but is, of course, incomplete.

Acyclic programs form another interesting subclass of general logic programs. A program Π is called *acyclic* if there is a function f from ground atoms of the language of Π into natural numbers such that for any rule $r \in ground(\Pi)$ of the form (6), $f(l_0) > f(l_i)$ for any $1 \le i \le n$. A theorem from (22) guarantees that an acyclic program has a unique recursive stable model; that this model determines semantics of the program which coincides with all the semantics for negation as failure mentioned above; and that for nonfloundering queries SLDNF resolution is sound and complete with respect to all these semantics. Another interesting area of research is related to complexity and expressibility of logic programs. Consider, for instance, a decision problem formulated as follows: given a finite propositional general logic program Π and a ground literal l, determine whether l is a consequence of Π. It can be shown (23) that for stratified programs this problem is $O(|\Pi|)$. (Here $|\Pi|$ stands for the number of rules in Π.) For programs of A-Prolog not containing $\neg$ the problem is co-NP complete (24).

In the case of finite general logic program with variables over signature σ (sometimes called finite predicate logic program) it is natural to attempt to characterize classes of sets of ground terms which can be defined by such programs. Among other results the authors in (25,26) show that a set of natural numbers is Π^1_1 definable iff it is definable by a predicate logic program under the stable model semantics. Ref. 27 shows that a set of natural numbers is definable by a stratified logic program iff it is definable by a first-order formula. A survey of recent results can be found in Ref. 28.

HISTORY

We conclude by a short historical overview. The use of logic based languages for representing declarative knowledge was proposed by McCarthy (29). Early application of this idea was tried by Green (30), who combined it with advances in automatic theorem proving, in particular, Robinson's resolution. A view of computation as controlled deduction was advocated by Hayes (31). Credit for founding a field of logic programming is usually given to Kowalski and Colmerauer, whose early work on the subject was done in the mid-1970s (1,32,33). Kowalski formulated the procedural interpretation of Horn clauses and a view of logic programming expressed by his famous equation Algorithm = Logic + Control. Later van Emden and Kowalski developed a formal semantics of logic programming and showed that operational, model-theoretic and fix-point semantics are the same. Colmerauer and his group designed the first Prolog interpreter and applied Prolog to solutions of nontrivial problems in natural language processing. This work was influenced by the developments in theorem proving, as well as in compiler construction. Warren and his colleagues developed the first efficient implementation of Prolog. Prolog is still the most widely used logic programming language. Its users number in the hundreds of thousands. It is used as a rapid-prototyping language and for symbol-manipulation tasks, such as writing compilers, natural language processing systems, knowledge intensive applications of various types, expert systems, and so forth. There are parallel logic programming systems that exploit natural parallelism of Prolog. Constraint Logic Programming systems (34) extend "classical" logic programming by allowing additional conditions on terms. These conditions are expressed by constraints, that is, equations, inequations, and so forth. Constraint logic programming combines resolution with special purpose constraint solving algorithms. Disjunctive and abductive logic programming (35,36) attempt to expand the types of reasoning allowed in programming languages. Finally, Inductive Logic Programming combines ideas from logic programming and machine learning (37).

BIBLIOGRAPHY

1. R. Kowalski, Predicate logic as a programming language, *Proc. Stockholm, Sweden IFIP-74 Congr.,* Elsevier, 1974, pp. 569–574.
2. R. Reiter, On closed world data bases, in H. Gallaire and J. Minker (eds.), *Logic and Data Bases,* New York: Plenum, 1978, pp. 119–140.
3. J. Herbrand, *Logical Writings,* Dordrecht, Holland: Reidel, 1971.
4. J. A. Robinson, A machine oriented logic based on the resolution principle, *J. ACM,* **12**: 23–41, 1965.
5. W. Chen, T. Swift, and D. Warren, Efficient top-down computation of queries under the well-founded semantics, *J. Log. Program.,* **24** (3): 161–201, 1995.
6. M. van Emden and R. Kowalski, The semantics of predicate logic as a programming language, *J. ACM,* **23** (4): 733–742, 1976.
7. S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases,* Reading, MA: Addison-Wesley, 1998.
8. A. Martelli and U. Montanari, An efficient unification algorithm, *ACM Trans. Program. Lang. Syst.,* **4** (2): 258–282, 1982.
9. K. Clark, Negation as failure, in H. Gallaire and J. Minker (eds.), *Logic and Data Bases,* New York: Plenum, 1978, pp. 293–322.
10. M. Fitting, A kripke-kleene semantics for logic programs, *J. Log. Program.,* **2** (4): 295–312, 1985.
11. K. Kunen, Negation in logic programming, *J. Log. Program.,* **4** (4): 289–308, 1987.
12. A. Van Gelder, K. Ross, and J. Schlipf, The well-founded semantics for general logic programs, *J. ACM,* **38** (3): 620–650, 1991.
13. M. Gelfond and V. Lifschitz, The stable model semantics for logic programming, in R. Kowalski and K. Bowen (eds.), *Logic Programming: Proc. Fifth Int. Conf. and Symp.,* Seattle, WA, 1988, pp. 1070–1080.
14. M. Gelfond and V. Lifschitz, Classical negation in logic programs and disjunctive databases, *New Gener. Comput.,* **9** (3–4): 365–385, 1991.
15. K. Apt and R. Bol, Logic programming and negation: A survey, *J. Log. Program.,* **12**: 9–71, 1994.
16. V. W. Marek and M. Truszczynski, *Nonmonotonic Logics: Context-Dependent Reasoning,* Berlin: Springer-Verlag, 1993.
17. C. Baral and M. Gelfond, Logic programming and knowledge representation, *J. Log. Program.,* **12**: 1–80, 1994.
18. K. Apt, H. Blair, and A. Walker, Towards a theory of declarative knowledge, in Jack Minker (ed.), *Foundations of Deductive Databases and Logic Programming,* San Mateo, CA: Morgan Kaufmann, 1988, pp. 89–148.
19. J. Lloyd, *Foundations of Logic Programming,* 2nd ext. ed., Berlin: Springer-Verlag, 1987.
20. V. Lifschitz, Foundations of declarative logic programming, in G. Brewka, (ed.), *Principles of Knowledge Representation,* Stanford, CA: CSLI Publications, 1996, pp. 69–128.
21. P. Dembinski and J. Maluszynski, And-parallelism with intelligent backtracking for annotated logic programs, in V. Saraswat and K. Ueda (eds.), *Proc. Int. Symp. Logic Programming,* 1985, pp. 25–38.
22. K. Apt and M. Bezem, Acyclic programs, in D. Warren and P. Szeredi (eds.), *Logic Programming: Proc. Seventh Int. Conf.,* Jerusalem, 1990, pp. 617–633.
23. W. F. Dowling and J. H. Gallier, Linear time algorithms for testing the satisfiability of propositional horn formulae, *J. Log. Program.,* **1**: 267–284, 1984.
24. W. Marek and M. Truszczynski, Autoepistemic logic, *J. ACM,* **3** (38): 588–619, 1991.
25. J. Schlipf, The expressive power of the logic programming semantics, in *Proc. 9th Symp. Principles of Database Systems,* Nashville, 1990, pp. 196–204.
26. W. Marek, A. Nerod, and J. Remmel, How complicated is the set of stable models of a recursive logic program?, *Ann. Pure and Appl. Logic,* **56**: 119–135, 1992.
27. K. Apt and H. Blair, Arithmetic classification of perfect models of stratified programs, *Fundamenta Informatica,* **13** (1): 1–18, 1990.
28. P. Dantsin et al., Complexity and expressive power of logic programming, *Proc. 12th IEEE Conf. Computational Complexity,* Ulm, Germany, 1997, pp. 1–20.
29. J. McCarthy, Programs with common sense, *Proc. Teddington Conf. Mechanization Thought Processes,* London: Her Majesty's Stationery Office, 1959, pp. 75–91.
30. C. Green, Theorem-proving by resolution as a basis for question-answering system, *Mach. Intelligence,* **4**: 183–205, 1969.
31. P. Hayes, Computation and deduction, *Proc. 2nd MFCS Symp.,* Strßske Pleso, Czechoslovakia, 1973, pp. 105–118.
32. R. A. Kowalski, *Logic for Problem Solving,* New York: Elsevier North Holland, 1979.
33. A. Colmerauer et al., Un systeme de communication homme-machine en francais, Technical report, Groupe de Intelligence Artificielle Universitae de Aix-Marseille, 1973.
34. J. Jaffar and M. Maher, Constraint logic programming: A survey. *J. Log. Program.,* **12**: 503–583, 1994.
35. J. Lobo, J. Minker, and A. Rajasekar, *Foundations of Disjunctive Logic Programming,* Cambridge, MA: MIT Press, 1992.
36. A. C. Kakas, R. A. Kowalski, and F. Toni, Abductive logic programming, *J. Logic and Computation,* **2** (6): 719–771, 1993.
37. F. Bergadano and D. Gunetti, *Inductive Logic Programming,* Cambridge, MA: MIT Press, 1996.

MICHAEL GELFOND
University of Texas at El Paso