An Arbitrary First Order Theory
Can Be Represented by a Logic Program: a Theorem

Olga Kosheleva¹,²

¹ Department of Electrical and Computer Engineering
and ²Knowledge Representation Laboratory
University of Texas at El Paso
El Paso, TX 79968
email: olga@ece.utep.edu

Abstract

How can we represent knowledge inside a computer? For formalized knowledge, classical logic seems to be the most adequate tool. Classical logic is behind all formalisms of classical mathematics, and behind many formalisms used in Artificial Intelligence.

There is only one serious problem with classical logic: due to the famous Gödel's theorem, classical logic is algorithmically undecidable; as a result, when the knowledge is represented in the form of logical statements, it is very difficult to check whether, based on this statement, a given query is true or not.

To make knowledge representations more algorithmic, a special field of logic programming was invented. An important portion of logic programming is algorithmically decidable. To cover knowledge that cannot be represented in this portion, several extensions of the decidable fragments have been proposed. In the spirit of logic programming, these extensions are usually introduced in such a way that even if a general algorithm is not available, good heuristic methods exist.

It is important to check whether the already proposed extensions are sufficient, or further extensions is necessary. In the present paper, we show that one particular extension, namely, logic programming with classical negation, introduced by M. Gelfond and V. Lifschitz, can represent (in some reasonable sense) an arbitrary first order logical theory.

1 Introduction

Intelligent data processing is extremely important in space applications. One of the main problems with space-related data processing is that the amount of data grows so fast that, by some estimates, only about 10% of the data is being processed.

We humans also get lots of information, but our brain is accustomed to filtering out the irrelevant information and processing only the relevant one. To use this experience, we need to use intelligent data processing techniques.

For that, we must be able to represent our knowledge in the computer in such a way that we will be able to use this knowledge for processing data.

Classical logic is the natural way of representing human knowledge, but classical logic is non-algorithmic. How can we represent knowledge inside a computer?

For formalized knowledge, the most adequate tool seems to be classical logic (see, e.g., [6, 2, 1]). Classical logic is behind all formalisms of classical mathematics, and behind many formalisms used in Artificial Intelligence.

There is only one serious problem with classical logic: due to the famous Gödel's theorem, classical logic is algorithmically undecidable; as a result, when the knowledge is represented in the form of logical statements, it is very difficult to check whether, based on this statement, a given query is true or not.

Logic programming: an attempt to make logic algorithmic. To make knowledge representations more algorithmic, a special field of logic programming was invented.

Extensions of traditional logic programming. An important portion of logic programming is algorithmically decidable.
To cover knowledge that cannot be represented in this portion, several extensions of the decidable fragments have been proposed.

In the spirit of logic programming, these extensions are usually introduced in such a way that even if a general algorithm is not available, good heuristic methods exist.

**An important problem: are the existing extensions sufficient?** It is important to check whether the already proposed extensions are sufficient, or further extensions is necessary.

**What we are planning to do.** In the present paper, we show that one particular extension, namely, logic programming with classical negation, introduced by M. Gel'fond and V. Lifschitz [3, 4], can represent (in some reasonable sense) an arbitrary first order logical theory.

Moreover, we will capitalize on the fact that logic programming can describe transitive closure that cannot be represented in traditional first order logic, and show that this logic programming formalism can describe extensions of first order theories obtained by adding this notion of a transitive closure.

The preliminary results of this paper first appeared as a draft [5].

**The structure of this paper.** To make this result more accessible to general readers, we will briefly recall the main definitions of classical logic and of logic programming with classical negation.

# 2 Basic definitions

## 2.1 Classical (first-order) logic: a reminder

**Definition 1.** Suppose that we are given three sets $C$, $V$, and $P$ with $|C| \leq \mathbb{N}$, $|V| \leq \mathbb{N}$, and $|P| \leq \mathbb{N}$, and a function $\text{ar}$ from $P$ to the set $\mathbb{N}$ of non-negative integers.

- Elements of the set $C$ will be called constants and denoted by $c_1, \ldots, c_n, \ldots$
- Elements of the set $V$ are called variables and denoted by $x_1, \ldots, x_n, \ldots$
- Elements of $P$ will be called predicate symbols and denoted by $P_1, \ldots, P_n$.
- The value $\text{ar}(P_i)$ will be called the arity of a predicate $P_i$:
  - a predicate of arity 1 is called unary;
  - a predicate of arity 2 is called binary;
  - a predicate of arity 3 is called ternary;
  - etc.
- By an atom, we mean an expression of the type $P(x_1, \ldots, y)$, where $P \in P$, each of the symbols $x_1, \ldots, y$ is either a constant or a variable, and the number of these symbols $x_1, \ldots, y$ coincides with the arity of the predicate symbol $P$.
- If all the symbols $x_1, \ldots, y$ in the definition of an atom are constants, then this atom is called a ground atom.
- By a first order formula we mean a closed formula $A$ that is formed from atoms by using logical connective ($\lor, \land, \neg, \rightarrow, \equiv$) and quantifiers $\forall x_i$ and $\exists x_i$.
- By a first order theory $T$ we will mean a finite set of first order formulas $\{A_1, \ldots, A_t\}$.

**Definition 2.** For an arbitrary first order theory, we can define a model as a set $U$ (called a Universe), and relations $\bar{P}_i$ on this set $U$ (for all $P_i$ that occur in $T$) that satisfy all the formulas $A_j$ from the theory $T$.

We say that a formula $F$ follows from $T$, and denote it $T \models F$, if $F$ is true in all models of $T$. 

432
2.2 Adding transitive closure (TC) to the first order logic

Definition 3.

- By a TC-formula we mean either a (closed) first order formula, or an expression of the type \( TC(P_i, P_j) \), where \( P_i \) and \( P_j \) are binary predicates.
- By a TC-theory we mean a finite set of TC-formulas \( \{ A_1, \ldots, A_t \} \).
- For a TC-theory \( T \), by its first order part \( \mathcal{T} \), we mean the set of all first order formulas from \( T \).
- By a model of a TC-theory \( T \) we mean such a model of \( \mathcal{T} \), that if \( TC(P_i, P_j) \in T \), then \( P_i \) is a transitive closure of \( P_j \).
- If a formula \( F \) is true in all models of a TC-theory \( T \), then we say that \( F \) follows from \( T \), and denote it by \( T \vdash F \).

2.3 Facts and queries

Motivations. Each theory represents a general description of the objects that we are interested in. E.g., it may describe a linear ordering. To be more specific, we must add some knowledge about our specific object. This knowledge is usually presented in the form of facts, i.e., atomic statements.

After we add this knowledge, we may ask whether some basic statement is true for the resulting theory or not. So, we arrive at the following definition:

Definition 4.

- By a fact we mean a ground atom or its negation. Facts will be denoted by \( F_1, \ldots, F_n, \ldots \).
- By a query we also mean a ground atom or its negation. Queries will be denoted by \( Q \).

Comment. In logic, what we call a fact, or a query, is usually called a literal.

2.4 Definitions of generalized logic programs: a reminder

In the present paper, we consider logic program with classical negation in the sense of \([3, 4]\).

We want to formulate logic programs that are equivalent to first order theories. It turns out that for that purpose, we must use additional (auxiliary) constants, predicates and functional symbols. So, we arrive at the following definitions:

Definition 5. Suppose that in addition to the sets \( \mathcal{C}, \mathcal{V}, \) and \( \mathcal{P} \), we have denumerable sets \( \mathcal{R}, \mathcal{B}, \) and \( \mathcal{S} \), and a function \( \text{arity}: \mathcal{F} \rightarrow \mathbb{N} \) such that for every \( n \in \mathbb{N} \), there are infinitely many \( f \in \mathcal{S} \) with \( \text{arity}(f) = n \).

- Elements of the set \( \mathcal{B} \) will be called auxiliary constants and denoted by \( b_1, \ldots, b_n, \ldots \).
- Elements of the set \( \mathcal{R} \) will be called auxiliary predicates and denoted by \( R_1, \ldots, R_n, \ldots \).
- Elements of \( \mathcal{F} \) will be called auxiliary functional symbols and denoted by \( g_1, \ldots, g_n, \ldots \). For each \( f \in \mathcal{F} \), the value \( \text{arity}(f) \) is called an arity off.
- A term is defined in the usual manner, starting from constants, auxiliary constants and variables, and applying function symbols of appropriate arity.
- By a generalized atom we mean an expression of the type \( P(t_1, \ldots, t_n) \), where \( P \in \mathcal{P} \cup \mathcal{R} \) is a predicate or auxiliary predicate of arity \( n \), and \( t_i \) are terms.
- A generalized literal is a generalized atom \( p \) or the expression of the type \( \neg p \), where \( p \) is a generalized atom; an expression \( \neg p \) is called classical \( \neg \) negation.
- A rule is an expression of the type \( A \leftarrow B_1, \ldots, B_m \), where \( A \) is a generalized literal, \( m \geq 0 \), and each of \( B_i \) is either a generalized literal, or an expression of \( \mathcal{F} \) type \( \neg p \) for some generalized literal \( p \).
- Rules with \( m = 0 \) are called facts. A fact \( A \leftarrow \) can also be written as \( A \).
- A finite set of rules is called a generalized logic program, or a logic program with classical negation. Such programs will be denoted by \( P, P_1, \) etc.
We say that a query $Q$ is true for a program $P$ (and denote it by $P \models Q$) if $Q$ belongs to any consistent answer set of $P$ (in the sense of [3,4]).

Comments

- Please note that in the formulation of the query, we only allow the symbols from the original theory, auxiliary symbols are not allowed.
- Since in this paper, we will only use logic programs with classical negation, we will call them, without confusion, simply logic programs.

3. Main result

**Theorem.** There exists an algorithm that transform every TC-theory $T$ into a logic program $P_T$ with classical negation so that for an arbitrary finite set of facts $\{F_1, \ldots, F_n\}$, and for an arbitrary query $Q$, $Q$ is true in $T + \{F_1, \ldots, F_n\}$ if and only if $Q$ is true in $P_T + \{F_1, \ldots, F_n\}$.

Comment. We would like to emphasize once again that we allow the use of auxiliary predicates, constants and function symbols while describing the rules of the logic program, but not in queries or facts. So, in this Theorem, we still apply Definitions 4 to describe facts and queries. According to these definitions, facts and queries are ground atoms (or negations of ground atoms) that are formed only from the original predicate symbols $P_i$ and original constants $c_i$.

4. Description of the algorithm and the main idea of the proof

Let us describe the algorithm that transform a theory into a logical program.

4.1. Case of first order theories

At first, we will consider the case when the TC-theory does not contain any statements about the transitive closure, i.e., when it is actually the first order theory. We will illustrate this case on the example of the following theory that describes dense order:

\[
\forall x, y, z (x < y & y < z \rightarrow x < z);
\]

\[
\forall x, y (x < y \rightarrow \neg y < x);
\]

\[
\forall x (\neg x < x);
\]

\[
\forall x, y \exists z (x < z & z < y).
\]

**Step 1: general description.** First we make a skolemization of the axioms of the given first-order theory (for definitions, see, e.g., [6, 2, 1]).

**Step 1: example.** In our example, skolemization leads to the following axioms (universal quantifiers are, for simplicity, omitted):

\[
x < y & y < z \rightarrow x < z;
\]

\[
x < y \rightarrow \neg y < x;
\]

\[
\neg x < x;
\]

\[
x < f(x, y) & f(x, y) < y.
\]

**Step 2: general description.** Every axiom is represented in conjunctive normal form ([6, 2, 1]), and each of the resulting conjunctions is written separately.
Step 2: example. In our example, we will get the following set of disjunctions:

\(-x < y) \vee (-y < z) \vee (x < z); \\
(-x < z); \\
(x < f(x, y)); \\
(f(x, y) < y).

Step 3: general description. On this step, we translate every disjunction \(a \vee \ldots \vee a_n\) into the following \(n\) rules:

\[ a_n \leftarrow \neg a_1, \neg a_2, \ldots, \neg a_{n-1}. \]

\[ a_{n-1} \leftarrow \neg a_1, \neg a_2, \ldots, \neg a_{n-2}, a_n. \]

\[ \ldots \]

\[ a_1 \leftarrow \neg a_2, \ldots, a_n. \]

Step 3: example. In our example, we will get the following program (in classical logic, it is a usual practice to have a predicate symbol like \(<\) in between the arguments, but in logic programming, the predicate symbol is usually in front; to follow this tradition, we will use a notation \(L(x, y)\) instead of \(x < y)\):

\[ L(x, x) \leftarrow L(x, y), L(y, z). \]

\[ \neg L(y, x) \leftarrow L(x, y), \neg L(x, z). \]

\[ \neg L(x, y) \leftarrow L(y, z), \neg L(x, z). \]

\[ \neg L(x, y) \leftarrow L(y, x). \]

\[ \neg L(y, x) \leftarrow L(x, y). \]

\[ \neg L(x, x). \]

\[ L(x, f(x, y)). \]

\[ L(f(x, y), y). \]

Step 4: general description. Finally, to obtain a program \(P_T\) that is “equivalent” to the original theory \(T\) (in the sense of Theorem 1) we add, for each of the predicates \(P(x, \ldots, y)\) from the resulting program, two statements called Closed World Assumption (CWA):

\[ P(x, \ldots, y) \leftarrow \neg P(x, \ldots, y). \]

\[ -P(x, \ldots, y) \leftarrow \neg P(x, \ldots, y). \]

Step 4: example. In particular, in our example, we add the following two statements:

\[ L(x, y) \leftarrow \neg L(x, y). \]

\[ \neg L(x, y) \leftarrow \neg L(x, y). \]

Idea of the proof. We need to prove that for any finite set of facts \(\{F_1, \ldots, F_n\}\), and for an arbitrary query \(Q\), \(Q\) is true in \(T + \{F_1, \ldots, F_n\}\) if and only if \(Q\) is true in \(\neg Q\) is inconsistent.

For classical logic, \(Q\) is true in \(T + \{F_1, \ldots, F_n\}\) if and only if \(T + \{F_1, \ldots, F_n\}\) is inconsistent. The inconsistency of the theory is equivalent to the inconsistency of its skolemization, so, it is sufficient to check whether the skolemized version \(S(T)\) is inconsistent, i.e., whether \(Q\) is deducible from the theory \(T' = S(T) + \{F_1, \ldots, F_n\}\), i.e., whether \(Q\) is true in all models of \(T'\). It is sufficient to consider Herbrand models of \(T'\).

It is easy to show that every Herbrand model of \(T'\) is a consistent answer set of the corresponding logic program (minimality follows from the present of the two closed world assumptions), and vice versa, every consistent answer set represents a Herbrand model of \(T'\). This observation concludes the proof.
4.2 Theories with transitive closure

If a theory $T$ contains statements about transitive closure, then we need to add the following additional step to our algorithm:

**Step 5.** If the original theory $T$ contains the expression $T \mathcal{C}(A,B)$ for some binary predicate symbols $A$ and $B$, then we:

- add an auxiliary predicate $\alpha_{AB}$ to the set $\mathcal{R}$ of auxiliary predicates; and
- add the following rules to the logic program obtained on Step 4:

$$
\alpha_{AB}(x, y) \leftarrow B(x, y) \\
\alpha_{AB}(x, y) \leftarrow B(x, y), \alpha_{AB}(y, z) \\
A(x, y) \leftarrow \alpha_{AB}(x, y) \\
A(x, y) \leftarrow \lnot \alpha_{AB}(x, y) \\
\lnot \alpha_{AB}(x, y) \leftarrow \lnot \alpha_{AB}(x, y)
$$

**Comment.** It is easy to show that the "standard" way of representing transitive closure in logic programming will not work. Indeed, traditionally, the fact that predicate $\text{anc}$ (ancestor) is a transitive closure of the predicate $\text{par}$ (parent) is expressed as follows:

$$
\text{anc}(x, y) \leftarrow \text{par}(x, y) \\
\text{anc}(x, y) \leftarrow \text{par}(x, z), \text{anc}(z, y) \\
\lnot \text{anc}(x, y) \leftarrow \lnot \text{anc}(z, y)
$$

However, if we add the facts

$$
\lnot \text{par}(a_i, a_j) \leftarrow
$$

for all $i, j$, and

$$
\text{anc}(a_1, a_2) \leftarrow
$$

then we get $P + F \models \lnot \text{anc}(a_1, a_2)$, but in this model, the transitive closure $\text{par}$ is empty, and is, therefore, different from $\text{ans}$.

4.3 General comment

The proof given above shows that the correspondence between theories and logic programs is even more straightforward than follows from our Theorem: Namely, if we add a new axiom $A_{i+1}$ to the theory $T$, then a logic program that corresponds to the resulting theory $T'$, can be obtained from $T$ by adding rules that correspond to $A_{i+1}$.

**Acknowledgments.** This work was partly supported by the NASA Pan American Center for Environmental and Earth Studies (PACES). The author is thankful to Chitta Baral, Ann Gates, Michael Gelfond, Vladik Kreinovich, Luc Longpré, Arthur Ramer, and Scott Starks for their help and encouragement.

**References**


