Reasoning in Answer Set Prolog

Chitta Baral & Michael Gelfond

Outline

- **Introduction**
- **Defaults in AnsProlog**
- **Syntax and Semantics**
- **Inference Engines**
- **Answer Set Programming**
- **Modeling Change**

Motivating Example 1

Story: On the first of Dec John and Bob met in Paris. Immediately after the meeting John took the plane from Paris to Baghdad.

Question: Where should one look for John and Bob on Dec 2nd?

Answer: John - Baghdad, Bob - Paris.

Among other things to get this answer we need to know that action of "John traveling to Baghdad" is normally successful and causes change in the whereabouts of John but not in those of Bob.

Motivating Example 2

Story: On the first of Dec John and Bob met in Paris. Immediately after the meeting John took the plane from Paris to Baghdad.

New Info: On the way the plane stopped in Rome where John was arrested.

Question: Where should one look for John and Bob on Dec 2nd?

The new information forced us to withdraw the previous conclusion. (i.e. our reasoning is non-monotonic).

How to automate such reasoning?

Basic Idea:

- **Use declarative language to describe the domain.**
- **Express various questions as formal queries to the resulting program.**
- **Use inference engine, i.e. a collection of reasoning algorithms, to answer these queries.**

Language Requirements

1. **Allow reasoning about defaults, causal reasoning, epistemic reasoning, etc.**
2. **Allow elaboration tolerant representations of knowledge - small additions to informal body of knowledge should correspond to small additions to the formal KB.**
3. **Have a methodology of representing knowledge.**
4. **Mathematical properties of the language should be well understood.**
5. **Inference engines should be reasonably efficient.**

Problem: What Language and What Inference?

CANDIDATES:

- **First-Order Language and Classical Logic.**
- **Super-Classical Non-monotonic Logics: Circumscription, Default Logic, Epistemic Logics.**
- **Answer Set Prolog**

Attempts to express "normally" in other languages

- T_1: **Bird's normally fly. Tweety is a Bird.**
- T_2: **Penguins are birds. Penguin's do not fly. Tweety is a penguin.**
- **How to represent T_1 and T_2 so that one can conclude 'Tweety flies' from T_1, while one does not conclude it (and in fact concludes 'Tweety does not fly') from $T_1 \cup T_2$.**

A first attempt using first order logic

- T_1': $\forall X.bird(X) \Rightarrow fly(X)$

  $bird(tweety)$

- T_2': $\forall X.penguin(X) \Rightarrow bird(X)$

  $\forall X.penguin(X) \Rightarrow \neg fly(X)$

  $penguin(tweety)$.

- $T_1' \models fly(tweety)$.

  But $T_1' \cup T_2' \models fly(tweety)$, **and** $T_1' \cup T_2' \models \neg fly(tweety)$.

Another attempt using (an extension of) first order logic

- T_1'': $\forall X.bird(X) \wedge \neg ab(X) \Rightarrow fly(X)$

  $bird(tweety)$

- T_2'': $\forall X.penguin(X) \Rightarrow bird(X)$

  $\forall X.penguin(X) \Rightarrow \neg fly(X)$

  $penguin(tweety)$.

- **Need careful minimization of ab. (not easy, one needs to worry about unintended impact of minimization on other predicates.)**

- **A quote from Brachman and Levesque, 2004 after a laborious attempt in formalizing this example in pages 215-222:**

  This means that there is a serious limitation in using circumscription for default reasoning.

Some history

Answer Set Prolog (AnsProlog) was introduced around 1990 by V. Lifschitz and M. Gelfond. It is rooted in research on the semantics of negation as failure in 'classical' Prolog (Clark, Apt, Blair) and in the work on nonmonotonic logics, especially that on Autoepistemic Logic of R. Moore and Default Logic of R. Reiter.

Several groups in Europe and the USA developed rather efficient inference engines for computing answer sets of (finite) logic programs.

Informal Example: Defaults

- **Defaults are statements containing words "normally, typically, as a rule".**
- **A large part of our education seems to consists of learning various defaults, their exceptions, and the skill of reasoning with them.**
- **Defaults do not occur in the language of mathematics, but play very important role in everyday, commonsense reasoning.**

Example: Family Problems

Suppose you are Sam's teacher and you strongly believe that to pass the class Sam needs some extra help. You convey this information to Sam's father, John, and expect some actions on his part. Your reasoning probably goes along these lines:

John is Sam's parent.

NORMALLY, parents care about their children.

Therefore John cares about Sam and will help him with his study.

The second statement is a typical example of a default.

About Caring

To model this reasoning we introduce relation

$cares(X, Y)$ — X **cares for** Y**, and theory** Π_0**:**

$father(john, sam).$ $\quad mother(mary, sam).$

$parent(X, Y) \leftarrow father(X, Y).$

$parent(X, Y) \leftarrow mother(X, Y).$

$child(X, Y) \leftarrow parent(Y, X).$

$cares(X, Y) \leftarrow parent(X, Y).$ **(Ignoring "normally")**

Informal Semantics

- **Program Π can be viewed as specification for sets of beliefs of a rational reasoner associated with Π.**
- **Beliefs are represented by consistent sets of literals, called answer sets, which must satisfy the rules and the Rationality Principle which says:**

"Believe nothing you are not forced to believe".

Answer Set A of Π_0:

$father(john, sam).$ $mother(mary, sam).$

$parent(john, sam).$ $parent(mary, sam).$

$child(sam, john).$ $child(sam, mary).$

$cares(john, sam).$ $cares(mary, sam).$

Answer to query q is YES if $q \in A$, NO if $\neg q \in A$, UNKNOWN otherwise.

$cares(john, sam)?$ **YES**

$father(mary, sam)?$ **UNKNOWN**

Adding Negative Information:

$\neg father(X_1, Y) \leftarrow father(X_2, Y), X_1 <> X_2$

The new answer set is

$father(john, sam).$ $mother(mary, sam).$

$\neg father(mary, sam).$ $\neg father(sam, sam) \ldots$

$parent(john, sam).$ $parent(mary, sam).$

$child(sam, john).$ $child(sam, mary).$

$cares(john, sam).$ $cares(mary, sam).$

$cares(john, sam)?$ **YES** $father(mary, sam)?$ **NO**

$father(mary, john)$ **is still UNKNOWN**

About Caring — Non-Monotonicity

Assume now that in addition to the default

1. "normally parents care about their children"

you learn that

2. "John is an exception to this rule. He does not care about his children."

In everyday reasoning this new information does not cause contradiction. We simply withdraw our previous conclusion, $cares(john, sam)$, and replace it by the new one, $\neg cares(john, sam)$. (Need non-monotonicity)

Representing Defaults in AnsProlog

To reason with defaults we need a new logical connective not **called Default Negation.**

$not\ p$ **says "no reason to believe** p**".**

$\neg p$ **says "**p **is false".**

Program $p \leftarrow\ not\ q$ **has one answer set** $\{p\}$**.**

Representing Defaults in AnsProlog

In Answer Set Prolog a default "Normally elements of class C have property P" is often represented by a rule:

$$p(X) \leftarrow c(X),$$
$$not\ ab(d, X),$$
$$not\ \neg p(X).$$

d is the default's name (given by the program designer). $ab(d, X)$ **says that default d is not applicable to X;** $not\ \neg p(X)$ **is read as '$p(X)$ MAY be true'.**

Example: Caring Parents

Default "normally parents care about their children" will be represented by the rule (d1):

$$cares(X, Y) \leftarrow parent(X, Y),$$
$$not\ ab(d1, X, Y),$$
$$not\ \neg cares(X, Y).$$

If all our program knows about John is

$father(john, sam)$

it will conclude $cares(john, sam)$**.**

Uncaring John

What happens when we learn that John does not care about his children? There is no way to incorporate this information into classical representation of the story without changing some of its axioms.

We can however add it to the AnsProlog program using the rule:

$$\neg cares(john, X) \leftarrow child(X, john).$$

The new program is consistent and entails

$\neg cares(john, sam)$**,** $cares(mary, sam)$**.**

Exceptions to Defaults

A default d = "Normally elements of c have property p" may have two types of exceptions:

- **"strong" - refute the default's conclusion**

(Birds normally fly but penguins do not.)

- **"weak" - render the default inapplicable.**

(Wounded birds may or may not fly.)

Representing Exceptions

A weak exception $e(X)$ to d is encoded by a so called CANCELATION axiom

$$ab(d, X) \leftarrow \ not\ \neg e(X).$$

which says that d is not applicable to X if X MAY BE a weak exception to d.

If e is a strong exception we need one more rule,

$$\neg p(X) \leftarrow e(X)$$

which will allow us to defeat d's conclusion.

Weak Exception - an Example

To illustrate the notion of weak exception let us emulate a cautious reasoner who does not want to apply default (d1) to people whose spouses do not care about their children. Such a reasoner will prefer not to make any judgment on Mary's relation to Sam. This can be achieved by a rule:

$$\begin{aligned} ab(d1, P1, C) \leftarrow\ & parent(P1, C), \\ & parent(P2, C), \\ & \neg cares(P2, C). \end{aligned}$$

New program answers no **to query** $cares(john, sam)$ **and** $maybe$ **to query** $cares(mary, sam)$**.**

Incompleteness in Databases

Consider a database table representing a tentative summer schedule of a CS department.

Professor	Course
mike	pascal
john	c
staff	lisp

"staff" is a special constant (called Null value) which stands for an unknown professor. It expresses the fact that Lisp will be taught by SOME professor (possibly different from Mike and John).

AnsProlog representation

Relation $t(P, C)$ - professor P teaches a course C.

Assume that we are given complete collections of professors and courses.

Positive info from the table

$$t(mike, pascal). \quad t(john, c). \quad t(staff, lisp).$$

To represent negative info we use default d: Normally, P teaches C only if this is listed in the schedule.

AnsProlog Representation

$$\neg t(P, C) \leftarrow prof(P), course(C),$$
$$not\ ab(d, P, C),$$
$$not\ \ t(P, C).$$

The default d shall not be applicable to Lisp, or any other course taught by "staff" - weak exception.

$$ab(d, P, C) \leftarrow t(staff, C).$$

ANSWERS:

?$t(mike, c)$ **-** NO

?$t(mike, lisp)$ **-** $UNKNOWN$

Semantics of Declarative Languages

- **A declarative program (DP) is a collection of statements describing objects of a domain and their properties.**
- **Semantics defines a notion of a model of a DP (i.e. a possible state of the world and/or agent compatible with the DP statements) and characterizes the collection of valid consequences of a program.**

SYNTAX and SEMANTICS of AnsProlog

TERMS over a signature σ are defined as usual.

ATOM – an expression of the form $p(t_1, \ldots, t_n)$.

LITERAL – $p(t_1, \ldots, t_n)$ or $\neg p(t_1, \ldots, t_n)$.

The syntax of AnsProlog

• **A program Π of AnsProlog (sometimes called a knowledge base) consists of a signature σ and a collection of rules of the form (1):**

$$l_0 \; or \; \ldots \; or \; l_i \leftarrow l_{i+1}, \ldots, l_m, \; not \; l_{m+1}, \ldots, \; not \; l_n.$$

where l's are literals of σ.

The left-hand side of a rule is called the Head and the right-hand side the Body. Both, the head and the body can be empty.

The syntax of AnsProlog

Object, function and predicate symbols of σ are denoted by identifiers starting with small letters. Variables are identifiers starting with the capital ones.

Variables of Π range over ground terms of σ. A rule r with variables is viewed as a set of its ground instantiations - rules obtained from r by replacing r's variables by ground terms of σ. This means that it is enough to define the semantics of ground (i.e. not containing variables) programs.

More notation and terminology

A ground set S of literals satisfies the body of rule r

$$l_0 \; or \; \ldots \; or \; l_i \leftarrow l_{i+1}, \ldots, l_m, \; not \; l_{m+1}, \ldots, \; not \; l_n.$$

if $\{l_{i+1}, \ldots, l_m\} \subseteq S$ and $\{l_{m+1}, \ldots, l_n\} \cap S = \emptyset$

S satisfies the head of r if $\{l_0, \ldots, l_i\} \cap S \neq \emptyset$

S satisfies r if whenever S satisfies the body of r it satisfies its head.

Sets $\{p(a)\}$ and $\{p(a), q(b)\}$ satisfy rule

$$p(a) \leftarrow q(b), \; not \; t(c)$$

while set $\{q(b)\}$ does not.

Informal semantics of AnsProlog

• **Ground program Π can be viewed as a specification for the sets of beliefs to be held by a rational reasoner associated with Π. Such sets will be represented by collection of ground literals. In forming such sets the reasoner must:**

1. Satisfy the rules of P.

2. Satisfy the "rationality principle" which says: "Believe nothing you are not forced to believe".

Informal semantics of AnsProlog

Beliefs of Π are represented by sets of ground literals called Answer Sets (Stable Models) of Π, e.g., a program

$$\Pi_0 \begin{cases} p(a) & \leftarrow \quad not\ q(a). \\ p(b) & \leftarrow \quad not\ q(b). \\ q(a). & \end{cases}$$

has one answer set $S_0 = \{q(a), p(b)\}$.

The answer set of

$$\Pi_1 = \Pi_0 \cup \{\neg q(X) \leftarrow \ not\ q(X).\}$$

is $S_1 = \{q(a), \neg q(b), p(b)\}$.

Defining answer sets - Part 1

Let program Π consist of rules of the form:

$$l_0 \; or \; \ldots \; or \; l_i \leftarrow l_{i+1}, \ldots, l_m. \qquad (1)$$

Answer Set of Π is a consistent set S of ground literals such that:

- **S is closed under the rules of Π;**
- **S is minimal i.e. no proper subset of S satisfies the rules of Π.**

Examples

- $p(a) \leftarrow \neg p(b).$ $\qquad\qquad$ $\neg p(a).$

  $A = \{\neg p(a)\}$

- $p(b) \leftarrow \neg p(a).$ $\qquad\qquad$ $\neg p(a).$

  $A = \{\neg p(a), p(b)\}$

- $p(b) \leftarrow \neg p(a).$ $\qquad\qquad$ $p(b) \leftarrow p(a).$

  $A = \{\quad\}$

- $p(a)\ or\ \ p(b).$

Defining answer sets - Part 2

Let Π be an arbitrary program. By Π^S we denote the program obtained from Π by

(i) removing all rules containing $not\ l$ such that $l \in S$;

(ii) removing all other premises containing not .

Definition: S is an Answer Set of Π iff S is an answer set of Π^S.

Example

Π	Π^S	$S = \{q(a), p(b)\}$
$p(a) \leftarrow \ not\ q(a).$		
$p(b) \leftarrow \ not\ q(b).$	$p(b).$	
$q(a).$	$q(a).$	

$\{q(a), p(b)\}$ **is an answer set of** Π

Examples

- $\Pi_0 = \{p(a) \leftarrow \ not\ p(a).\}$ **No answer set.**

- $\Pi_1 = \{p(a) \leftarrow \ not\ p(b). \qquad p(b) \leftarrow \ not\ p(a).\}$

  $A1 = \{p(a)\} \qquad A2 = \{p(b)\}$

- $\Pi_2 = \Pi_1 \cup \{\leftarrow p(b).\}$

  $A = \{p(a)\}$

- $\Pi_3 = \Pi_2 \cup \{\neg p(a).\}$ **No answer set**

Inference Engines

Answer set solvers, *Smodels*, *Dlv*, etc. compute answer sets of logic programs without function symbols.

The problem is NP-complete for programs without disjunction and $\Sigma_2 P$ for arbitrary programs.

The sound and complete algorithms use efficient grounding methods from deductive databases and Davis - Putnam procedure with various heuristics from propositional satisfiability algorithms.

Inference Engines

Answer set solvers *Asset*, *Cmodels*, etc. reduce computation of answer sets to (possibly multiple) calls to satisfiability solvers.

Even though for a large class of problems one call is sufficient in general it cannot be limited.

Since satisfiability solvers were in existence for a long time for non-recursive programs this approach is usually faster.

Inference Engines

Resolution based systems, Prolog, XSB, SLG etc, answer queries of the form $q(X)$.

Allow function symbols but only applicable to programs with one answer set and no serious recursion.

The problem is undecidable and hence the algorithms are incomplete.

Answer Set Programming

- **Model a domain by describing its objects and relations between these objects.**
- **Describe properties of these relations by a program T of AnsProlog.**
- **Reduce a problem to be solved to the problem of finding answer set(s) of T.**
- **Use AnsProlog reasoning systems to find these sets.**

Computing Hamiltonian Paths

Given: Directed graph G, initial vertex s_0. Find a path from s_0 to s_0 which visit each node exactly once.

Graph represented by

$node(s_0)....$ $edge(s_i, s_j)....$ $init(s_0).$

Idea: Represent every HP by a set of atoms of the form

$in(s_0, s_1).$ $...in(s_k, s_0).$

which belongs to an answer set of program Π associated with the problem.

Constructing the program

Describe conditions on a collection P of atoms of the form $in(s1, s2)$ which will make P a HP.

- **P visits every node V at most once:**

$\leftarrow in(V_1, V), in(V2_2, V), V_1 <> V_2$

$\leftarrow in(V, V_1), in(V, V_2), V_1 <> V_2$

- **P visits every node of the graph.**

Introduce relation $reached(V)$ which holds if P visits V on its way from the initial node:

$reached(V_2) \leftarrow init(V_1), in(V_1, V_2).$

$reached(V_2) \leftarrow reached(V_1), in(V_1, V_2).$

The constraint

$\leftarrow \ not\ reached(V).$

guarantees that every node is reached.

To complete the solution we need to make sure that our program will consider every edge of the graph. This can be done by the rule

$in(V_1, V_2)$ **or** $\neg in(V_1, V_2) \leftarrow edge(V_1, V_2)$.

Proposition.

There is one-to-one correspondence between answer sets of Π **and HPs in** G**.**

These models can be computed by systems like *smodels* **or** *dlv***.**

Reasoning in Dynamic domains

Need to describe

1. **transition diagram representing possible trajectories of the system;**
2. **history of observations and action occurrences;**
3. **goals and information about preferred or most promising actions needed to achieve these goals.**

Describing the Diagram - Action Languages

Effects of actions are given by:

- **dynamic causal law:**

$$a \textbf{ causes } f \textbf{ if } p$$

- **state constraint:**

$$f \textbf{ if } p$$

- **impossibility condition:**

$$a \textbf{ impossible\_if } p$$

Semantics is given by defining states and transitions of the diagram (McCain, Turner).

Translation to AnsProlog

a **causes** f **if** p

$$holds(f, T+1) \leftarrow holds(p, T), occurs(a, T).$$

f **if** p

$$holds(f, T) \leftarrow holds(p, T).$$

a **impossible_if** p

$$\leftarrow occurs(a, T), holds(p, T).$$

Translation to AnsProlog

Causal laws describe changes caused by action a.

To describe things that stay the same we use Inertia Axiom - "Things tend to stay as they are".

$holds(F, T+1) \leftarrow holds(F, T),\ not\ \neg holds(F, T+1)$

$\neg holds(F, T+1) \leftarrow \neg holds(F, T),\ not\ holds(F, T+1)$

This is a standard representation of a default.

Computing successor states

Proposition. Let σ_0 be a state of a dynamic domain, a be an action, and

$$h(\sigma_0) = \{holds(f, 0) : f \in \sigma_0\} \cup \{\neg holds(f, 0) : \neg f \in \sigma_0\}$$

Then there is one-to-one correspondence between the successor states of $\langle \sigma_0, a \rangle$ and answer sets of the program

$$\Pi \cup h(\sigma_0) \cup o(a, 0)$$

A COMMONSENSE THEORY OF TRAVEL

- **Basic Objects**
- **Actions and Fluents**
- **Causal Laws**
- **Examples**

The basic objects

A trip may have many participants, use vehicles of different types, and follow complex routes.

It needs to have the name, the origin, the destination, and may have other attributes such as "mode_of_transp", "vehicle_used", etc.

We'll often name trips as $f(C_1 1, C_2)$ - a trip from city C_1 to city C_2.

A trip may be in transit (en_route) or at one of its possible stops.

Travel Documents

```
travel_document(passport).

need(P,passport,F) :-
                crosses_border(F).

crosses_border(F) :-
            origin(F,C1),
            dest(F,C2),
            C1 <> C2.
```

ACTIONS and FLUENTS

ACTIONS:

```
depart(F)  stop(F,D)
go_on(P,F) get(P,TD)  become_participant(P,F)
```

FLUENTS:

```
at(F,L)  at(P,L)
participant(P,F)  has(P,TD)
```

Inertia Axioms

```
h(Fl,T+1) :- T < n,
             h(Fl,T),
             not -h(Fl,T+1).
-h(Fl,T+1) :- T < n,
              -h(Fl,T),
              not h(Fl,T+1).
```

Effects of Actions

```
h(participant(P,F),T+1) :-
            o(become_participant(P,F),T).
h(has(P,TD),T+1) :-
              o(get(P,TD),T).
h(at(P,D),T) :-
           h(participant(P,F),T),
           h(at(F,D),T).
```

Example

- **John is in Boston on Dec 1. He has no passport. Can he go to Paris on Dec. 3?**

- **Translation of first two statements is:**

```
h(at(john,boston),0).   -h(has(john,passport),0).
time(0,d,1).            time(0,m,12).
```

The last statement is translated as:

```
goal(T) :-
     o(go_on(john,f(boston,paris)),T),
     time(T,d,3).time(T,m,12).
```

Example 2

Load the commonsense theory of travel and geography and calendar modules. Since the query has a form $goal(T)$ we load a planning module.

If the resulting program has a model the answer to our query is *yes*. Otherwise it is *no*.

Planning Module

n - length of plan; "goal" and "actor" (John) is obtained from the text.

```
yes :- goal(T).

o(A,T) or -o(A,T) :- actor(john,A),T < n.

:- o(A1,T),o(A2,T),A1 <> A2.
```

Example: modeling RCS

In its simplest form the RCS can be viewed as

(a) a directed graph, with nodes corresponding to tanks, pipe junctions, and jets, and links labeled by valves.

(b) A collection of switches controlling the positions of valves.

Simplified view of the RCS

Facts describing objects of the domain and their connections:

```
tank_of(tank,fwd_rcs).
jet_of(jet,fwd_rcs).
link(tank,junc2,v3).
controls(sw3,v3).
```

Modeling the RCS

A state of the RCS is given by fluents, including:

- $pressurized\_by(N, Tk)$ **- node** N **is pressurized by a tank** Tk
- $in\_state(V, P)$ **- valve** V **is in valve position** P
- $in\_state(Sw, P)$ **- switch** Sw **is in switch position** P

A typical action is:

- $flip(Sw, P)$ **- flip switch** Sw **to position** P

Effects of actions

Direct effect of $flip(Sw, P)$ on in_state:

```
holds(in_state(Sw,P),T+1) :-
                    occurs(flip(Sw,P),T),
                    not stuck(Sw).
```

Indirect effect of $flip(Sw, P)$ on in_state:

```
holds(in_state(V,P),T) :-
                controls(Sw,V),
                holds(in_state(Sw,P),T),
                not stuck(V),
                not bad_circuitry(V).
```

Recursive rules for indirect effects

Definition of indirect effects of $flip(Sw, P)$ on

$pressurized\_by(N, Tk)$ **is recursive:**

```
holds(pressurized_by(Tk,Tk),T) :-
            tank(Tk).
holds(pressurized_by(N1,Tk),T) :-
            link(N2,N1,V),
            holds(in_state(V,open),T),
            holds(pressurized_by(N2,Tk),T).
```

The effect of a single action propagates and affects several fluents.

Planning with AnsProlog

Given:

(a) collection I of RCS faults

(b) goal - state satisfying set G of fluents

Finding a plan for G of max length n can be reduced to finding an answer set of program

$$T \cup I \cup PM \tag{2}$$

where PM is

The Planning Module

```
time(0..n).
goal :-
      holds(maneuver_of(plus_z,left_rcs),T),
      holds(maneuver_of(plus_z,right_rcs),T),
      holds(maneuver_of(plus_z,fwd_rcs),T).
:- not goal.
1{occurs(A,T): action(A)}1 :- not goal(T).
```

Planning module is a VERY small program of AnsProlog!