ANSWER SET PROGRAMMING

KNOWLEDGE REPRESENTATION, REASONING AND DECLARATIVE PROBLEM SOLVING USING

ANSPROLOG*: WWW.BARAL.US/BOOKONE

Chitta Baral

Department of Computer Science and Engg.
Arizona State University
Tempe, AZ 85287
chitta@asu.edu

http://www.public.asu.edu/∼cbaral/

June 2004

Nomenclature

- AnsProlog*: **Pro**gramming in **log**ic with **ans**wer sets
- Also referred to as A-Prolog
- A collection of rules of the form

  L_0 *or* $\ldots$ *or* $L_k \leftarrow L_{k+1}, \ldots, L_m, \textbf{not}\ L_{m+1}, \ldots, \textbf{not}\ L_n.$

  where L_i's are literals in the sense of first order logic.
- The language
  - with syntax encompassing
    - general logic programs (normal logic programs): $k = 0$, L_is are atoms.
    - extended logic programs: $k = 0$.
    - disjunctive logic programs: L_is are atoms, $m = n$.
    - disjunctive normal logic programs: L_is are atoms.
    - extended disjunctive normal logic programs: no restrictions.
  - and semantics characterized with answer sets

MOTIVATION: “INTELLIGENCE” TO “AI” TO “KR” TO “ANSPROLOG”

A dictionary meaning of the word: "intelligence"

(1)

(a) The capacity to acquire and apply knowledge.

(b) The faculty of thought and reason.

(c) Superior powers of mind. See Synonyms at mind.

(2) An intelligent, incorporeal being, especially an angel.

(3) Information; news. See Synonyms at news.

(4)

(a) Secret information, especially about an actual or potential enemy.

(b) An agency, staff, or office employed in gathering such information.

(c) Espionage agents, organizations, and activities considered as a group

Source: The American Heritage Dictionary of the English Language, Fourth Edition

Wordnet meaning of 'intelligence"

n

1: **the ability to comprehend; to understand and profit from experience [ant: stupidity]**

2: a unit responsible for gathering and interpreting intelligence

3: secret information about an enemy (or potential enemy); "we sent out planes to gather intelligence on their radar coverage"

4: new information about specific and timely events; "they awaited news of the outcome" [syn: news, tidings, word]

5: the operation of gathering information about an enemy [syn: intelligence activity, intelligence operation]

Source: WordNet 1.6, 1997 Princeton University

Artificial Inteligence and 'Knowledge representation and reasoning'

- AI is the science and engineering necessary to create artifacts that can
  - acquire knowledge, i.e., can learn; and
  - **reason with knowledge (leading to doing tasks such as planning, explaining, diagnosing, acting rationally, etc.),**
- Thus to build intelligent artifacts one of two important aspects that is needed is to be able to 'Reasoning with knowledge'
- To build an *artificial entity* (or artifact) that can 'reason with knowledge' one must 'represent the knowledge'.
- Central to Knowledge Representation and Reasoning is a knowledge representation framework consisting of
  - a knowledge representation language, **[KR]**
  - a language to reason (i.e., ask various questions), **[R]**
  - and connection between theories of the above two. **[KR & R]**

- An analogy: In summary, developing a **KR & R framework is as fundamental to build AI artifacts** as **Calculus is to Engineering.**

Knowledge Representation has come of age.

- Project Halo. (Digital Aristotle.)
- Semantic Web.
- Advance Question Answering
- Cellular event modeling.
- Knowledge based planning. (HTN, TLPLAN, TALPLAN)

What properties should a good KR & R language have?

1. Should be non-monotonic. (So that the system can revise its earlier conclusion in light of new information.)
2. Should have the ability to represent normative statements, exceptions, and default statements, and should be able to reason with them.
3. Should be expressive enough to express and answer problem solving queries such as planning queries, counterfactual queries, explanation queries and diagnostic queries.
4. Should have a simple and intuitive syntax so that domain experts (who may be non-computer scientists) can express knowledge using it.
5. Should have enough existing research (or building block results) about this language so that one does not have to start from scratch.
6. Should have interpreters or implementation of the language so that one can indeed represent and reason in this language. (I.e., it should not be just a theoretical language.)
7. Should have existing applications that have been built on this language so as to demonstrate the feasibility that applications can be indeed built using this language.

Knowledge Representation Languages: a short (biased) history

- First-order logic and propositional logics
  - both monotonic in nature.
- Description logic
  - a fragment of FOL;
  - is monotonic;
  - mostly focussed on representing and reasoning about ontologies or inheritance hierarchies
- Semantic Nets
  - basically a graphical sub-class of FOL
- Non-monotonic logics
  - Circumscription
  - Default logic

  - Non-monotonic modal logics (including auto-epistemic logic)

** complex syntax, no expressiveness advantages, limited building block results.

- AnsProlog* vs Pure Prolog
  - Differences:
    * Prolog is sensitive to ordering of rules and ordering of literals in the body of rules.
    * Inappropriate ordering leads to infinite loops. (Thus Prolog is not declarative; hence not a knowledge representation language)
    * Prolog stumbles with recursion through negation
    * No disjunction in the head (less expressive power in the propositional case)
  - Similarities: For certain subclasses of AnsProlog*, Prolog can be thought of as a top-down engine.

Why AnsProlog* ?

1. Is non-monotonic.
2. Has the ability to represent normative statements, exceptions, and default statements, and should be able to reason with them.
3. Is expressive enough to express and answer problem solving queries such as planning queries, counterfactual queries, explanation queries and diagnostic queries.
4. Has a simple and intuitive syntax so that domain experts (who may be non-computer scientists) can express knowledge using it.
5. Has enough existing research (or building block results) about this language so that one does not have to start from scratch.
6. Has interpreters or implementation of the language so that one can indeed represent and reason in this language. (I.e., it should not be just a theoretical language.)
7. Has existing applications that have been built on this language so as to demonstrate the feasibility that applications can be indeed built using this language.

SYNTAX AND SEMANTICS OF ANSPROLOG*

Terminologies – many brorrowed from classical logic

- variables: X, Y, Z, etc.
- object constants (or simply constants): a, b, c, etc.
- function symbols: f, g, h, etc.
- predicate symbols: p, q, etc.
- terms: variables, constants, and $f(t_1, \ldots, t_n)$ such that $t_i s$ are terms.
- atoms: $p(t_1, \ldots, t_n)$ such that $t_i s$ are terms.
- literals: atoms or an atom preceded by $\neg$.
- naf-literals: atoms or an atom preceded by **not**.
- gen-literals: literals or a literal preceded by **not**.
- ground terms (atoms, literals) : terms (atoms, literals resp.) without variables.

Herbrand Universe and Herbrand Base

- Some components of a language: variables, constants, functions, predicates
- $HU_{\mathcal{L}}$ – Herbrand Universe of a language $\mathcal{L}$: the set of all ground terms which can be formed with the functions and constants in $\mathcal{L}$.
- $HB_{\mathcal{L}}$ – Herbrand Base of a language $\mathcal{L}$: the set of all ground atoms which can be formed with the functions, constants and predicates in $\mathcal{L}$.
- Example:
  - Consider a language $\mathcal{L}_1$ with variables X, Y; constants a, b; function symbol f of arity 1; and predicate symbol p of arity 1.
  - $HU_{\mathcal{L}_1} = \{a, b, f(a), f(b), f(f(a)), f(f(b)), f(f(f(a))), f(f(f(b))), \ldots\}$.
  - $HB_{\mathcal{L}_1} = \{p(a), p(b), p(f(a)), p(f(b)), p(f(f(a))), p(f(f(b))),$
    $p(f(f(f(a)))), p(f(f(f(b)))), \ldots\}$.

AnsProlog* programs and their grounding

- A *rule* is of the form: L_0 *or* $\ldots$ *or* $L_k \leftarrow L_{k+1}, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n$.
- $ground(r, \mathcal{L})$: the set of all rules obtained from r by all possible substitution of elements of $HU_{\mathcal{L}}$ for the variables in r.
- Example:

  Consider the rule $p(f(X)) \leftarrow p(X)$. and the language $\mathcal{L}_1$. Then $ground(r, \mathcal{L}_1)$ will consist of the following rules:

  $p(f(a)) \leftarrow p(a).$
  $p(f(b)) \leftarrow p(b).$
  $p(f(f(a))) \leftarrow p(f(a)).$
  $p(f(f(b))) \leftarrow p(f(b)).$
  $\vdots$

- For an AnsProlog* program Π:
  - $ground(\Pi, \mathcal{L}) = \cup_{r \in \Pi} ground(r, \mathcal{L})$
  - $\mathcal{L}(\Pi)$: The langauge that is defined by the predicates, functions and constants occurring in Π.

- $ground(\Pi) = \cup_{r \in \Pi} ground(r, \mathcal{L}(\Pi))$.

- Example:
  - Π:

$$p(a).$$
$$p(b).$$
$$p(c).$$
$$p(f(X)) \leftarrow p(X).$$

  - $ground(\Pi)$:

$$p(a) \leftarrow.$$
$$p(b) \leftarrow .$$
$$p(c) \leftarrow .$$
$$p(f(a)) \leftarrow p(a).$$
$$p(f(b)) \leftarrow p(b).$$
$$p(f(c)) \leftarrow p(c).$$
$$p(f(f(a))) \leftarrow p(f(a)).$$
$$p(f(f(b))) \leftarrow p(f(b)).$$
$$p(f(f(c))) \leftarrow p(f(c)).$$
$$\vdots$$

AnsProlog: Syntax and semantics

- Syntax: A collection of rules of the form $L_0 \leftarrow L_1, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n.$ where L_i's are atoms.
- Semantics of a program Π: Given in terms of answer sets of the program $ground(\Pi)$.
  - Answer sets of AnsProlog$^{-\textbf{not}}$ programs (where $m = n$ for all rules):
    - The least Herbrand model of the program.
    - The minimal set of atoms (from Herbrand base of the program) that is closed under the program.
    - The least fixpoint of van Emden and Kowalski's iteration operator.
  - Gelfond-Lifschitz transformation: Given an AnsProlog program Π and a set S of atoms from HB_Π, the Gelfond-Lifschitz transformation Π^S is obtained by deleting
    (i) each rule that has a naf-literal **not** L in its body with $L \in S$, and
    (ii) naf-literals of the form **not** L in the bodies of the remaining rules.
  - Answer sets: S is an answer set of an AnsProlog program Π if S is the answer set of the AnsProlog$^{-\textbf{not}}$ program Π^S.

Examples

- Π_1:

  $p \leftarrow a.$
  $q \leftarrow b.$
  $a \leftarrow .$

- Π_2:

  $p \leftarrow p.$

- Π_3:

  $p \leftarrow p.$
  $q \leftarrow .$

- Π_4:

  $a \leftarrow \textbf{not}\ b.$

- Π_5:

  $a \leftarrow \textbf{not } b.$
  $b \leftarrow \textbf{not } a.$

- Π_6:

  $p \leftarrow a.$
  $a \leftarrow \textbf{not } b.$
  $b \leftarrow \textbf{not } a.$

- Π_7:

  $a \leftarrow \textbf{not } b.$
  $b \leftarrow \textbf{not } c.$
  $d \leftarrow .$

- Π_8:

  $p \leftarrow \textbf{not } p.$

- Π_9:

  $p \leftarrow \textbf{not } p, d.$
  $r \leftarrow.$
  $d \leftarrow .$

- Π_{10}:

  $p \leftarrow \textbf{not } p, d.$
  $r \leftarrow \textbf{not } d.$
  $d \leftarrow \textbf{not } r.$

- Π_{11}:

  $p \leftarrow \textbf{not } p.$
  $p \leftarrow \textbf{not } d.$
  $r \leftarrow \textbf{not } d.$
  $d \leftarrow \textbf{not } r.$

SOME BUILDING BLOCK RESULTS

Further intuitions behind the semantics

- A set of atoms S is **closed under** an AnsProlog program Π if for all rules of the form $L_0 \leftarrow L_1, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n$.
  in Π, $\{L_1, \ldots, L_m\} \subseteq S$ and $\{L_{m+1}, \ldots, L_n\} \cap S = \emptyset$ implies that $L_0 \in S$.
- A set of atoms S is said to be **supported by** Π if for all $p \in S$ there is a rule of the form $p \leftarrow L_1, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n$.
  in Π, such that $\{L_1, \ldots, L_m\} \subseteq S$ and $\{L_{m+1}, \ldots, L_n\} \cap S = \emptyset$.
- Clark defined a notion of completion of a program called Clark's completion. Models of the Clark's completion of an AnsProlog program Π are closed under and supported by Π and vice-versa.
- Answer sets of Π are models of Clark's Completion of Π. But not vice-versa. (example: $p \leftarrow p$.)

- **A set of atoms S is an answer set of an AnsProlog program Π iff (i) S is closed under Π and (ii) there exists a level mapping function λ (that maps atoms in S to a number) such that for each $p \in S$ there is a rule in Π of the form $p \leftarrow L_1, \ldots, L_m,$ not $L_{m+1}, \ldots,$ not L_n. such that $\{L_1, \ldots, L_m\} \subseteq S$, $\{L_{m+1}, \ldots, L_n\} \cap S = \emptyset$ and $\lambda(p) > \lambda(L_i)$, for $1 \leq i \leq m$.**
- Note that (ii) above implies that S is supported by Π.
- A logical characterization of (ii) above needs to be added to Clark's completion so as to have a 1-1 correspondence between models and answer sets.

Clark's Completion and some examples

- Given a propositional AnsProlog program Π consisting of rules of the form:

  $Q \leftarrow P_1, \ldots, P_n, \textbf{not } R_1, \ldots, \textbf{not } R_m.$

  its completion $Comp(\Pi)$ is obtained in two steps:

  - Step 1: Replace each rule of the above mentioned form with the formula:

    $Q \Leftarrow P_1 \wedge \ldots \wedge P_n \wedge \neg R_1 \wedge \ldots \neg R_m$
  - Step 2: For each symbol Q, let $Support(Q)$ denote the set of all clauses with Q in the head. Suppose $Support(Q)$ is the set:

    $Q \Leftarrow Body_1$

    $\vdots$

    $Q \Leftarrow Body_k$

    Replace this set with the single formula,

    $Q \Leftrightarrow Body_1 \vee \ldots \vee Body_k$.

    If $Support(Q) = \emptyset$ then replace it by $\neg Q$. $\square$

- Π_4:

  $a \leftarrow$ **not** b.

- Π_5:

  $a \leftarrow$ **not** b.
  $b \leftarrow$ **not** a.

- Π_2:

  $p \leftarrow p$.

- Π_8:

  $p \leftarrow$ **not** p.

- Initially: Characterizing **not** was thought to be the main problem, especially when dealing with non-stratified programs.
- New Insight: Clark's completion is a good and intuitive semantics for **not** ; but has problems with positive loops.

Analyzing AnsProlog programs using 'splitting'

- Purpose: Break down an AnsProlog program to smaller components in such a way that the analysis of the components can be carried over to the whole program.
- 'Splitting' is more general than the notions of 'stratification' and 'local stratification'. Those programs have a unique answer set.
- The program

  $p \leftarrow a.$
  $p \leftarrow b.$
  $a \leftarrow \textbf{not}\ b.$
  $b \leftarrow \textbf{not}\ a.$

  can be split to two layers.

- A *splitting set* for an AnsProlog program Π is any set U of atoms such that, for every rule $r \in \Pi$ of the form $L_0 \leftarrow L_1, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n.$ if $L_o \in U$ then $lit(r) = \{L_1, \ldots, L_n\} \subseteq U$. If U is a splitting set for Π, we also say that U splits Π. The set of rules $r \in \Pi$ such that $lit(r) \subseteq U$ is called the *bottom* of Π relative to the splitting set U and denoted by $bot_U(\Pi)$. The subprogram $\Pi \setminus bot_U(\Pi)$ is called *the top of* Π relative to U and denoted $top_U(\Pi)$.
- Consider the following program Π_1:

  $a \leftarrow b, \textbf{not } c.$
  $b \leftarrow c, \textbf{not } a.$
  $c \leftarrow.$

  The set $U = \{c\}$ splits Π_1 such that the last rule constitutes $bot_U(\Pi_1)$ and the first two rules form $top_U(\Pi_1)$.
- Once a program is split into top and bottom with respect to a splitting set, we can compute the answer sets of the bottom part and for each of these answer sets, we can further simplify the top part by partial evaluation before analyzing it further.

- Consider the following program:

  $p \leftarrow \textbf{not}\ q.$
  $p \leftarrow \textbf{not}\ p.$
  $q \leftarrow \textbf{not}\ r.$
  $r \leftarrow \textbf{not}\ q.$

  $\{q, r\}$ is a splitting set and the only answer set of the above program is $\{r, p\}$.

SOME KNOWLEDGE REPRESENTATION EXAMPLES

Normative statements and exceptions

- Normative statements are statements of the form "normally elements belonging to a class c have the property p."

  A good representation of normative statements should at least allow us to easily 'incorporate' information about exceptional elements of c with respect to the property c.

- "Normally birds Fly. Tweety and Sam are birds."

  $flies(X) \leftarrow bird(X), \textbf{not}\ ab(X).$
  $bird(tweety) \leftarrow.$
  $bird(sam) \leftarrow.$

- Adding "Sam is a penguin, and penguins are exceptional birds that do not fly," in an elaboration tolerant manner:

  $bird(X) \leftarrow penguin(X).$
  $ab(X) \leftarrow penguin(X).$
  $penguin(sam) \leftarrow.$

- Adding "Ostriches are exceptional birds that do not fly," in an elaboration tolerant manner:

  $bird(X) \leftarrow ostrich(X)$.
  $ab(X) \leftarrow ostrich(X)$.

- Weak defaults: For wounded birds we can not conclude whether they fly or not.
- AnsProlog is not quite appropriate for this. We need to be able to eliminate the embedded CWA and allow specification of CWA in a case by case basis.
- Expand AnsProlog to allow classical negation $\neg$.
- An example:

  $cross \leftarrow \mathbf{not}\ train$

  vs

  $cross \leftarrow \neg train$

AnsProlog$^\neg$ programs

- Syntax: A collection of rules of the form $L_0 \leftarrow L_1, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n.$ where L_i's are literals.
- Semantics of a program Π: Given in terms of answer sets of the program $ground(\Pi)$.
  - Answer sets of AnsProlog$^{\neg,-\textbf{not}}$ programs (where $m = n$ for all rules):
    - The minimal set of literals S (made up of atoms in the Herbrand base of the program) such that:
      - for any rule $L_0 \leftarrow L_1, \ldots, L_m.$ in the program , if $\{L_1, \ldots, L_m\} \subseteq S$ then $L_0 \in S$, and
      - if S contains a pair of complementary literals then S consists of all literals. (called Lit)
  - Gelfond-Lifschitz transformation: as before
  - Answer sets: S is an answer set of an AnsProlog program Π if S is the answer set of the AnsProlog$^{\neg,-\textbf{not}}$ program Π^S.

AnsProlog$^\neg$ examples

AnsProlog$^\neg$ programs	Their answer sets
$\{p \leftarrow q. \quad \neg p \leftarrow r. \quad q \leftarrow .\}$	$\{q, p\}$
$\{p \leftarrow q. \quad \neg p \leftarrow r. \quad r \leftarrow .\}$	$\{r, \neg p\}$
$\{p \leftarrow q. \quad \neg p \leftarrow r.\}$	$\{\}$
$\{p \leftarrow q. \quad \neg p \leftarrow r. \quad q \leftarrow . \quad r \leftarrow .\}$	Lit
$\{\neg p \leftarrow . \quad p \leftarrow \neg q. \}$	
$\{\neg p \leftarrow . \quad q \leftarrow \neg p. \}$	
$\{\neg q \leftarrow \textbf{not } p. \}$	
$\{a \leftarrow \textbf{not } b. \quad b \leftarrow \textbf{not } a. \quad q \leftarrow a. \quad \neg q \leftarrow a.\}$	

Explicit CWA and wounded birds

- Birds and penguins

  $flies(X) \leftarrow bird(X), \textbf{not } ab(X).$
  $bird(X) \leftarrow penguin(X).$
  $ab(X) \leftarrow penguin(X).$
  $bird(tweety) \leftarrow.$
  $penguin(sam) \leftarrow.$

  $\neg bird(X) \leftarrow \textbf{not } bird(X).$
  $\neg penguin(X) \leftarrow \textbf{not } penguin(X).$
  $\neg ab(X) \leftarrow \textbf{not } ab(X).$

  $\neg flies(X) \leftarrow penguin(X).$
  $\neg flies(X) \leftarrow \neg bird(X).$

- Advantage of the above representation: To remove CWA about a predicate, just remove the explicit CWA rule for that predicate.

- Incorporating the new knowledge: "John is an wounded bird and wounded birds are weak exceptions" by adding the following rules.

  $wounded\_bird(john) \leftarrow.$
  $\neg wounded\_bird(X) \leftarrow \textbf{not}\ wounded\_bird(X).$
  $bird(X) \leftarrow wounded\_bird(X).$
  $ab(X) \leftarrow wounded\_bird(X).$

OWA: i.e., no CWA

- Earlier: CWA about birds, penguins, ab and wounded birds, OWA about flies.
- Now: Assume that our information about birds, penguins, ab and wounded birds is incomplete.
- After removing explicit CWA about birds, penguins, ab and wounded birds.

$flies(X) \leftarrow bird(X), \textbf{not}\ ab(X).$
$bird(X) \leftarrow penguin(X).$
$ab(X) \leftarrow penguin(X).$
$bird(tweety) \leftarrow.$
$penguin(sam) \leftarrow.$
$\neg flies(X) \leftarrow penguin(X).$
$\neg flies(X) \leftarrow \neg bird(X).$
$wounded\_bird(john) \leftarrow.$
$bird(X) \leftarrow wounded\_bird(X).$
$ab(X) \leftarrow wounded\_bird(X).$

Add bird(et);

flies(et) will succeed

- *et* is a bird. Does it fly?
  - Earlier representation will answer yes!
  - But, now that we no longer have CWA about being a penguin, its possible that et might be a penguin. We now need to be more conservative in our reasoning.

$flies(X) \leftarrow bird(X), \textbf{not}\ ab(X).$
$bird(X) \leftarrow penguin(X).$
$ab(X) \leftarrow \textbf{not}\ \neg penguin(X).$

Notice how the ab(X) definition is different.

$bird(tweety) \leftarrow.$
$penguin(sam) \leftarrow.$

flies(et) will now fail, as I fail to establish that et is not a penguin

$\neg penguin(X) \leftarrow \neg bird(X).$
$\neg flies(X) \leftarrow penguin(X).$
$\neg flies(X) \leftarrow \neg bird(X).$
$wounded\_bird(john) \leftarrow.$
$\neg wounded\_bird(X) \leftarrow \neg bird(X).$
$bird(X) \leftarrow wounded\_bird(X).$
$ab(X) \leftarrow \textbf{not}\ \neg wounded\_bird(X).$
$bird(et).$

Allowing for explicit negative information

- Suppose we have explicit information about the non-flying ability of certain birds.
- Adding such information such as $\{\neg fly(tweety), \neg penguin(tweety), \neg wounded\_bird(tweety)\}$ should not lead to inconsistency. But it does!
- Such a breakdown can be avoided by replacing the rule

  $flies(X) \leftarrow bird(X), \textbf{not } ab(X).$

  by the rule:

  $flies(X) \leftarrow bird(X), \textbf{not } ab(X), \textbf{not } \neg flies(X).$

  In addition, for *exceptions* we only need to have the rule:

  $\neg flies(X) \leftarrow exceptional\_bird(X).$

  and no longer need a rule of the form:

  $ab(X) \leftarrow \textbf{not } \neg exceptional\_bird(X).$

  For *weak exceptions* we will need the rule:

  $ab(X) \leftarrow \textbf{not } \neg weakly\_exceptional\_bird(X).$

Systematic removal of CWA : an example

- Transitive Closure:

  $anc(X, Y) \leftarrow parent(X, Y)$.
  $anc(X, Y) \leftarrow parent(X, Z), anc(Z, Y)$.

  It assumes complete information about parent and gives a complete definition of ancestor.

- Consider the case that where the objects are fossils of dinosaurs dug at an archaeological site, and for pairs of objects (a, b) we can sometimes determine $par(a, b)$ through tests, sometimes determine $\neg par(a, b)$, and sometimes neither.

  How do we define when anc is true and when it is false.

- To define when anc is true we keep the old rules.

  $anc(X, Y) \leftarrow parent(X, Y).$
  $anc(X, Y) \leftarrow parent(X, Z), anc(Z, Y).$

  Next we define a predicate $m\_par(X, Y)$ which encodes when X may be a parent of Y.

  $m\_par(X, Y) \leftarrow \textbf{not } \neg par(X, Y).$

  Using m_par we now define $m\_anc(X, Y)$ which encodes when X may be an ancestor of Y.

  $m\_anc(X, Y) \leftarrow m\_par(X, Y).$
  $m\_anc(X, Y) \leftarrow m\_par(X, Z), m\_anc(Z, Y).$

  Now we use m_anc to define when $\neg anc(X, Y)$ is true.

  $\neg anc(X, Y) \leftarrow \textbf{not } m\_anc(X, Y).$

Reasoning about what is known and what is not

(1) $eligible(X) \leftarrow highGPA(X)$.
(2) $eligible(X) \leftarrow special(X), fairGPA(X)$.
(3) $\neg eligible(X) \leftarrow \neg special(X), \neg highGPA(X)$.
(4) $interview(X) \leftarrow \textbf{not}\ eligible(X), \textbf{not}\ \neg eligible(X)$.
(5) $fairGPA(john) \leftarrow$.
(6) $\neg highGPA(john) \leftarrow$.

BASIC DECLARATIVE PROBLEM SOLVING MODULES

Constraints

- Π:

  $a \leftarrow$ **not** b.
  $b \leftarrow$ **not** a.

- Π has two answer sets $\{a\}$ and $\{b\}$.
- Now suppose we would like to incorporate the constraint r :

  $\leftarrow a$.

  to prune any answer set where a is true.

- To achieve this we can add the following rule.

  $p \leftarrow$ **not** p, a.

  to Π.

- In general an integrity constraint r written as

  $\leftarrow l_1, \ldots, l_m, \textbf{not } l_{m+1}, \ldots, \textbf{not } l_n,$

  where l_is are literals, *forbids* answer sets which contain the literals $l_1, \ldots, l_m$ and do not contain the literals $l_{m+1}, \ldots, l_n$.

- r can be simulated by adding the following rule to the program.

  $p \leftarrow \textbf{not } p, l_1, \ldots, l_m, \textbf{not } l_{m+1}, \ldots, \textbf{not } l_n.$

  where p does not appear in the original program.

- r can also be simulated by adding the following rules to the program.

  $p \leftarrow l_1, \ldots, l_m, \textbf{not } l_{m+1}, \ldots, \textbf{not } l_n.$

  $q \leftarrow \textbf{not } p.$

  $q \leftarrow \textbf{not } q.$

  where p and q do not appear in the original program.

Enumeration

- Finite enumeration:

  $p_1 \leftarrow \textbf{not}\ n\_p_1.$
  $n\_p_1 \leftarrow \textbf{not}\ p_1.$
  $\vdots$
  $p_n \leftarrow \textbf{not}\ n\_p_n.$
  $n\_p_n \leftarrow \textbf{not}\ p_n.$

- General enumeration but at least one:

  $chosen(X) \leftarrow possible(X), \textbf{not}\ not\_chosen(X).$
  $not\_chosen(X) \leftarrow possible(X), \textbf{not}\ chosen(X).$
  $some \leftarrow chosen(X).$
  $\leftarrow \textbf{not}\ some.$

- Consider $\{possible(a), possible(b), possible(c)\}$.

- Choice: General enumeration with exactly one:

  Our goal is to have a program which has answer sets with the following as subsets.

  $S_1 = \{chosen(a), \neg chosen(b), \neg chosen(c)\}$

  $S_2 = \{\neg chosen(a), chosen(b), \neg chosen(c)\}$

  $S_3 = \{\neg chosen(a), \neg chosen(b), chosen(c)\}$

  $\neg chosen(X) \leftarrow chosen(Y), X \neq Y.$

  $chosen(X) \leftarrow possible(X), \mathbf{not}\ \neg chosen(X).$

- Choice using AnsProlog:

  $diff\_chosen\_than(X) \leftarrow chosen(Y), X \neq Y.$

  $chosen(X) \leftarrow possible(X), \mathbf{not}\ diff\_chosen\_than(X).$

- Choice in Smodels syntax:

  $1\{chosen(X) : possible(X)\}1.$

Propositional Satisfiability: an illustration

- $S = \{p_1 \vee p_2 \vee p_3, p_1 \vee \neg p_3, \neg p_2 \vee \neg p_4\}$.
- The AnsProlog program $\Pi(S)$ consists of the following:

$p_1 \leftarrow \textbf{not } n\_p_1.$ $\quad n\_p_1 \leftarrow \textbf{not } p_1.$
$p_2 \leftarrow \textbf{not } n\_p_2.$ $\quad n\_p_2 \leftarrow \textbf{not } p_2.$
$p_3 \leftarrow \textbf{not } n\_p_3.$ $\quad n\_p_3 \leftarrow \textbf{not } p_3.$
$p_4 \leftarrow \textbf{not } n\_p_4.$ $\quad n\_p_4 \leftarrow \textbf{not } p_4.$

$c_1 \leftarrow p_1.$ $\quad c_1 \leftarrow p_2.$ $\quad c_1 \leftarrow p_3.$ $\quad \leftarrow \textbf{not } c_1.$

$c_2 \leftarrow p_1.$ $\quad c_2 \leftarrow n\_p_3.$ $\quad \leftarrow \textbf{not } c_2.$

$c_3 \leftarrow n\_p_2.$ $\quad c_3 \leftarrow n\_p_4.$ $\quad \leftarrow \textbf{not } c_3.$

- S is satisfiable iff $\Pi(S)$ has an answer set.
- Basic idea: enumerate possibilities; add constraints.

N-queens: naming queens and placing them one by one

1. Declarations:

   $queen(1) \leftarrow .$ $\quad \ldots \quad$ $queen(n) \leftarrow .$
   $row(1) \leftarrow .$ $\quad \ldots \quad$ $row(n) \leftarrow .$
   $col(1) \leftarrow .$ $\quad \ldots \quad$ $col(n) \leftarrow .$

2. Enumeration:

   2.1. For each locations (X, Y) and each queen I, either I is in location (X, Y) or not.
   $at(I, X, Y) \leftarrow queen(I), row(X), col(Y), \textbf{not}\ not\_at(I, X, Y).$
   $not\_at(I, X, Y) \leftarrow queen(I), row(X), col(Y), \textbf{not}\ at(I, X, Y).$

   2.2. For each queen I it is placed in at most one location.
   $\leftarrow queen(I), row(X), col(Y), row(U), col(Z), at(I, X, Y), at(I, U, Z), Y \neq Z.$
   $\leftarrow queen(I), row(X), col(Y), row(Z), col(V), at(I, X, Y), at(I, Z, V), X \neq Z.$

   2.3. For each queen I it is placed in at least one location.
   $placed(I) \leftarrow queen(I), row(X), col(Y), at(I, X, Y).$
   $\leftarrow queen(I), \textbf{not}\ placed(I).$

2.4. No two queens are placed in the same location.

$\leftarrow queen(I), row(X), col(Y), queen(J), at(I,X,Y), at(J,X,Y), I \neq J.$

3. Elimination:

3.1. No two distinct queens in the same row.

$\leftarrow queen(I), row(X), col(Y), col(V), queen(J), at(I,X,Y), at(J,X,V), I \neq J.$

3.2. No two distinct queens in the same column.

$\leftarrow queen(I), row(X), col(Y), row(U), queen(J), at(I,X,Y), at(J,U,Y), I \neq J.$

3.3. No two distinct queens attack each other diagonally.

$\leftarrow row(X), col(Y), row(U), col(V), queen(I), queen(J), at(I,X,Y),$
$at(J,U,V), I \neq J, abs(X-U) = abs(Y-V).$

N-queens: Similar to last one but now using Choice rules

1. Declarations: As in the previous formulation.
2. Enumeration:
   2.1. Choice using $other\_at(I, X, Y)$ which intuitively means that the queen I has been placed in a location other than (X, Y).
   $$other\_at(I, X, Y) \leftarrow queen(I), row(X), col(Y), row(U), col(Z), at(I, U, Z), Y \neq Z.$$
   $$other\_at(I, X, Y) \leftarrow queen(I), row(X), col(Y), row(Z), col(V), at(I, Z, V), X \neq Z.$$
   $$at(I, X, Y) \leftarrow queen(I), row(X), col(Y), \textbf{not } other\_at(I, X, Y).$$
   2.2. No two queens are placed in the same location. (Same as in 2.4 in the last slide.)
   $$\leftarrow queen(I), row(X), col(Y), queen(J), at(I, X, Y), at(J, X, Y), I \neq J.$$
3. Elimination: As in the previous formulation.

N-queens: a solution without naming queens

1. Declarations: The simpler domain specification is as follows:

   $row(1) \leftarrow .$ $\quad \ldots \quad$ $row(n) \leftarrow.$
   $col(1) \leftarrow .$ $\quad \ldots \quad$ $col(n) \leftarrow.$

2. Enumeration:

   2.1. A queen should not be placed in (X, Y) if there is a queen placed in the same row.
   $not\_in(X, Y) \leftarrow row(X), col(Y), col(YY), Y \neq YY, in(X, YY)$

   2.2. A queen should not be placed in (X, Y) if there is a queen placed in the same column.
   $not\_in(X, Y) \leftarrow row(X), col(Y), row(XX), X \neq XX, in(XX, Y)$

   2.3. A queen must be placed in (X, Y) if it is not otherwise prevented.
   $in(X, Y) \leftarrow row(X), col(Y), \mathbf{not}\ not\_in(X, Y)$

3. Elimination: No two queens attack each other diagonally.

   $\leftarrow row(X), col(Y), row(XX), col(YY), X \neq XX, Y \neq YY,$
   $in(X, Y), in(XX, YY), abs(X - XX) = abs(Y - YY).$

AnsProlog^{or} : Syntax and Semantics

- Syntax: A collection of rules of the form
  L_0 *or* $\ldots$ *or* $L_k \leftarrow L_{k+1}, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n$.
  where L_i's are atoms.
- Semantics of a program Π: Given in terms of answer sets of the program $ground(\Pi)$.
  - Answer sets of AnsProlog$^{-\textbf{not}, or}$ programs (where $m = n$ for all rules):
    - The minimal models of the program.
  - Gelfond-Lifschitz transformation: as before
  - Answer sets: S is an answer set of an AnsProlog^{or} program Π if S is **an** answer set of the AnsProlog$^{-\textbf{not}, or}$ program Π^S.
- Translating AnsProlog^{or} to AnsProlog:

– For head-cycle free programs:

Replace $L_0 \; or \; \ldots \; or \; L_k \leftarrow Body.$ by

$L_0 \leftarrow Body, \mathbf{not}\ L_1, \ldots, \mathbf{not}\ L_k.$

$\vdots$

$L_k \leftarrow Body, \mathbf{not}\ L_1, \ldots, \mathbf{not}\ L_{k-1}.$

– Consider

$a \; or \; b$

$a \leftarrow b$

$b \leftarrow a$

Only answer set is $\{a, b\}$.

– The AnsProlog translation

$a \leftarrow \mathbf{not}\ b$

$b \leftarrow \mathbf{not}\ a$

$a \leftarrow b$

$b \leftarrow a$

has no answer sets.

Satisfiability of Universal-existential QBFs using AnsProlog^{or}

- The QBF
  $\forall p_1, \ldots p_k.\ \exists q_1, \ldots, q_l.\ (\phi_1(p_1, \ldots, p_k, q_1, \ldots, q_l) \wedge \ldots \wedge \phi_n(p_1, \ldots, p_k, q_1, \ldots, q_l))$,
  where ϕ_i's are disjunction of propositional literals, is satisfiable iff $not\_satisfied$ is false in all answer sets of the following program:

  $p_1\ or\ p_1' \leftarrow.$ $\quad \ldots \quad$ $p_k\ or\ p_k' \leftarrow.$

  $q_1\ or\ q_1' \leftarrow.$ $\quad \ldots \quad$ $q_l\ or\ q_l' \leftarrow.$

  $not\_satisfied \leftarrow \hat{\phi}_1.$ $\quad \ldots \quad$ $not\_satisfied \leftarrow \hat{\phi}_n.$

  $q_1 \leftarrow not\_satisfied.$

  $q_1' \leftarrow not\_satisfied.$

  $\ldots$

  $q_l \leftarrow not\_satisfied.$

  $q_l' \leftarrow not\_satisfied.$

- Intuition: For a given interpretation I_p of the ps if for some interpretation I_q of the qs the formula $F = \phi_1 \wedge \ldots \wedge \phi_n$ is satisfied then it forces out any answer set due to I_p and $I_{q'}$ which does not satisfy F as the later will be forced to have all combination of q_is making it non-minimal.
- Thus AnsProlog^{or} can express $\Pi_2\mathbf{P}$ problems while AnsProlog can only express $\Pi_1\mathbf{P}$ problems.

Sorting using AnsProlog

- Smallest largest and next:

  $not\_smallest(X) \leftarrow p(X), p(Y), less\_than(Y, X).$
  $smallest(X) \leftarrow p(X), \textbf{not}\ not\_smallest(X).$

  $not\_largest(X) \leftarrow p(X), p(Y), less\_than(X, Y).$
  $largest(X) \leftarrow p(X), \textbf{not}\ not\_largest(X).$

  $not\_next(X, Y) \leftarrow X = Y.$
  $not\_next(X, Y) \leftarrow less\_than(Y, X).$
  $not\_next(X, Y) \leftarrow p(X), p(Y), p(Z), less\_than(X, Z), less\_than(Z, Y).$
  $next(X, Y) \leftarrow p(X), p(Y), \textbf{not}\ not\_next(X, Y).$

- $q(1, X) \leftarrow smallest(X).$
  $q(i + 1, X) \leftarrow q(i, Y), next(Y, X).$

- Grounding of the above leads to $O(n^3)$ ground rules. Can we do better?

Aggregation within AnsProlog$^\neg$

- Consider the following database, where $sold(a, 10, Jan1)$ means that 10 units of item a was sold on Jan 1.

  $sold(a, 10, Jan1) \leftarrow$.
  $sold(a, 21, Jan5) \leftarrow$.
  $sold(a, 15, Jan16) \leftarrow$.
  $sold(b, 16, Jan4) \leftarrow$.
  $sold(b, 31, Jan21) \leftarrow$.
  $sold(b, 15, Jan26) \leftarrow$.
  $sold(c, 24, Jan8) \leftarrow$.

  We would like to answer queries such as: "List all items of which more than 50 units (total) were sold, and the total quantity sold for each."

- The encoding in AnsProlog$^\neg$.

1. Assigning numbers to each tuple of *sold* while grouping them based on their item. I.e., we would like the answer sets to contain the following facts (or similar ones with a different numbering).

   $assigned(a, 10, 1) \leftarrow$.
   $assigned(a, 21, 2) \leftarrow$.
   $assigned(a, 15, 3) \leftarrow$.
   $assigned(b, 16, 1) \leftarrow$.
   $assigned(b, 31, 2) \leftarrow$.
   $assigned(b, 15, 3) \leftarrow$.
   $assigned(c, 24, 1) \leftarrow$.

   The AnsProlog program with such answer sets has the following three groups of rules:

   1.1. Unique assignment of numbers to each pair of item, and units.
   $assigned(X, Y, J) \leftarrow sold(X, Y, D), \mathbf{not}\ \neg assigned(X, Y, J)$.
   $\neg assigned(X, Y, J) \leftarrow assigned(X, Y', J), Y \neq Y'$.
   $\neg assigned(X, Y, J) \leftarrow assigned(X, Y, J'), J \neq J'$

   1.2. Among the tuples corresponding to each item there is no gap in the number assignment.
   $numbered(X, J) \leftarrow assigned(X, Y, J)$.

$\leftarrow numbered(X, J+1), \textbf{not}\ numbered(X, J), J \geq 1.$

1.3. The following rules ensure that for each item, there is a tuple that is assigned the number 1.
$one\_is\_assigned(X) \leftarrow assigned(X, Y, 1).$
$\leftarrow sold(X, Y, D), \textbf{not}\ one\_is\_assigned(X).$

2. Initializing and updating the aggregate operations. Depending on the aggregate operators the *initialize* and *update* facts describe how to start the aggregation process, when the first tuple in each grouping is encountered, and how the aggregate value is updated when additional tuples are encountered.

2.1. Sum:
$initialize(sum, Y, Y) \leftarrow.$
$update(sum, W, Y, W+Y) \leftarrow.$

2.2. $initialize(sum, Y, Y)$ means that for the aggregate sum, during the aggregation process when the tuple $(\_, Y)$ that is assigned the initial number (which is 1) is considered, Y is the value from which the aggregation starts.

2.3. The aggregation starts from the tuple assigned 1, and runs though the other tuples in the linear order of their assignment.

2.4. The fact, $update(sum, W, Y, W+Y)$ is used in that process and intuitively means that while doing the aggregation sum, if the next tuple (based on the

ordering of its assignment) is $(\_, Y)$, and the current accumulated value is W, then after considering this tuple the accumulated value is to be updated to $W + Y$.

2.5. Count:
$initialize(count, Y, 1)$.
$update(count, W, Y, W + 1) \leftarrow$.

2.6. Min:
$initialize(min, Y, Y) \leftarrow$.
$update(min, W, Y, W) \leftarrow W \leq Y$.
$update(min, W, Y, Y) \leftarrow Y \leq W$.

3. Using the predicates *initialize* and *update* we can define other aggregate operators of our choice. Thus, AnsProlog allows us to express user-defined aggregates.
4. The following three rules describe how the *initialize* and *update* predicates are used in computing the aggregation. The first rule uses *initialize* to account for the tuple that is assigned the number 1, and the second rule encodes the aggregate computation when we already have computed the aggregate up to the Jth tuple, and we encounter the J+1th tuple.

$aggr(Aggr\_name, 1, X, Z) \leftarrow assigned(X, Y, 1), initialize(Aggr\_name, Y, Z).$
$aggr(Aggr\_name, J+1, X, Z) \leftarrow J > 0, aggr(Aggr\_name, J, X, W),$
$assigned(X, Y, J+1),$
$update(Aggr\_name, W, Y, Z).$

5. Computing new aggregate predicates: Once the aggregation is done, we can define new predicates for the particular aggregation that we need. Following are some examples of the encoding of such predicates.

   5.1. Total sold per item:
   $total\_sold\_per\_item(X, Q) \leftarrow aggr(sum, J, X, Q),$
   not $aggr(sum, J+1, X, Y).$

   5.2. Number of transactions per item:
   $number\_of\_transactions\_per\_item(X, Q) \leftarrow aggr(count, J, X, Q),$
   not $aggr(count, J+1, X, Y).$

   5.3. Minimum amount (other than zero) sold per item:
   $min\_sold\_per\_item(X, Q) \leftarrow aggr(min, J, X, Q),$
   not $aggr(min, J+1, X, Y).$

- Using the above program the answer sets we will obtain will contain the following:
  $total\_sold\_per\_item(a, 46).$ $total\_sold\_per\_item(b, 62).$ $total\_sold\_per\_item(c, 24).$

COMING TOGETHER OF KR, R AND DPS: REASONING ABOUT ACTIONS AND PLANNING

$\mathcal{A}$: A simple action language

- Syntax:
  - Domain description (D): A collection of effect propositions of the form:
    a **causes** f **if** $p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$
    where a is an action, f is a fluent literal, and $p_1, \ldots, p_n, q_1, \ldots, q_r$ are fluents.
  - Observations(O):
    - General observations (Oracles): f **after** $a_1, \ldots, a_m$
    - Initial observations: **initially** f
  - Queries(Q): f **after** $a_1, \ldots, a_m$
- Semantics: Defining $D \models_O Q$

Examples of various kinds of reasoning in $\mathcal{A}$

- Temporal projection:
  - $D = \{load$ **causes** $loaded;$ $\quad shoot$ **causes** $\neg alive$ **if** $loaded\}$
  - $O = \{$ **initially** $alive\}.$
  - $D \not\models_O \neg alive$ **after** $shoot$
  - $D \models_O \neg alive$ **after** $load, shoot$
  - $O' = \{$ **initially** $alive;$ $\quad$ **initially** $loaded\}$
  - $D \models_{O'} \neg alive$ **after** $shoot$
- Reasoning about the initial situation
  - $O_1 = \{$ **initially** $alive;$ $\quad \neg alive$ **after** $shoot\}$
  - $D \models_{O_1}$ **initially** $loaded$

- Observation assimilation:
  - generalizes temporal projection and reasoning about the initial situation
  - $D = \{load$ **causes** $loaded;$ $\quad shoot$ **causes** $\neg alive$ **if** $loaded\}$
  - $O_2 = \{$ **initially** $alive;$ $\quad loaded$ **after** $shoot\}$
  - $D \models_{O_2} \neg alive$ **after** $shoot.$
- Planning
  - $O = \{$ **initially** $alive\}$ and $O' = \{$ **initially** $alive;$ $\quad$ **initially** $loaded\}$
  - The goal: $G = \{\neg alive\}$
  - $shoot$ is a plan for G with respect to (D, O')
  - $shoot$ is not a plan for G with respect to (D, O)
  - $load; shoot$ is a plan for G with respect to (D, O).

Implementing temporal projection using AnsProlog: π_1

- Assumption: Observations are only about the initial state and are complete.
- Translating effect propositions: For every effect proposition of the form a **causes** f **if** $p_1, \ldots, p_n, \neg q_1, \ldots \neg q_r$, if f is a fluent then have the following rule

  $holds(f, res(a, S)) \leftarrow$
  $holds(p_1, S), \ldots, holds(p_n, S), \textbf{not } holds(q_1, S), \ldots, \textbf{not } holds(q_r, S).$

  else, if f is the negative fluent literal $\neg g$ then have the following rule:

  $ab(g, a, S) \leftarrow holds(p_1, S), \ldots, holds(p_n, S), \textbf{not } holds(q_1, S), \ldots, \textbf{not } holds(q_r, S).$
- Translating observations: For every value proposition of the form **initially** f, if f is a fluent then have the following:

  $holds(f, s_0) \leftarrow.$
- Inertia rule:

  $holds(F, res(A, S)) \leftarrow holds(F, S), \textbf{not } ab(F, A, S).$

- An Example:
  - $D = \{load$ **causes** $loaded;$ $\quad shoot$ **causes** $\neg alive$ **if** $loaded\}$
  - $O_3 = \{$ **initially** $alive;$ $\quad$ **initially** $\neg loaded\}$
  - The AnsProlog rules will be:
    $holds(loaded, res(load, S)) \leftarrow.$
    $ab(alive, shoot, S) \leftarrow holds(loaded, S).$
    $holds(alive, s_0) \leftarrow .$
    $holds(F, res(A, S)) \leftarrow holds(F, S), \textbf{not}\ ab(F, A, S).$
  - Notation: $[a_n, \ldots, a_1]$ denotes the situation $res(a_n \ldots res(a_1, s_0) \ldots).$
  - From the program we can entail $holds(alive, [load])$, and $holds(loaded, [load])$, and $\neg holds(alive, [shoot, load])$.
- Proposition: Let D be a consistent domain description and O be an initial state complete set of observations such that (D, O) is consistent. Let f be a fluent.
  (i) $D \models_O f$ **after** $a_1, \ldots, a_n$ iff $\pi_1(D, O) \models holds(f, [a_n, \ldots, a_1])$.
  (ii) $D \models_O \neg f$ **after** $a_1, \ldots, a_n$ iff $\pi_1(D, O) \not\models holds(f, [a_n, \ldots, a_1])$

Implementing temporal projection using AnsProlog$^\neg$: π_2

- Assumption: Observations are only about the initial state and are complete.
- Translating effect propositions: For every effect proposition of the form a **causes** f **if** $p_1, \ldots, p_n, \neg q_1, \ldots \neg q_r$, if f is a fluent then we have:

  $holds(f, res(a, S)) \leftarrow holds(p_1, S), \ldots, holds(p_n, S),$
  $\neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

  $ab(f, a, S) \leftarrow holds(p_1, S), \ldots, holds(p_n, S), \neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

  else, if f is the negative fluent literal $\neg g$ then we have:

  $\neg holds(g, res(a, S)) \leftarrow holds(p_1, S), \ldots, holds(p_n, S),$
  $\neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

  $ab(g, a, S) \leftarrow holds(p_1, S), \ldots, holds(p_n, S), \neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

- Translating observations: For every value proposition of the form **initially** f, if f is a fluent then we have:

  $holds(f, s_0) \leftarrow$.

  else, if f is the negative fluent literal $\neg g$ then we have:

  $\neg holds(g, s_0) \leftarrow$.

- Inertia rules:

  $holds(F, res(A, S)) \leftarrow holds(F, S), \textbf{not } ab(F, A, S)$.

  $\neg holds(F, res(A, S)) \leftarrow \neg holds(F, S), \textbf{not } ab(F, A, S)$.

- Proposition: Let D be a consistent domain description and O be an initial state complete set of observations such that (D, O) is consistent. Let f be a fluent.

  (i) $D \models_O f$ **after** $a_1, \ldots, a_n$ iff $\pi_2(D, O) \models holds(f, [a_n, \ldots, a_1])$.

  (ii) $D \models_O \neg f$ **after** $a_1, \ldots, a_n$ iff $\pi_2(D, O) \models \neg holds(f, [a_n, \ldots, a_1])$.

Temporal projection in AnsProlog$^\neg$ in presence of incompleteness: π_3

- An Example where the previous encoding falters:
  - $D = \{load$ **causes** $loaded$; $\quad shoot$ **causes** $\neg alive$ **if** $loaded\}$
  - $O_4 = \{$ **initially** $alive\}$.
  - The earlier AnsProlog$^\neg$ implementation will consist of:
    $holds(loaded, res(load, S)) \leftarrow$.
    $ab(loaded, load, S) \leftarrow$.
    $\neg holds(alive, res(shoot, S)) \leftarrow holds(loaded, S)$.
    $ab(alive, shoot, S) \leftarrow holds(loaded, S)$.
    $holds(alive, s_0) \leftarrow$.
    $holds(F, res(A, S)) \leftarrow holds(F, S), \textbf{not}\ ab(F, A, S)$.
    $\neg holds(F, res(A, S)) \leftarrow \neg holds(F, S), \textbf{not}\ ab(F, A, S)$.
  - The program will incorrectly entail $holds(alive, [shoot])$, while $D \not\models_{O_4} alive$ **after** $shoot$.

- A conservative encoding:
  - Translating effect propositions: The effect propositions in D are translated as follows and are collectively referred to as π_3^{ef}.

    For every effect proposition of the form a **causes** f **if** $p_1, \ldots, p_n, \neg q_1, \ldots, q_r$, if f is a fluent then we have:

    $holds(f, res(a, S)) \leftarrow holds(p_1, S), \ldots, holds(p_n, S),$
    $\neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

    $ab(f, a, S) \leftarrow$ **not** $\neg holds(p_1, S), \ldots,$ **not** $\neg holds(p_n, S),$
    not $holds(q_1, S), \ldots,$ **not** $holds(q_r, S).$

    else, if f is the negative fluent literal $\neg g$ then we have:

    $\neg holds(g, res(a, S)) \leftarrow holds(p_1, S), \ldots, holds(p_n, S),$
    $\neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

    $ab(g, a, S) \leftarrow$ **not** $\neg holds(p_1, S), \ldots,$ **not** $\neg holds(p_n, S),$
    not $holds(q_1, S), \ldots,$ **not** $holds(q_r, S).$
  - Observations and inertia rules are as before.

- Proposition: Let D be a consistent domain description and O be an initial state complete set of observations such that (D, O) is consistent. Let f be a fluent.

  (i) $D \models_O f$ **after** $a_1, \ldots, a_n$ iff $\pi_3(D, O) \models holds(f, [a_n, \ldots, a_1])$.

  (ii) $D \models_O \neg f$ **after** $a_1, \ldots, a_n$ iff $\pi_3(D, O) \models \neg holds(f, [a_n, \ldots, a_1])$.

- Proposition: Let D be a consistent domain description, and O be a (possibly incomplete) set of observations about the initial state such that (D, O) is consistent. Let f be a fluent.

  (i) If $\pi_3(D, O) \models holds(f, [a_n, \ldots, a_1])$ then $D \models_O f$ **after** $a_1, \ldots, a_n$.

  (ii) If $\pi_3(D, O) \models \neg holds(f, [a_n, \ldots, a_1])$ then $D \models_O \neg f$ **after** $a_1, \ldots, a_n$.

Assimilating observations using enumeration and constraints: π_4

- Assumption: Observations are not necessarily about the initial state and do not have to be complete.
- Translating effect propositions: For every effect proposition of the form a **causes** f **if** $p_1, \ldots, p_n, \neg q_1, \ldots \neg q_r$, if f is a fluent then we have:

  $holds(f, res(a, S)) \leftarrow holds(p_1, S), \ldots, holds(p_n, S),$
  $\neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

  $ab(f, a, S) \leftarrow holds(p_1, S), \ldots, holds(p_n, S), \neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

  else, if f is the negative fluent literal $\neg g$ then we have:

  $\neg holds(g, res(a, S)) \leftarrow holds(p_1, S), \ldots, holds(p_n, S),$
  $\neg holds(q_1, S), \ldots, \neg holds(q_r, S).$

  $ab(g, a, S) \leftarrow holds(p_1, S), \ldots, holds(p_n, S), \neg holds(q_1, S), \ldots, \neg holds(q_r, S).$
- Inertia rules:

  $holds(F, res(A, S)) \leftarrow holds(F, S), \textbf{not } ab(F, A, S).$

  $\neg holds(F, res(A, S)) \leftarrow \neg holds(F, S), \textbf{not } ab(F, A, S).$

- Enumeration:

  $holds(F, s_0) \leftarrow \textbf{not } \neg holds(F, s_0).$

  $\neg holds(F, s_0) \leftarrow \textbf{not } holds(F, s_0).$

- Translating observations: For an observation of the form f **after** $a_1, \ldots, a_m$, if f is a fluent then we have:

  $\leftarrow \textbf{not } holds(f, [a_m, \ldots, a_1]).$

  else if f is the fluent literal $\neg g$, then we have:

  $\leftarrow \textbf{not } \neg holds(g, [a_m, \ldots, a_1]).$

- Proposition: Let D be a consistent domain description, and O be a set of observations such that (D, O) is consistent. Let f be a fluent.

  (i) $\pi_4(D, O) \models holds(f, [a_n, \ldots, a_1])$ iff $D \models_O f$ **after** $a_1, \ldots, a_n$.

  (ii) $\pi_4(D, O) \models \neg holds(f, [a_n, \ldots, a_1])$ iff $D \models_O \neg f$ **after** $a_1, \ldots, a_n$.

A narrative-based formulation of temporal projection: $\pi_{2.nar}$

- Translating effect propositions: For every effect proposition of the form a **causes** f **if** $p_1, \ldots, p_n, \neg q_1, \ldots \neg q_r$, if f is a fluent then we have:

  $holds(f, T+1) \leftarrow occurs(a, T), holds(p_1, T), \ldots, holds(p_n, T),$
  $not\_holds(q_1, T), \ldots, not\_holds(q_r, T).$

  $ab(f, a, T) \leftarrow occurs(a, T), holds(p_1, T), \ldots, holds(p_n, T),$
  $not\_holds(q_1, T), \ldots, not\_holds(q_r, T).$

  else, if f is the negative fluent literal $\neg g$ then we have:

  $not\_holds(g, T+1) \leftarrow occurs(a, T), holds(p_1, T), \ldots, holds(p_n, T),$
  $not\_holds(q_1, T), \ldots, not\_holds(q_r, T).$

  $ab(g, a, T) \leftarrow occurs(a, T), holds(p_1, T), \ldots, holds(p_n, T),$
  $not\_holds(q_1, T), \ldots, not\_holds(q_r, T).$

- Translating observations: For every value proposition of the form **initially** f, if f is a fluent then we have:

  $holds(f, 1) \leftarrow$.

  else, if f is the negative fluent literal $\neg g$ then we have:

  $not\_holds(g, 1) \leftarrow$.

- Inertia rules: Besides the above we have the following inertia rules:

  $holds(F, T+1) \leftarrow occurs(A, T), holds(F, T), \mathbf{not}\ ab(F, A, T)$.

  $not\_holds(F, T+1) \leftarrow occurs(A, T), not\_holds(F, T), \mathbf{not}\ ab(F, A, T)$.

- Proposition: Let D be a consistent domain description and O be an initial state complete set of observations such that (D, O) is consistent. Let f be a fluent.

  (i) $D \models_O f$ **after** $a_1, \ldots, a_n$ iff
  $\pi_{2.nar}(D, O) \cup \{occurs(a_1, 1), \ldots, occurs(a_n, n)\} \models holds(f, n+1)$.

  (ii) $D \models_O \neg f$ **after** $a_1, \ldots, a_n$ iff
  $\pi_{2.nar}(D, O) \cup \{occurs(a_1, 1), \ldots, occurs(a_n, n)\} \models not\_holds(f, n+1)$

From reasoning with narratives to Answer Set Planning: $\pi_{2.planning}$

- We have all the rules from the narrative reasoning program $\pi_{2.nar}$.
- Choice rules: We have the following choice rules that make sure that one and only one action occurs at each time point up to l.

  $not\_occurs(A,T) \leftarrow occurs(B,T), A \neq B.$

  $occurs(A,T) \leftarrow T \leq l, \textbf{not}\ not\_occurs(A,T).$
- Goal: Finally we have the following constraint, for our goal h.

  $\leftarrow \textbf{not}\ holds(h, l+1).$

- Proposition: Let D be a consistent domain description, O be an initial state complete set of observations such that (D, O) is consistent, l be the length of the plan that we are looking for, and h be a fluent which is the goal.

  (i) If there is a sequence of actions $a_1, \ldots, a_l$ such that $D \models_O h$ **after** $a_1, \ldots, a_l$ then there exists a consistent answer of $\pi_{2.planning}(D, O, h, l)$ containing $\{occurs(a_1, 1), \ldots, occurs(a_l, l)\}$ as the only facts about $occurs$.

  (ii) If there exists a consistent answer of $\pi_{2.planning}(D, O, h, l)$ containing $\{occurs(a_1, 1), \ldots, occurs(a_l, l)\}$ as the facts about $occurs$ then $D \models_O h$ **after** $a_1, \ldots, a_l$.

More on reasoning about actions and planning

- Allowing executability conditions: **executable** *drive* **if** *has_car*
- Allowing static constraints:
  - *move_to*(*Y*) **causes** *at*(*Y*).
  - A person can only be at one place at one time.
  - **initially** *at*(*A*).
  - Impact of *move_to*(*B*)?
  - *marry*(*Y*) **causes** *married_to*(*Y*)
  - A person can only be married to one person at one time.
  - **initially** *married_to*(*A*).
  - Impact of *marry*(*B*)?
  - Classical logic: Both constraint written the same way.
  - In AnsProlog*:
    $\neg at(Y) \leftarrow at(X), X \neq Y.$
    $\leftarrow married\_to(X), married\_to(Y), X \neq Y.$

- Concurrent actions: when to inherit and when not to.

  $\{close\}$ **causes** $\neg opened$

  $\{open\}$ **causes** $opened$

  $\{paint\}$ **causes** $painted$.

- Planning with hierarchical (HTN), temporal (LTL) and procedural constraints (Golog).
  - HTN: hierarchical task networks
  - LTL: Linear temporal logic.
  - Golog.
- Agent architecture based on observation assimilation, projection and planning from the current situation.
  - Observe the world and add the observations to the agent's set of observations (O).
  - Construct a plan from the current moment of time to achieve the goal.
  - Execute the first action of the plan and add this execution as an observation to the set O.

- Action based explanation: Explaining observations by what might have happened.
- Action based diagnosis
  - Assume that initially all components were fine.
  - Find missing action occurrences that explain the observations.
  - Determine what components are bad in the current situation.

COMPUTING ANSWER SETS

Direct algorithms

- Branch-n-bound using Well-founded semantics
- SLG approach
- Smodels algorithm
- DLV algorithm
- Algorithms based on graphs and colorings

Well-founded semantics for AnsProlog programs

- For an AnsProlog program Π, let $\Gamma_\Pi(S)$ denote the answer set of Π^S.
- Thus S is an answer set of Π if $S = \Gamma_\Pi(S)$.
- Well-founded semantics of AnsProlog programs is given by $\{lfp(\Gamma^2_\Pi), gfp(\Gamma^2_\Pi)\}$
- I.e., for an atom p we have $\Pi \models_{wf} p$ iff $p \in lfp(\Gamma^2_\Pi)$ and $\Pi \models_{wf} \neg p$ iff $p \notin gfp(\Gamma^2_\Pi)$.
- Since fixpoints of Γ_Π are also fixpoints of Γ^2_Π, the well-founded semantics is an approximation of the answer set semantics for AnsProlog programs.
- Proposition: Let Π be an AnsProlog program and A be an atom.
  (i) $A \in lfp(\Gamma^2_\Pi)$ implies $\Pi \models A$.
  (ii) $A \notin gfp(\Gamma^2_\Pi)$ implies $\Pi \not\models A$.
- $p \leftarrow a.$
  $p \leftarrow b.$
  $a \leftarrow$ **not** $b.$
  $b \leftarrow$ **not** $a.$

wfs-bb: an illustration

- Consider the following program P:

  $r_1 : a \leftarrow.$
  $r_2 : b \leftarrow a.$
  $r_3 : d \leftarrow \textbf{not}\ e.$
  $r_4 : e \leftarrow \textbf{not}\ d, c.$
  $r_5 : f \leftarrow g, a.$
  $r_6 : g \leftarrow f, d.$
  $r_7 : h \leftarrow \textbf{not}\ h, f.$
  $r_8 : i \leftarrow \textbf{not}\ j, b.$
  $r_9 : j \leftarrow \textbf{not}\ i, \textbf{not}\ c.$
  $r_a : k \leftarrow \textbf{not}\ l, i.$
  $r_b : l \leftarrow \textbf{not}\ k, j.$

- WFS $= \langle \{a, b, d\}, \{c, e, f, g, h\} \rangle$

Answer set computation using SAT solvers

- Basic Idea: Compile the program to an appropriate propositional theory and compute its models.
- Tight programs: An AnsProlog program Π is said to be *tight (or positive order consistent)*, if there exists a function λ from HB_Π to the set of natural numbers such that for every $L_0 \leftarrow L_1, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n$ in $ground(\Pi)$, and for every $1 \leq i \leq m$: $\lambda(L_0) > \lambda(L_i)$.
- Proposition: For any propositional AnsProlog program, if Π is tight then X is an answer set of Π iff X is a model of $Comp(\Pi)$.
- An AnsProlog$^{\neg,\perp}$ program Π is said to be *tight on a set X of literals*, if there exists a partial mapping λ with domain X from literals to the set of natural numbers such that for every rule $L_0 \leftarrow L_1, \ldots, L_m, \textbf{not } L_{m+1}, \ldots, \textbf{not } L_n$ in $ground(\Pi)$, if $L_0, \ldots, L_m \in X$, then for every $1 \leq i \leq m$: $\lambda(L_0) > \lambda(L_i)$.
- $p \leftarrow p$.

  This program is obviously not tight. But it is tight on the set of literals $\{\}$.

- Proposition: For any AnsProlog$^{\neg,\bot}$ program Π and any consistent set X of literals such that Π is tight on X, X is an answer set of Π iff X is closed under and supported by Π.
- Proposition: For any propositional AnsProlog program and any set X of atoms such that Π is tight on X, X is an answer set of Π iff X is a model of $Comp(\Pi)$.
- What if the above two approaches do not work?
  - Transform the program to a tight program. (Lin and Reiter.)
  - Transform the program by finding strongly connected components and adding rules corresponding to it to make the program inherently tight – a necessary and sufficient condition for the equivalence between answer sets and models of $Comp(\Pi)$. (Lin and Jicheng Zhao.)
  - Add additional propositional formulas to $Comp(\Pi)$.
    - Ben-Eliyahu and Dechter find strongly connected components and use it to construct new formulas.
    - Lin and Y. Zhao find positive loops and add loop formulas.

Loop formulas

- Definition: A set of atoms L is said to be a loop in a program Π, if for every pair p, q in L, there is a positive path from p to q in the dependency graph of Π.
- Definition: Let L be a loop with respect to a program Π, then $R^+(L, \Pi)$, the set of rules in Π with head in L that are involved in L is $\{r \mid r \in \Pi, head(r) \in L, body(r) \cap L \neq \emptyset\}$.
- Definition: Let L be a loop with respect to a program Π, then $R^-(L, \Pi)$, the set of rules in Π with head in L that are not involved in L is $\{r \mid r \in \Pi, head(r) \in L, body(r) \cap L = \emptyset\}$.
- Intuition behind loop formulas: "If none of the atoms in the loop can be proved using other rules, then these atoms must be false."

- Definition: Let $L = \{p_1, \ldots, p_n\}$ be a loop with respect to a program Π, and let $R^-(L, \Pi)$ consist of the following rules:

  $p_1 \leftarrow body_{11}. \ldots p_1 \leftarrow body_{1k_1}.$

  $\vdots$

  $p_n \leftarrow body_{1n}. \ldots p_1 \leftarrow body_{1k_n}.$

  Then the loop formula corresponding to L is

  $\neg(body_{11} \vee body_{1k_1} \vee \ldots \vee body_{n1} \vee body_{nk_n}) \supset (\neg p_1 \wedge \ldots \wedge \neg p_n)$

- Proposition: Let Π be an AnsProlog program and $LF(\Pi)$ be the set of all the loop formulas for all the loops in Π. S is an answer set of Π iff S is a model of $Comp(\Pi) \cup LF(\Pi)$.
- Consider the program $p \leftarrow p$.

  $Comp(\Pi) = \{p \Leftrightarrow p\}$.
- The number of loop formulas may be exponential in the size of the program.
- ASSAT does not find all the loop a-priori.

So why not use Propositional logic in the first place?

- Propositional logic is good for machines in that it is easier to find models for it.
- But AnsProlog is good for people in terms of ease of knowledge representation; is more elaboration tolerant; is non-monotonic. ...
- It is proven that AnsProlog can not be translated to propositional logic with an exponential blow up in the worst case.
- Best of both worlds: Program in AnsProlog; Compile to propositional logic to obtain answer sets whenever it is necessary and is feasible.

ANSWER SET PROGRAMMING SYSTEMS

Some existing systems

- Smodels: Core syntax is AnsProlog; has additional features.
- DLV: Core syntax is AnsProlog^{or} ; has additional features.
- ASSAT: Core syntax is AnsProlog.
- Prolog
- Cmodels
- NoMoRe

Programming using Smodels: http://www.tcs.hut.fi/Software/smodels/

- Graph Colorability: The graph is described by the facts with respect to predicates *vertex* and *edge* and the given set of colors are expressed using the predicate *col*.

```
vertex(1..4).
edge(1,2). edge(1,3). edge(2,3).
edge(1,4). edge(2,4).
% edge(3,4).
col(a;b;c).
1 { color(X,C) : col(C) } 1 :- vertex(X).
:- edge(X,Y), col(C), color(X,C), color(Y,C).
:- edge(Y,X), col(C), color(X,C), color(Y,C).
hide col(X). hide vertex(Y). hide edge(X,Y).
```

- Knapsack problem: We have five items (1-5) and each item has a cost (or size) and value associated with it. We have a sack with a given capacity (12) and the goal is to select a subset of the items which can fit the sack while maximizing the total value.

  item(1..5).

  weight val(1) = 5. weight val(2) = 6. weight val(3) = 3. weight val(4) = 8.

  weight val(5) = 2.

  weight cost(1) = 4. weight cost(2) = 5. weight cost(3) = 6. weight cost(4) = 5.
  weight cost(5) = 3.

  in_sack(X) :- item(X), not not_in_sack(X).

  not_in_sack(X) :- item(X), not in_sack(X).

  val(X) :- item(X), in_sack(X).

  cost(X) :- item(X), in_sack(X).

  cond1 :- [cost(X) : item(X)] 12.

  :- not cond1.

  maximize { val(X) : item(X) }.

  hide item(X). hide not_in_sack(X). hide cost(X). hide val(X).

- Single Unit Combinatorial Auction: Bidders are allowed to bid on a bundle of items. The auctioneer has to select a subset of the bids so as to maximize the price it gets, and making sure that it does not accept multiple bids that have the same item as each item can be sold only once.

  The auctioneer has the set of items $\{1, 2, 3, 4\}$, and the buyers submit bids $\{a, b, c, d, e\}$ where a constitutes of $\langle\{1, 2, 3\}, 24\rangle$, meaning that the bid a is for the bundle $\{1, 2, 3\}$ and its price is \$24. and so on.

  bid(a;b;c;d;e).

  item(1..4).

  in(1,a). in(2,a). in(3,a). in(2,b). in(3,b). in(3,c). in(4,c). in(2,d). in(3,d). in(4,d). in(1,e). in(4,e).

  weight sel(a) = 24. weight sel(b) = 9. weight sel(c) = 8. weight sel(d) = 25. weight sel(e) = 15.

  sel(X) :- bid(X), not not_sel(X).

  not_sel(X) :- bid(X), not sel(X).

  :- bid(X), bid(Y), sel(X), sel(Y), not eq(X,Y), item(I), in(I,X), in(I,Y).

  maximize [sel(X) : bid(X)].

  hide bid(X). hide not_sel(X). hide item(X). hide in(X,Y).

Programming using DLV: http://www.dbai.tuwien.ac.at/proj/dlv/

- Single Unit Combinatorial auction using weak constraints

  bid(a). bid(b). bid(c). bid(d). bid(e).

  in(1,a). in(2,a). in(3,a). in(2,b). in(3,b). in(3,c). in(4,c). in(2,d). in(3,d). in(4,d). in(1,e). in(4,e).

  sel(X) v not_sel(X) :- bid(X).

  :- sel(X), sel(Y), X != Y, in(I,X), in(I,Y).

  :∼ not sel(a). [24:1]

  :∼ not sel(b). [9:1]

  :∼ not sel(c). [8:1]

  :∼ not sel(d). [25:1]

  :∼ not sel(e). [15:1]

Programming Tricks

- Neither Smodels nor DLV allow function symbols or explicit sets and lists.
- Aggregation in Smodels

  sold(a, 10, jan1). sold(a, 21, jan5). sold(a, 15, jan16). sold(b, 16, jan4). sold(b, 31, jan21). sold(b, 15, jan26). sold(c, 24, jan8).

  item(a;b;c). number(1..100). date(jan1;jan5;jan16;jan4;jan21;jan26;jan8).

  weight sold(X,Y,Z) = Y.

  total_sold(I, N) :- item(I), number(N), N [sold(I, X, D) : number(X) : date(D)] N.

  total_sell_transactions(I, N) :- item(I), number(N),
  N { sold(I, X, D) : number(X) : date(D) } N.

- Sets

  causes(a, f, s).

  set(s).

  in(p1, s).

  ...

  in(pn, s).

- Lists of arbitrary depth: $p([a, [b, [c, d])$

  p(l).

  head(l,a). body(l, l1).

  head(l1, b). body(l1, l2).

  head(l2, c). body(l2, l3).

  head(l3, d). body(l3, nil).

Correct query answering using Prolog

- Pure Prolog: Syntax of AnsProlog and semantics of Prolog.
- Pure Prolog semantics is based on

  (i) SLDNF resolution with the leftmost selection rule (i.e., LDNF resolution) where by only the leftmost literal in each query is marked selected;

  (ii) but ignoring floundering;

  (iii) omitting occur check during unification; and

  (iv) selecting input rules during resolution from the beginning of the program to the end.

- Theorem: If an AnsProlog program Π is well moded for some input-output specification and there is no rule in Π whose head contains more than one occurrence of the same variable in its output positions then Π is occur check free with respect to any ground query.
- Theorem: If an AnsProlog program Π is well moded for some input-output specification and all predicate symbols occurring under **not** are moded completely by input then a ground query to Π does not flounder.

- Proposition: If Π is an acceptable AnsProlog program and Q is a ground query then all SLDNF derivations – with left most selection – of Q (with respect to Π) are finite and therefore the Pure Prolog interpreter terminates on Q.
- Summary: One way to show that the Pure prolog semantics of a program agrees with the AnsProlog semantics is by showing that the conditions in above three results are satisfied.

COMPLEXITY, EXPRESSIVENESS AND RELATIONS WITH OTHER LANGUAGES

Complexity vs Expressiveness

- Let L be a sub-class of AnsProlog*, Π be a program in L, D_{in} be a set of ground facts in L, and A be a literal.
  - *Data complexity* of L: Complexity of checking $D_{in} \cup \Pi \models A$, in terms of the length of the input $\langle D_{in}, A \rangle$, given a fixed Π in L.
  - *Program complexity* of L: Complexity of checking $D_{in} \cup \Pi \models A$, in terms of the length of the input $\langle \Pi, A \rangle$, given a fixed D_{in} in L.
  - *Combined complexity* of L: Complexity of checking $D_{in} \cup \Pi \models A$, in terms of the length of the input $\langle \Pi, D_{in}, A \rangle$.
- Given an AnsProlog* program Π the recognition problem associated with it is to determine if $\Pi \cup I \models A$, given some facts I and a ground literal A. Alternatively, the recognition problem of Π is to determine membership in the set $\{\langle I, A \rangle \ : \ \Pi \cup I \models A\}$.
- An AnsProlog* program Π is said to be in complexity class C if the recognition problem associated with Π is in C.

- An AnsProlog* sub-class L is *data-complete for complexity class C* (or equivalently, the data complexity class of L is C-complete) if

  (i) *(membership)*: each program in L is in C, and

  (ii) *(hardness)*: there exists a program in L for which the associated recognition problem is complete with respect to the class C.

- Expressiveness: An AnsProlog* sub-class L is said to *capture the complexity class C* if

  (i) each program in L is also in C, and

  (ii) every problem of complexity C can be expressed in L.

- L is data-complete in C does not imply that L captures C.

  Even though there may exist queries in L for which the associated recognition problem is complete with respect to the class C, there may be problems in the class C which can not be expressed in L.

- L captures C does not imply that L is data-complete in C.

  This is because even though L captures C there may not exist a problem that is C-complete.

Summary of the complexity results of AnsDatalog* sub-classes

AnsDatalog* Class	Complexity Type	Complexity Class
AnsDatalog$^{\neg}$**not**	Data complexity	P-complete
AnsDatalog$^{\neg}$**not**	Program complexity	EXPTIME-complete
Stratified AnsDatalog	Data complexity	P-complete
Stratified AnsDatalog	Program complexity	EXPTIME-complete
AnsDatalog (under WFS)	Data complexity	P-complete
AnsDatalog (under WFS)	Program complexity	EXPTIME-complete
AnsDatalog (answer set existence)	complexity of $SM(\Pi) \neq \emptyset$	**NP**-complete
AnsDatalog	Data complexity	**coNP**-complete
AnsDatalog	Program complexity	coNEXPTIME complete
AnsDatalog$^{\neg}$	Existence of answer set	**NP**-complete
AnsDatalog$^{\neg}$	Data complexity	**coNP**-complete
AnsDatalog$^{or,\neg}$**not**	Deciding $\Pi \models_{GCWA} A$	**coNP**-complete
AnsDatalog$^{or,\neg}$**not**	Deciding $\Pi \models_{GCWA} \neg A$	Π_2P-complete
AnsDatalog^{or}	Data complexity	Π_2P-complete
AnsDatalog^{or}	Program complexity	coNEXPTIME$^{\mathbf{NP}}$-comp.

Summary of expressiveness of AnsDatalog* sub-classes

AnsDatalog* Sub-class	Relation	Complexity Class (or a non-AnsProlog* class)
Datalog$^+$	$\subset\neq$	P
Datalog$^+$ (on ordered databases)	captures	P
Datalog$^+$	equal	$FPL^+(\exists)$
Stratified AnsDatalog	$\subset\neq$	FPL
Non-recursive range restr AnsDatalog	equal	relational algebra
Non-recursive range restr AnsDatalog	equal	relational calculus
Non-recursive range restr AnsDatalog	equal	FOL (without function symbols
AnsDatalog (under WFS)	equal	FPL
Stratified AnsDatalog (on ordered databases)	captures	P
AnsDatalog under WFS (on ordered databases)	captures	P
AnsDatalog under brave semantics	captures	**NP**
AnsDatalog	captures	**coNP**
AnsDatalog $^{or,\text{–}\mathbf{not},\neq}$ (under brave semantics)	captures	Σ_2P
AnsDatalog $^{or,\text{–}\mathbf{not},\neq}$	captures	Π_2P
AnsDatalog ^{or} (under brave semantics)	captures	Σ_2P
AnsDatalog ^{or}	captures	Π_2P

Complexity and expressiveness of programs with function symbols

AnsProlog* Class (With Functions)	Complexity type	Complexity Class
AnsProlog$^{\neg\mathbf{not}}$	complexity	r.e. complete
AnsProlog$^{\neg\mathbf{not}}$ (without recursion)	complexity	NEXPTIME-complete
AnsProlog$^{\neg\mathbf{not}}$ (with restrictions[1])	complexity	PSPACE-complete
Stratified AnsProlog (n levels of stratification)	complexity	Σ^0_{n+1}-complete
Non-recursive AnsProlog	Data complexity	P
AnsProlog (under WFS)	complexity	Π^1_1-complete
AnsProlog	complexity	Π^1_1 complete
AnsProlog$^{or,\neg\mathbf{not}}$	under GCWA	Π^0_2 complete
AnsProlog^{or}	complexity	Π^1_1-complete

AnsProlog* Class	relation	Complexity Class
AnsProlog (under WFS)	captures	Π^1_1
AnsProlog	captures	Π^1_1
AnsProlog^{or}	captures	Π^1_1

AnsProlog* vs Classical logic

- Propositional theories can be easily mapped to AnsProlog.
- There is no rule-modular mapping from propositional AnsProlog to propositional logic.
  - A mapping $T(.)$ from the language L_1 to L_2 is said to be *rule-modular* if for any theory (or program) Π in L_1, for each set (possibly empty) of atomic facts F, the "models" of $\Pi \cup F$ and $T(\Pi) \cup F$ coincide.
- Classical logic vs AnsProlog*
  - Since the semantics of AnsProlog* is based on the Herbrand universe and answer sets of AnsProlog programs are Herbrand Interpretations there is often a mismatch between classical theories and AnsProlog* programs if we do not restrict ourselves to Herbrand models.

- $T_1 : ontable(a) \wedge ontable(b)$
- Π_1 obtained by translating T_1
  $ontable(X) \leftarrow$ **not** $n\_ontable(X)$.
  $n\_ontable(X) \leftarrow$ **not** $ontable(X)$.
  $good\_model \leftarrow ontable(a), ontable(b)$.
  $\leftarrow$ **not** $good\_model$.
  Query Q given by $\forall X.ontable(X)$.
- $\Pi_1 \models \forall X.ontable(X)$.
- But $T_1 \not\models \forall X.ontable(X)$.
- Note: While using resolution with respect to T_1 and Q, the clauses obtained from $T_1 \cup \{\neg Q\}$ is the set $\{ontable(a), ontable(b), \neg ontable(c)\}$ – where c is a skolem constant, which does not lead to a contradiction.
- Way out: In AnsProlog* judiciously introduce skolem constants derived by transforming the query into a clausal form as done during resolution.

Description logic vs AnsProlog*

- A straightforward translation of $\{child = \exists\ childof, \quad son = male \sqcap child\}$ to AnsProlog would be the following rules.

  $r_1 : child(X) \leftarrow childof(X, Y).$
  $r_2 : son(X) \leftarrow male(X), child(X).$

- But while $\{child = \exists\ childof, son = male \sqcap child, son(a)\} \models^{dl} child(a)$, $\{r_1, r_2, son(a)\} \not\models child(a)$.

- Besides r_1 and r_2, a correct AnsProlog translation will consist of the following rules:

  $r_3 : male(X) \leftarrow top(X), \mathbf{not}\ female(X).$
  $r_4 : female(X) \leftarrow top(X), \mathbf{not}\ male(X).$
  $r_5 : childof(X, Y) \leftarrow top(X), top(Y), X \neq Y, \mathbf{not}\ not\_childof(X, Y).$
  $r_6 : not\_childof(X, Y) \leftarrow top(X), top(Y), X \neq Y, \mathbf{not}\ childof(X, Y).$
  $r_7 : \ \leftarrow childof(X, Y), childof(Y, X).$

  and facts about the predicate *top* listing all elements belonging to the concept $\top$.

- Now to assimilate $son(a)$ we need to add the following:

  $r_8 : \leftarrow \textbf{not}\ son(a).$

  whose effect is to eliminate all potential answer sets which do not contain $son(a)$.

CONCLUSION AND SOME FUTURE DIRECTIONS

Conclusion

- Being able to represent and reason with knowledge is one of the two key components of AI systems.
- AnsProlog* is a good candidate as the 'core' of a knowledge representation language – satisfies all the 'desirable' criteria that we listed.
- The compilation to propositional logic approach of computing answer sets adds a new dimension to their fast computation.
- Still lot more needs to be done.

Emerging and Future directions

- Further development of the language. The 'core' is there, need features such as cardinality constraints, sets, aggregates, preference between rules, etc.
- As language is broadened, need to expand the building block results.
- Need better systems: Smodels and dlv work only with function free programs, and Prolog works with a very restricted subset.
- Integration of probability with logic.
- Integration with other knowledge representation languages and/or borrowing their features: description logic, constraints of the kind in CLP, higher level syntax such as in F-logic, function symbols and numbers as in Prolog.
- Discovering efficient encodings for polynomially solvable problem.
- Emerging applications: Semantics Web, data and knowledge integration.
- Inductive AnsProlog* or learning AnsProlog* rules from data; using knowledge encoded in AnsProlog* during learning.
- Using it in more, bigger and eye-catching application domains.