Answer Set Programming

Matthias Knorr

Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, Portugal
mkn@fct.unl.pt

November 17, 2020

Outline

Outline

Some Historical Remarks

- In the 1950s, John McCarthy expressed the need to use logic-based languages for representing and reasoning about knowledge
- First attempts used classical logic of the predicate calculus (First-Order Logic)
  - well-defined semantics
  - well-understood inference mechanism
  - expressive power capable of representing mathematical knowledge
- However, common-sense reasoning is inherently non-monotonic, leading to the development of non-monotonic logics (late 1970s and 1980s)
  - circumscription
  - default logic
  - non-monotonic modal logics

Some Historical Remarks

- Also in the 1970s, others were investigating the idea of combining logic as a representation language with the theory of automated deduction.
- Kowalski and Colmerauer et al. defined and implemented the first PROLOG interpreter, based on a model-theoretic, fixpoint and operational semantics for the Horn-clause fragment.
- The beginning of the paradigm of Logic Programming
- Formal foundations of LP during late 1970s:
  - least model semantics (van Emden and Kowalski)
  - first PROLOG compiler (Warren)
  - program completion (Clark)
  - closed world assumption (Reiter) - leading to negation-as-finite-failure in PROLOG

Some Historical Remarks

Logic Programming introduced Declarative Programming in Computer Science.

- Procedural Language: specify how
- Declarative Language: specify what

Algorithm = Logic + Control (Kowalski, 1979)

Some Historical Remarks

Features of Prolog (Colmerauer, Kowalski)

- Declarative (relational) programming language
- Based on SLD(NF) Resolution
- Top-down query evaluation
- Terms as data structures
- Parameter passing by unification
- Solutions are extracted from instantiations of variables occurring in the query

Some Historical Remarks

- Prolog is only almost declarative! To see this, consider:

```
above(X, Y) :- on(X, Y).
above(X, Y) :- on(X, Z), above(Z, Y).
```

  and compare it to

```
above(X, Y) :- above(Z, Y), on(X, Z).
above(X, Y) :- on(X, Y).
```

- An interpretation in classical logic amounts to

$$\forall xy(on(x, y) \vee \exists z(on(x, z) \wedge above(z, y)) \supset above(x, y))$$

Some Historical Remarks

- Prolog offers negation as failure via operator not.
- But, for instance,

```
info(a).
ask(X) :- not info(X).
```

  cannot be captured by

$$info(a) \land \forall x(\neg info(x) \supset ask(x))$$

- but by appeal to Clark's completion by

$$\forall x(x = a \equiv info(x)) \land \forall x(\neg info(x) \equiv ask(x)) \Leftrightarrow$$
$$\Leftrightarrow info(a) \land \forall x(x \neq a \equiv ask(x))$$

The idea of completion

- In LP one uses “if" but means “iff" [Clark78]

```
naturalN(0).
naturalN(s(Y)) :- naturalN(Y).
```

- This does not imply that -1 is not a natural number!
- With this program we mean:

$$naturalN(x) \Leftrightarrow \forall x(x = 0 \vee \exists y(x = s(y) \wedge naturalN(y)))$$

- This is the idea of Clark’s completion:
  - Syntactically transform if’s into iff’s
  - Use classical logic in the transformed theory to provide the semantics of the program

Completion Semantics

Definition (Program Completion)

The completion of P is the theory *comp*(P) obtained by:

- Replace $p(\vec{t}) \leftarrow \varphi$ by $p(\vec{x}) \leftarrow \vec{x} = \vec{t}, \varphi$
- Replace $p(\vec{x}) \leftarrow \varphi$ by $p(\vec{x}) \leftarrow \exists \vec{y} \varphi$, where $\vec{y}$ are the original variables of the rule
- Merge all rules with the same head into a single one $p(\vec{x}) \leftarrow \varphi_1 \vee \cdots \vee \varphi_n$
- For every $q(\vec{x})$ without rules, add $q(\vec{x}) \leftarrow \bot$
- Replace $p(\vec{x}) \leftarrow \varphi$ by $\forall \vec{x}(p(\vec{x}) \Leftrightarrow \varphi)$

Definition (Completion Semantics)

The completion semantics of P is given by the semantics of *comp*(P) where not is interpreted as classical negation.

- Though completion's definition is not that simple, the idea behind it is quite simple
- Also, it defines a non-classical semantics by means of classical inference on a transformed theory

SLDNF proof procedure

- By adopting completion, procedurally we have:
  - not is “negation as finite failure"

Definition (SLDNF Proof Procedure)

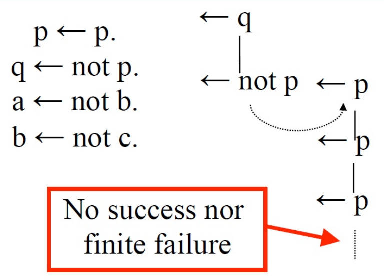
In SLDNF proceed as in SLD. To prove not A:

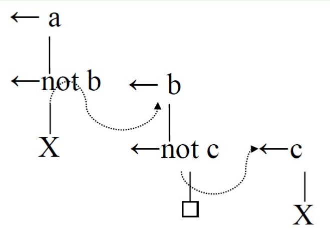
- If there is a finite derivation for A, fail not A
- If, after any finite number of steps, all derivations for A fail, remove not A from the resolvent (i.e. succeed not A)

- SLDNF can be efficiently implemented (cf. Prolog)

SLDNF example

Example





According to completion:

- $comp(P) \models \{not\ a,\ b,\ not\ c\}$
- $comp(P) \not\models p,\ comp(P) \not\models not\ p$
- $comp(P) \not\models q,\ comp(P) \not\models not\ q$

Problems with completion

- Some consistent programs may became inconsistent:

$$\texttt{p} \leftarrow \texttt{not p} \textit{ becomes } \texttt{p} \Leftrightarrow \texttt{not p}$$

- Does not correctly deal with deductive closures

$$\texttt{edge}(\texttt{a},\texttt{b}). \quad \texttt{edge}(\texttt{c},\texttt{d}). \quad \texttt{edge}(\texttt{d},\texttt{c}).$$
$$\texttt{reachable}(\texttt{a}).$$
$$\texttt{reachable}(\texttt{B}) \leftarrow \texttt{edge}(\texttt{A},\texttt{B}),\ \texttt{reachable}(\texttt{A}).$$

- Completion does not conclude not *reachable*(c), due to the circularity caused by *edge*(c, d) and *edge*(d, c)
- Circularity is a procedural concept, not a declarative one

Some Historical Remarks

- Clark's completion has other problems:
- For example:

```
bird(tweety).
fly(X) :- bird(X), not abnormal(X).
abnormal(X) :- irregular(X).
irregular(X) :- abnormal(X).
```

  ...does not allow the conclusion that tweety flies.
- Or even more complex yet analogous situations.
- An explanation would be: “the rules for abnormal and irregular cause a loop".
  - But looping is a procedural concept, not a declarative one, and should be rejected when defining declarative semantics

Some Historical Remarks

- While the Logic Programming community was developing PROLOG into a full-fledged Programming Language...
- Some devoted their time to the development of appropriate semantics for logic programs with negation.
- The 1980s and early 1990s saw “the war of the semantics", mainly focusing on the meaning of programs like:

```
a :- not b.          a :- not a.
b :- not a.
```

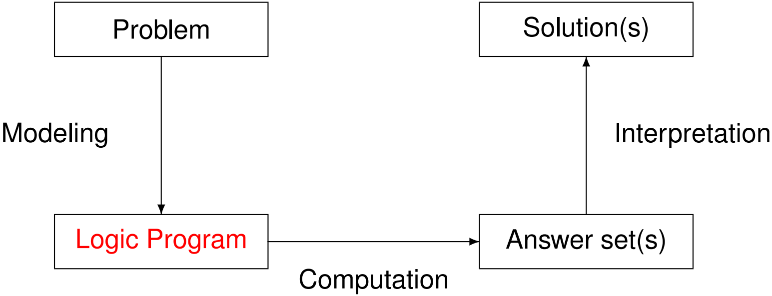
- Great Schism: Single-model vs. multiple-model semantics

LP for Knowledge Representation

- Due to its declarative nature, LP has become a prime candidate for Knowledge Representation and Reasoning
- This has been more noticeable since its relations to other NMR formalisms were established
- For this usage of LP, a precise declarative semantics was in order.
- To date:
  - Well-Founded Semantics by van Gelder et al. (1991).
  - Stable Model Semantics by Gelfond & Lifschitz (1988,1991).

Outline

Problem solving in ASP



Normal Logic Programs: Syntax

Definition (Rule)

A (normal) rule, r, is an ordered pair of the form

$$A_0 \leftarrow A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n$$

where $n \geq m \geq 0$, and each A_i $(0 \leq i \leq n)$ is an atom.

Definition (Logic Program)

A (normal) logic program is a finite set of rules.

Notation

$$\begin{array}{rcl} \mathit{head}(r) & = & A_0 \\ \mathit{body}(r) & = & \{A_1, \ldots, A_m, \mathit{not}\ A_{m+1}, \ldots, \mathit{not}\ A_n\} \\ \mathit{body}^+(r) & = & \{A_1, \ldots, A_m\} \\ \mathit{body}^-(r) & = & \{A_{m+1}, \ldots, A_n\} \end{array}$$

Normal Logic Programs: Syntax

Definition (Positive Logic Program)

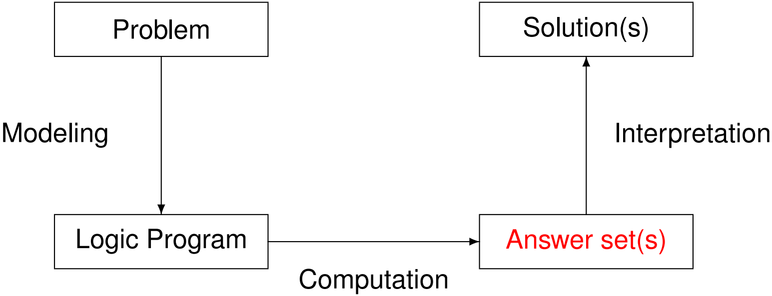
A program is called positive if $body^-(r) = \emptyset$ for all its rules.

Notation

We often use the following notation interchangeably in order to stress the respective view:

	if	and	or	negation as failure	classical negation
source code	`:-`	`,`	`|`	`not`	`-`
logic program	$\leftarrow$	,	;	*not*/$\sim$	$\neg$
formula	$\rightarrow$	$\wedge$	$\vee$		$\neg$

Problem solving in ASP: Semantics



Positive Logic Programs: Semantics

Definition (Closure)

A set of atoms X is closed under a positive program Π iff for any $r \in \Pi$, $head(r) \in X$ whenever $body^+(r) \subseteq X$.

- X corresponds to a model of Π (seen as a formula).

Definition ($Cn(\Pi)$)

The least (smallest) set of atoms which is closed under a positive program Π is denoted by $Cn(\Pi)$.

- $Cn(\Pi)$ corresponds to the $\subseteq$-least model of Π (seen as a formula).

Definition (Answer Set of a Positive Logic Program)

The set $Cn(\Pi)$ of atoms is the answer set of a *positive* program Π.

Some "logical" remarks

- Positive rules are also referred to as definite clauses.
  - Definite clauses are disjunctions with exactly one positive atom:

$$A_0 \vee \neg A_1 \vee \cdots \vee \neg A_m$$

  - A set of definite clauses has a (unique) smallest model.
- Horn clauses are clauses with at most one positive atom.
  - Every definite clause is a Horn clause but not vice versa.
  - A set of Horn clauses has a smallest model or none.
- This smallest model is the intended semantics of a set of Horn clauses.
  - ☞ Given a positive program Π, $Cn(\Pi)$ corresponds to the smallest model of the set of definite clauses corresponding to Π.

Another "logical" remark

Answer sets versus (minimal) models

- Program $\{a \leftarrow \mathit{not}\ b\}$ has answer set $\{a\}$.
- Clause $a \vee b$ (being equivalent to $a \leftarrow \neg b$)
  - has models $\{a\}$, $\{b\}$, and $\{a, b\}$,
  - among which $\{a\}$ and $\{b\}$ are minimal.

☞ The negation-as-failure operator *not* makes a difference!

Normal Logic Programs: Semantics

Informally, a set of atoms X is an answer set of a logic program Π

- if X is a (classical) model of Π and
- if all atoms in X are justified by some rule in Π
  - rooted in intuitionistic logics HT (Heyting, 1930) and G3 (Gödel, 1932))

Example

Consider the logical formula Φ and its three (classical) models:

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}.$$

This formula has one answer set:

$$\{p, q\}$$

Φ $\boxed{q \;\wedge\; (q \wedge \neg r \rightarrow p)}$

Π

$$\boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q,\ \mathit{not}\ r \end{array}}$$

Answer set: Basic idea

For instance, interpreting

$$\left\{ \begin{array}{lcl} b & \leftarrow & \\ a & \leftarrow & b,\ not\ c \end{array} \right\} \quad \text{as} \quad b \wedge (b \wedge \neg c \rightarrow a), \text{ that is, } b \wedge (a \vee c)\ ,$$

we obtain

- 3 models: $\{a, b\}$, $\{b, c\}$, and $\{a, b, c\}$,
- 2 minimal models: $\{a, b\}$ and $\{b, c\}$, and
- 1 stable model: $\{a, b\}$ ✔ $\Longleftrightarrow$ answer set

Informally, a set of atoms X is an answer set of a logic program Π

- if X is a minimal (classical) model of Π [1] and
- if all atoms in X are *justified* by some rule in Π.

[1] That is, interpreting '$\leftarrow$', ',', and '*not*' as in classical logic.

Normal Logic Programs: Semantics

Definition (GL-Reduct (Gelfond and Lifschitz 1988))

The reduct, Π^X, of a program Π relative to a set of atoms X is given by

$$\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi \text{ and } body^-(r) \cap X = \emptyset\}.$$

Intuitively, given a set of atoms X from Π, Π^X is obtained from Π by:

1. deleting each rule having a *not* A in its body with $A \in X$, and then
2. deleting all negative atoms of the form *not* A in the bodies of the remaining rules.

Definition (Answer Set of a Normal Logic Program)

A set X of atoms is an answer set of a program Π iff $Cn(\Pi^X) = X$.

Intuition: X is stable under *"applying rules from* Π*"*

Note: Every atom in X is justified by an *"applicable rule from* Π*"*

Normal Logic Programs: Examples

Example (First Example)

Consider the program

$$\Pi = \{p \leftarrow p \qquad q \leftarrow not\ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✘
$\{p\}$	$p \leftarrow p$	$\emptyset$	✘
$\{q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✔
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	✘

Normal Logic Programs: Examples

Example (Even Loop)

Consider the program

$$\Pi = \{p \leftarrow \mathit{not}\ q \qquad q \leftarrow \mathit{not}\ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	✘
$\{p\}$	$p \leftarrow$	$\{p\}$	✔
$\{q\}$	$q \leftarrow$	$\{q\}$	✔
$\{p, q\}$		$\emptyset$	✘

Normal Logic Programs: Examples

Example (Odd Loop)

Consider the program

$$\Pi = \{p \leftarrow not\ p\}$$

X	Π^X	$Cn(\Pi^X)$	
$\emptyset$	$p \leftarrow$	$\{p\}$	✘
$\{p\}$		$\emptyset$	✘

Answer Sets: Properties

Property

If X is an answer set of a logic program Π, then X is a model of Π (seen as a formula).

Property (Minimality)

Every answer set X of Π is a minimal model of Π (wrt. $\subseteq$).

Property (Supportedness)

If X is an answer set of a logic program Π, and $p \in X$, then $\exists r \in \Pi$ such that $head(r) = p$ and $body^-(r) \cap X = \emptyset$ and $body^+(r) \subseteq X$.

Answer Sets: Alternative Definition

Definition (Modified-Reduct (Faber et al. 2004))

The modified reduct, Π_X, of a program Π relative to a set of atoms X is given by

$$\Pi_X = \{r \in \Pi \mid body^+(r) \subseteq X \text{ and } body^-(r) \cap X = \emptyset\}.$$

Intuitively, given a set of atoms X from Π, Π_X (dubbed the set of generating rules of X wrt. Π) is obtained from Π by:

1. deleting each rule having a body literal that is false w.r.t. X.

Definition (Answer Set of a Normal Logic Program - Alternative)

A set X of atoms is an answer set of a program Π iff $X \in \min_{\subseteq}(\Pi_X)$, where $\min_{\subseteq}(\Pi)$ is the set of minimal models of a program Π (wrt. $\subseteq$).

Theorem (Soundness and completeness of the Alternative Definition)

$$X \in \min_{\subseteq}(\Pi_X) \quad \textit{iff} \quad Cn(\Pi^X) = X$$

Example: Even Loop Revisited

Example (Even Loop)

Consider the program

$$\Pi = \{p \leftarrow not\ q \qquad q \leftarrow not\ p\}$$

X	Π_X	$\min_\subseteq(\Pi_X)$	
$\emptyset$	$p \leftarrow not\ q$ $q \leftarrow not\ p$	$\{p\}, \{q\}$	✘
$\{p\}$	$p \leftarrow not\ q$	$\{p\}, \{q\}$	✔
$\{q\}$	$q \leftarrow not\ p$	$\{p\}, \{q\}$	✔
$\{p, q\}$		$\emptyset$	✘

A closer look at Cn

Definition (Immediate Consequence Operator)

Let Π be a positive program and X a set of atoms. The immediate consequence operator T_Π is defined as follows:

$$T_\Pi((X)) = \{head(r) \mid r \in \Pi \text{ and } body(r) \subseteq X\}$$

Let $T^0_\Pi(X) = X$ and $T^i_\Pi(X) = T_\Pi\left(T^{i-1}_\Pi(X)\right)$.
Further let $T_\Pi \uparrow^\omega = \bigcup_{i=0}^{\infty} T^i_\Pi(\emptyset)$.

Theorem

Let Π be a positive program. Then:

- $Cn(\Pi) = T_\Pi \uparrow^\omega$.
- $X \subseteq Y$ *implies* $T_\Pi(X) \subseteq T_\Pi(Y)$.
- $Cn(\Pi)$ *is the least fixpoint of* T_Π.

Immediate Consequence Operator: Example

Example

$$\Pi = \{p \leftarrow \qquad q \leftarrow \qquad r \leftarrow p \qquad s \leftarrow q, t \qquad t \leftarrow r \qquad u \leftarrow v\}$$

$$\begin{array}{lclclcl}
T^0_\Pi(\emptyset) & = & \emptyset & & & & \\
T^1_\Pi(\emptyset) & = & \{p, q\} & = & T_\Pi\left(T^0_\Pi(\emptyset)\right) & = & T_\Pi(\emptyset) \\
T^2_\Pi(\emptyset) & = & \{p, q, r\} & = & T_\Pi\left(T^1_\Pi(\emptyset)\right) & = & T_\Pi(\{p, q\}) \\
T^3_\Pi(\emptyset) & = & \{p, q, r, t\} & = & T_\Pi\left(T^2_\Pi(\emptyset)\right) & = & T_\Pi(\{p, q, r\}) \\
T^4_\Pi(\emptyset) & = & \{p, q, r, t, s\} & = & T_\Pi\left(T^3_\Pi(\emptyset)\right) & = & T_\Pi(\{p, q, r, t\}) \\
T^5_\Pi(\emptyset) & = & \{p, q, r, t, s\} & = & T_\Pi\left(T^4_\Pi(\emptyset)\right) & = & T_\Pi(\{p, q, r, t, s\}) \\
T^6_\Pi(\emptyset) & = & \{p, q, r, t, s\} & = & T_\Pi\left(T^5_\Pi(\emptyset)\right) & = & T_\Pi(\{p, q, r, t, s\})
\end{array}$$

To see that $Cn(\Pi) = \{p, q, r, t, s\}$ is the smallest fixpoint of T_Π, note that $T_\Pi(\{p, q, r, t, s\}) = \{p, q, r, t, s\}$ and $T_\Pi(X) \neq X$ for every $X \subset \{p, q, r, t, s\}$.

Logic Programs with Variables

Definition (Alphabet)

Let Π be a logic program.

- Herbrand Universe U^{Π}: Set of constants in Π.
- Herbrand Base B^{Π}: Set of (variable-free) atoms constructible from U^{Π}. We usually denote B^{Π} as $\mathcal{A}$ and call it Alphabet

Definition (Grounding of a rule)

Let Π be a logic program (with variables). The ground instantiation of a rule $r \in \Pi$ is the set of variable-free rules obtained by replacing all variables in r by elements from U^{Π}:

$$ground(r) = \{r\theta \mid \theta : var(r) \to U^{\Pi}\}$$

where $var(r)$ stands for the set of all variables occurring in r and θ is a (ground) substitution.

Logic Programs with Variables

Definition (Grounding of a Program)

Let Π be a logic program (with variables). The Ground Instantiation of a program Π is the set of all ground instantiations of its rules

$$ground(\Pi) = \bigcup_{r \in \Pi} ground(r)$$

Definition (Answer Set a Logic Program with Variables)

Let Π be a normal logic program with variables. A set of ground atoms X (i.e. $X \subseteq B^{\Pi}$) is an answer set of Π iff X is an answer set of $ground(\Pi)$, i.e. iff

$$Cn(ground(\Pi)^{X}) = X$$

Logic Programs with Variables: Example

Example

Consider the program:

$$\Pi = \{\ r(a,b) \leftarrow \qquad r(b,c) \leftarrow \qquad t(X,Y) \leftarrow r(X,Y)\}$$

We have:

$$\begin{aligned} U^{\Pi} &= \{a,b,c\} \\ B^{\Pi} &= \left\{ \begin{array}{l} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c) \end{array} \right\} \end{aligned}$$

$$ground(\Pi) = \left\{ \begin{array}{lllllllll} r(a,b) & \leftarrow & & & & & & & \\ r(b,c) & \leftarrow & & & & & & & \\ t(a,a) & \leftarrow & r(a,a) & t(b,a) & \leftarrow & r(b,a) & t(c,a) & \leftarrow & r(c,a) \\ t(a,b) & \leftarrow & r(a,b) & t(b,b) & \leftarrow & r(b,b) & t(c,b) & \leftarrow & r(c,b) \\ t(a,c) & \leftarrow & r(a,c) & t(b,c) & \leftarrow & r(b,c) & t(c,c) & \leftarrow & r(c,c) \end{array} \right\}$$

Logic Programs with Variables: Example

Example

Consider the program:

$$\Pi = \{\ r(a,b) \leftarrow \qquad r(b,c) \leftarrow \qquad t(X,Y) \leftarrow r(X,Y)\}$$

We have:

$$U^{\Pi} = \{a,b,c\}$$

$$B^{\Pi} = \left\{ \begin{array}{l} r(a,a), r(a,b), r(a,c), r(b,a), r(b,b), r(b,c), r(c,a), r(c,b), r(c,c), \\ t(a,a), t(a,b), t(a,c), t(b,a), t(b,b), t(b,c), t(c,a), t(c,b), t(c,c) \end{array} \right\}$$

$$ground(\Pi) = \left\{ \begin{array}{l} r(a,b) \leftarrow \\ r(b,c) \leftarrow \\ \\ t(a,b) \leftarrow \\ t(b,c) \leftarrow \end{array} \right\}$$

- Intelligent grounding!

Safety

- A normal rule is safe, if each of its variables also occurs in some positive body literal
- A normal program is safe, if all of its rules are safe

Example

	Safe ?
$d(a)$	✔
$d(c)$	✔
$d(d)$	✔
$p(a, b)$	✔
$p(b, c)$	✔
$p(c, d)$	✔
$p(X, Z) \leftarrow p(X, Y), p(Y, Z)$	✔
$q(a)$	✔
$q(b)$	✔
$q(X) \leftarrow not\ r(X), d(X)$	✔
$r(X) \leftarrow not\ q(X), d(X)$	✔
$s(X) \leftarrow not\ r(X), p(X, Y), q(Y)$	✔

Programs with Integrity Constraints: Syntax

- Integrity constraints eliminate unwanted candidate solutions

Definition (Integrity Constraint)

An integrity constraint is a (special kind of) rule of the form

$$\leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n$$

where $n \geq m \geq 1$, and each A_i $(1 \leq i \leq n)$ is an atom.

Example

The integrity constraint

$$\leftarrow painted(X, C), painted(Y, C), adjacent(X, Y)$$

intuitively, would prevent the existence of answer sets in which two adjacent nodes (X and Y) are painted with the same colour (C).

Programs with Integrity Constraints: Semantics

Definition (Semantics of Integrity Constraints)

An integrity constraint of the form

$$\leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n$$

is mapped to the rule (where x is a new atom not appearing anywhere else in the program)

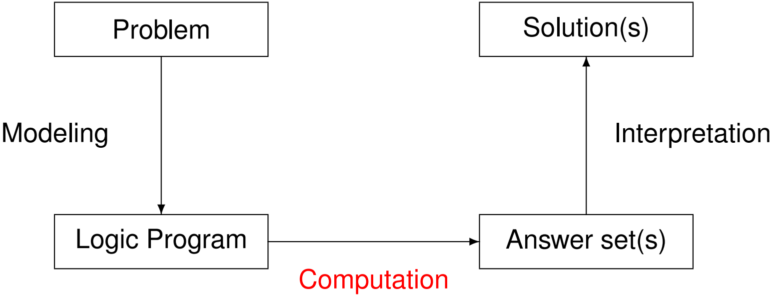
$$x \leftarrow A_1, \ldots, A_m, not\ A_{m+1}, \ldots, not\ A_n, not\ x$$

Example

Compare the answer sets of the following logic programs:

$$\begin{aligned}
\Pi \ &= \{\ p \leftarrow not\ q \qquad q \leftarrow not\ p\} \\
\Pi' \ &= \{\ p \leftarrow not\ q \qquad q \leftarrow not\ p \qquad \leftarrow p\} \\
\Pi'' &= \{\ p \leftarrow not\ q \qquad q \leftarrow not\ p \qquad \leftarrow not\ p\}
\end{aligned}$$

Problem solving in ASP: Computation



Standard Computation Scheme

- Global parameters: Logic program Π and its set of atoms $\mathcal{A}$.

Definition ($answerset_\Pi(T, F)$)

1. $(T, F) \leftarrow propagation_\Pi(T, F)$
2. **if** $(T \cap F) \neq \emptyset$ **then fail**
3. **if** $(T \cup F) = \mathcal{A}$ **then return**(X)
4. **select** $A \in \mathcal{A} \setminus (T \cup F)$
5. $answerset_\Pi(T \cup \{A\}, F)$
6. $answerset_\Pi(T, F \cup \{A\})$

Comments:

- (T, F) is supposed to be a 3-valued model such that $T \subseteq X$ and $F \cap X = \emptyset$ for an answer set X of Π.
- Key operations: $propagation_\Pi(T, F)$ and '**select** $A \in \mathcal{A} \setminus (T \cup F)$'
- Worst case complexity: $\mathcal{O}(2^{|\mathcal{A}|})$