An Introduction to Logic Programming by way of Answer Set Programming

Alexander Berenbeim

November 30, 2015

What Is The ASP paradigm?

- Perhaps the simplest motivation for Answer Set Programming are problems I dealing with search, diagnosis, information integration, routing & scheduling, knowledge management, etc., where the reasoning occurs with modeling constraints, or modeling preferences, or incomplete information, etc.
- This leads to the Answer Set Programming paradigm:
  1. **Encode** our problem I as a logic program P, such that the solutions of I will be models of P;
  2. **Compute** some models M of P using an AS solver, such as `dlv` or `Prolog`;
  3. **Extract** from M a solution for I.
- In effect, this switches the onus on the programmer from stating how to *solve* the problem I to how to *state* the problem I.
- ASP is rooted strongly logic programming, particularly in the fields of Knowledge Representation and Reasoning with formalisms aimed at belief sets, commonsense reasoning, defeasible reasoning, preferences and priorities.

The Formal Motivation for Answer Sets

- Consider the following metalogical statement :

$$(\star)\text{If } \phi \vdash \exists x, \varphi, \text{ then there is a term } t \text{ such that } \phi \vdash_{[t/x]} \varphi$$

- $(\star)$ says an existential proposition under an assumption ϕ, φ will have a **constructible solution** t.
- $(\star)$ is almost always not true. However it is true for **sets of universal Horn formulae**, which are of central importance in the field of *logic programming*.
- We may think of logic programs P as being built from simple constituent blocks that syntactically correspond to the language of predicate calculus, where **constants** correspond to *objects* and **variables** correspond to *subjects* related to one another by **predicates** through **atoms**, the sum total of which describe the *scenario* being modeled.

Logic Programming Primer: Horn Logic Programming

- A **positive logic program** P is a finite set of clauses called **rules** of the form
$$a \leftarrow b_1, \ldots, b_m$$
where a, b_i are atoms of a first-order language $\mathcal{L}$.
- By convention, we call a the **head of the rule** and $b_1, \ldots, b_m$ the **body of the rule**, while a rule with an empty body is called a **fact**. Rules without variables are **ground** while those with variables are **non-ground**.
- Rules do not strictly correspond to the procedural scheme of imperative languages, as a variable X in an imperative language associates a single valued to it, standing in for a named storage cell, while in a logic program, as a declarative construct, X reads as *any* X *having a certain property*.

Logic Programming Primer: Proof Calculi

Universal Horn formulae are derived using the following calculus:

1. (Rules)

$$\frac{}{(\neg\varphi_0 \vee \cdots \vee \neg\varphi_n \vee \varphi)} \; (n \in \mathbb{N}, \varphi_1, \ldots, \varphi_n, \varphi \text{ atomic})$$

   As in classical logic, $\varphi \leftarrow \varphi_0, \ldots, \varphi_n \equiv \varphi \vee \neg\varphi_0 \vee \neg\varphi_1 \vee \cdots \vee \neg\varphi_n$.

2. (Goals)

$$\frac{}{(\neg\varphi_0 \vee \cdots \vee \neg\varphi_n)} \; (n \in \mathbb{N}, \varphi_0, \ldots, \varphi_n \text{ atomic})$$

3. (Conjunction)

$$\frac{\varphi \qquad \psi}{(\varphi \wedge \psi)}$$

4. (Universal Extension)

$$\frac{\varphi}{\forall x, \varphi}$$

5. (**Selective Linear Definite (SLD) resolution**)

$$\frac{\leftarrow \varphi_0, \ldots, \varphi_i, \ldots, \varphi_m \qquad \varphi \leftarrow \psi_0, \ldots, \psi_n}{\leftarrow \varphi_0, \ldots, \psi_0, \ldots, \psi_m, \ldots, \varphi_n} \; (\varphi \text{ unifies with } \varphi_i)$$

Logic Programming Primer: Model Semantics

Let P be a logic program.

Definition

A **Herbrand universe of P**, denoted by $\text{HU}(P)$, consists of the set of all terms formed by the language $\mathcal{L}_P$.

A **Herbrand base of P**, denoted by $\text{HB}(P)$, consists of all ground atoms formed from predicates in P and terms in $\text{HU}(P)$, such that an **interpretation** over $\text{HU}(P)$ is simply a subset $I \subseteq \text{HB}(P)$ may be understood a set of of grounds atoms true in a given *scenario*. An interpretation M may be a **model** of

1. a ground clause $C \equiv a \leftarrow b_1, \ldots, b_n$ if $\{b_1, \ldots, b_n\} \not\subseteq M$ or $a \in M$;
2. a clause C if $M \models C'$ for all $C' \in grnd(C)$, the set of all ground instances of C appearing in $\text{HU}(P)$;
3. a program P if $M \models C$ for all clauses $C \in P$.

Logic Programming Primer: Minimal Models

Consider the following program P

$$a \leftarrow b. \qquad b \leftarrow a. \qquad c.$$

The only model that is necessarily true for P is $M = \{c\}$. Of course, it may be the case that $M' = \{a, b, c\}$. If there is no model N of P such that $N \subsetneq M$, then M is **minimal**

Can you think of a program P' where M' is minimal?

$$a \leftarrow b. \qquad b \leftarrow c. \qquad c.$$

Logic Programming Primer: Minimal Model Computation

If P is a positive logic program, then there is a single minimal model denoted $\mathrm{LM}(P)$. We iteratively compute $\mathrm{LM}(P)$ by the **immediate consequence operator**, where $T_P : 2^{\mathrm{HB}(P)} \to 2^{\mathrm{HB}(P)}$ is defined by

$$I \mapsto \{a \mid \exists (a \leftarrow b_1, \ldots, b_n) \in grnd(P), \{b_1, \ldots, b_m\} \subseteq I\}$$

i.e. under T_P, for all founded atoms in the body of a rule r, then a will be founded. Consider P' from the previous slide.

$$T^0_{P'} = \{\}. \qquad T^1_{P'} = \{c\}. \qquad T^2_{P'} = \{c, b\}.$$

$$T^3_{P'} = \{c, b, a\}. \qquad T^n_{P'} = T^3_{P'}, n \geq 3.$$

Negation in Logic Programs

- We extend positive logic programs to **normal logic programs** by adding a notion of negation different from negation in classical logic, pragmatically interpreted as **Negation as failure** with falsity denoted by *fail*, and where one considers $\texttt{not}\, a(\cdot)$ to be true if no corresponding positive literal $a(\cdot)$ can be finitely proved through SLD resolution.
- For example, consider the following program P :

$$man(dilbert)$$

$$single(X) \leftarrow student(X), \texttt{not}\, husband$$

$$husband(X) \leftarrow fail$$

- The Prolog query $? - single(X)$ will return $X = dilbert$, since $husband(dilbert)$ cannot be proved for P.

Negation in Logic Programs: Dependency Graphs

- Now instead of P, consider the program Q

$$man(dilbert)$$

$$single(X) \leftarrow student(X), \texttt{not}\, husband(X)$$

$$husband(X) \leftarrow man(X), \texttt{not}\, single(X)$$

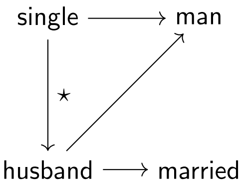
- SLD resolution algorithms will loop forever, though we get around this by introducing and examining the order of evaluation of rules. A **dependency graph** of Q, $dep(Q) = (V_Q, E_Q)$ has its set of nodes V_Q correspond to the set of all predicates p, q in Q, and the pair (p, q) is in E_Q iff there is a rule r such that for the pairs of vertices p, q, p is the head of the rule r and q is in the body of a rule r. If the literals are positive, then by convention this is rather confusingly denoted by $p \rightarrow q$. If a literal is under negation, convention dictates we denote this by $\star(p, q)$, or $(p \rightarrow^{\star} q)$.

Negation in Logic Programs: Stratification

- Dependency graphs allow us to check whether a program can be stratified.
- A **stratification** of a program P is a partitioning $\Sigma = \{S_i \mid i \in [n]\}$ of $pred(P)$, the set of predicate names occurring in a program P such that
  1. if $p \in S_i, q \in S_j$, and $p \rightarrow q$ are in $dep(P)$, then $i \geq j$;
  2. $if p \in S_i, q \in S_j$ and $p \rightarrow^\star q$ is in $dep(P)$, then $i > j$
- A stratification Σ of length $k \geq 1$ specifies an evaluation order for the predicates in a logic program P; this can be computed by a series of **iterative least models**, denoted $M_{P,\Sigma}$:
- With P_{S_i} denoting the subset of rules of$P$ whose head belongs to S_i, and $\mathrm{HB}(P_{S_i})^\star = \bigcup_{j \in [i]} \{p(t) \in \mathrm{HB}(P) \mid p \in S_j\}$, the iterative least model $M_i \subseteq \mathrm{HB}(P)$ with $i \in [k]$ is defined as
  1. M_1 least model of P_{S_1};
  2. For $i > 1$, M_i is the least subset of $\mathrm{HB}(P)$ such that $M_i \models P_{S_i}$ and $M_i \cap \mathrm{HB}(P_{S_{i-1}})^\star = M_{i-1} \cap \mathrm{HB}(P_{S_{i-1}})^\star$.

Negation in Logic Programs: Example

Recalling program Q, we have the following dependency graph



which stratifies as

$$S_0 = \{\}$$

$$S_1 = \{man, married\}$$

$$S_2 = \{husband\}$$

$$S_3 = \{single\}$$

Negation in Logic Programs: Unstratified Negation

- This can break down though, as not all models can be stratified. In fact, P' from earlier is not stratified, as more than one predicates are mutually defined over `not`, so that there are two mutually exclusive minimal models,

$$M = \{man(dilbert), single(dilbert)\}$$

$$N = \{man(dilbert), husband(dilbert)\}$$

  that is, we have *two different answer sets* to the query
- When faced with multiple plausible models, we are faced with the problem of specifying a *preferred model*, denoted $PM(P)$.
- The most commonly investigated notion of preferred model are **stable models**, which are not self-contradicting. Formally, a stable interpretation M of P is an *assumption we make*, with $P^M \subseteq P$ such that
  1. rules with `not` a are removed in the body for each $a \in M$;
  2. literals `not` a are removed from all other rules.
- In other words, an interpretation of M is a stable model of P if $M = LM(P^M)$

NLP: Reasoning From Stable Models

- Now that we've introduced negation, SLD resolution is no longer a sufficient inference rule. We rectify this situation by introducing two different inference rules:
    1. (**Brave Reasoning**) If $M \models a$ for a stable model M, then an atom a is **brave** a brave consequence of P, denoted $P \models_b a$
    2. (**Cautious Reasoning**) If $M \models a$ for every stable model of P, then a is a **cautious** consequence of P, denoted $P \models_c a$.

  Both $\models_b, \models_c$ are non-monotonic, as introducing further rules to P may invalidate the conclusions.

Normal Logic Programs: Computationally Understood

- Deciding whether a given program P has a stable model is $NP - complete$.
- This amounts to guessing a stable candidate M, checking in polynomial time if M is stable by verifying that the set of unfounded atoms in M is empty, where an unfounded atom a is the head of some rule r such that either an atom b appears as a positive literal in the body of r which is such that either $b \notin M$ or b is also unfounded, or b appears as a negative literal in the body of r such that $b \in M$.
- Introducing functions can make this undecidable, as we may have models of infinite size. Consider the program F:

$$p(a)$$

$$p(f(X)) \leftarrow p(X)$$

$grnd(F) = \{p(a), p(f(a)) \leftarrow p(a), p(f(f(a))) \leftarrow p(f(a)), \ldots\}$ is infinite, and is the unique stable model. For non-ground programs with function symbols, this problem becomes as difficult as the Halting program.

Further Extending Logic Programs

- We can extend our logic programs further by considering disjunctive rule heads or **strong (classical first order negation)** by considering P with rules of the form:

$$a_1 \vee a_2 \vee \cdots \vee a_k \leftarrow b_1, \ldots, b_m, \mathtt{not}\, c_1, \ldots, \mathtt{not}\, c_n$$

  where $k, m, n \in \mathbb{N}$ and a_i, b_j, c_l are atoms (or strongly negated atoms, denoted $-a_i, -b_j, -c_l$), and stable models are the minimal models M of a reduct P^M, so that disjunctive heads may as well be read as XOR.
- Strong negation is different from provably *knowing* a is false; `not` a means a cannot be derived from a given body of rules, while $-a$ assumes that a is false by default.
- We can compile strong negation away by doing the following:
  1. view $-p(X)$ as an atom with a fresh predicate symbol;
  2. add the clause $NC : \mathit{falsity} \leftarrow \mathtt{not}\, \mathit{falsity}, p(X), -p(X)$ to P, i.e extend P to P' and reduce from $EL(D)P$ to $(D)NLP$;
  3. select the stable models of P'.

  The stable models of P' will still be *answers sets to P*.

One Last Example of ASP

We can consider the ASP approach to the problem of computing legal 3-colorings of a graph $G = (V, E)$. We store the facts of our graph as $node(n)$ for each $n \in V$ and $edge(n, m)$ for each $(n, m) \in E$. The general specification for solutions is then

$$red(X) \leftarrow node(X), \mathtt{not}\, green(X), \mathtt{not}\, blue(X)$$

$$green(X) \leftarrow node(X), \mathtt{not}\, blue(X), \mathtt{not}\, red(X)$$

$$blue(X) \leftarrow node(X), \mathtt{not}\, red(X), \mathtt{not}\, green(X)$$

with a single disjunctive rule

$$blue(X) \vee red(X) \vee green(x) \leftarrow node(X)$$

The Answer Sets will correspond to all legal 3-colorings of G.

Questions?