ASP and Negation(s)

Víctor Gutiérrez Basulto

List of Tasks

1 Classical and Default Negation

Read the following excerpt from the textbook:

2.3.4 TWO (AND A HALF) KINDS OF NEGATION

The addition of a second kind of negation, resembling classical negation, is mainly motivated by a desire to ease knowledge representation. Pragmatically, the introduction of negation amounts to the addition of new language symbols for all atoms, along with the addition of rules fixing the relation of these new atoms to their original counterparts.

Given that an atom a is satisfied by a stable model whenever $a \in X$, the difference between a classically negated literal $\neg a$ and a default negated one, $\sim a$, intuitively boils down to the difference between $\neg a \in X$ and $a \notin X$, respectively. A popular example illustrating this distinction is given by the two programs

$$P = \{\mathit{cross} \leftarrow \sim \mathit{train}\} \qquad \text{and} \qquad P' = \{\mathit{cross} \leftarrow \neg \mathit{train}\}.$$

While informally the first program suggests crossing the tracks whenever we *do not know* whether a train approaches, the second one advises doing so whenever we *know* that no train arrives. Accordingly, P has the stable model $\{\mathit{cross}\}$, whereas P' yields the empty stable model. We must add the fact $\neg \mathit{train} \leftarrow$ to P' to obtain the same conclusion.

To make things precise, we extend our set of atoms $\mathcal{A}$ by $\overline{\mathcal{A}} = \{\neg a \mid a \in \mathcal{A}\}$ such that $\mathcal{A} \cap \overline{\mathcal{A}} = \emptyset$. That is, $\neg a$ is the classical negation of a and vice versa. The semantics of classical negation is

enforced by the addition of the following set of rules[4]

$$P^{\neg} = \{a \leftarrow b, \neg b \mid a \in (\mathcal{A} \cup \overline{\mathcal{A}}), b \in \mathcal{A}\}.$$

For illustration, let us extend once more Program P_3 from Page 15.

$$P_3 \cup \{c \leftarrow b,\ \neg c \leftarrow b\} \ = \ \{a \leftarrow {\sim}b,\ b \leftarrow {\sim}a\} \cup \{c \leftarrow b,\ \neg c \leftarrow b\}.$$

The resulting program, viz. $P_3 \cup \{c \leftarrow b,\ \neg c \leftarrow b\} \cup P^{\neg}$, has a single stable model $\{a\}$. Note that the second stable model $\{b\}$ of P_3 is eliminated by the contradiction, c and $\neg c$, obtained from b.

Strictly speaking, the stable models obtained from programs with classical negation are no models because a stable model of the transformed program may contain an atom and its negation. For instance, the program $P_3 \cup \{c \leftarrow,\ \neg c \leftarrow\} \cup P^{\neg}$ yields the stable model $\mathcal{A} \cup \overline{\mathcal{A}}$. In fact, a transformed program has either only stable models free of complementary literals or the single stable model $\mathcal{A} \cup \overline{\mathcal{A}}$.

Finally, the appearance of default negation in rule heads can be reduced to normal programs. To this end, we must also extend our set $\mathcal{A}$ of atoms by $\widetilde{\mathcal{A}} = \{\widetilde{a} \mid a \in \mathcal{A}\}$ such that $\mathcal{A} \cap \widetilde{\mathcal{A}} = \emptyset$. The stable models of a program P with default negation in the head are then provided by the stable models (projected to $\mathcal{A}$) of the following normal program.

$$\begin{aligned} \widetilde{P} \ = \ & \{r \in P \mid \mathit{head}(r) \neq {\sim}a\} \\ & \cup \{\leftarrow \mathit{body}(r) \cup \{{\sim}\widetilde{a}\} \mid r \in P, \mathit{head}(r) = {\sim}a\} \\ & \cup \{\widetilde{a} \leftarrow {\sim}a \mid r \in P, \mathit{head}(r) = {\sim}a\}. \end{aligned} \tag{2.13}$$

This translation also works for disjunctive logic programs, as illustrated below.

[4] Existing ASP systems implement a different semantics by setting $P^{\neg}$ to $\{\leftarrow b, \neg b \mid b \in \mathcal{A}\}$. This eliminates the putative "stable model" $\mathcal{A} \cup \overline{\mathcal{A}}$.

Write a program, `train.dl` with the following rule:
`if ¬ train, then cross`
Execute `clingo` and check the resulting answer sets.

Modify the program to consider the following rule:
`if ∼ train, then cross`
This rule should be read as: `if there is no evidence of train, then cross` or, equivalently, `usually you can cross, unless there is evidence of train`.
Execute `clingo` and check the resulting answer sets, against the previous case.

NB Lines can be commented using % symbol.

2 Non-monotonic Reasoning

Write a program, `tweety.dl` with the following information:

- `usually if bird(X) then fly(X), unless abnormal(X)`
- `if X is a penguin, X is a bird`
- `if X is a penguin, X is abnormal`
- `tweety is a bird`

Once you execute with `clingo`, you should see that it is reasonable to conclude that `tweety` flies.
Let us now add the piece of information that `tux` is a penguin. Now it is reasonable to conclude that `tux` does not fly.
Write a program, `tweety1.dl` with the following information:

- `if bird(X) then fly(X), unless is known that X doesn't fly.`
- `if bird (X) then X doesn't fly, unless X is know to fly.`
- `if X is a penguin, then is definitely known that X doesn't fly.`
- `tweety is a bird`
- `tux is a penguin`

Once you execute with `clingo`, you should see that it is given the above we can conclude that `tweety` flies or not, and tux does not fly. That is you will get two stable models in one tweety flies in the other it doesn't.

IMPORTANT: you can use `-` (symbol for classical negation) in the head, i.e. you can write rules of the form `-p(X):- body`.

Run `clingo -n 0 <program.dl>` to display all the stable models of your program.

3 Wedding Dinner Seating

Read the following excerpt from the textbook:

2.3.2 CORE LANGUAGE

This section introduces the essential modeling language of ASP, sufficient for expressing all search problems in *NP*. To this end, we mainly focus on logic programs consisting of

- normal rules,
- choice rules,
- cardinality rules, and
- weight rules.

Together with optimization statements introduced in the following section, this collection constitutes the basic language constructs accepted by ASP solvers like *smodels* and *clasp* (see also Section 7.1.4). Except for optimization statements, all language extensions are complexity-preserving. Hence, we generally fix their meaning via translations reducing them to normal logic programs rather than providing genuine semantics.

A compact formal account on the core language of ASP (extended by disjunction) is given in Appendix A.

Integrity constraints An integrity constraint is of the form

$$\leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n \tag{2.2}$$

where $0 \leq m \leq n$ and each a_i is an atom for $1 \leq i \leq n$.

An integrity constraint rules out stable models satisfying its body literals. Their purpose is to eliminate unwanted solution candidates. No atoms are derivable through integrity constraints. For instance, the integrity constraint

```
:- edge(3,7), color(3,red), color(7,red).
```

can be used in a graph coloring problem to express that vertices `3` and `7` must not both be colored red if they are connected.

An integrity constraint can be translated into a normal rule. To this end, the constraint in (2.2) is mapped onto the rule

$$x \leftarrow a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n, \sim x$$

where x is a new symbol, that is, $x \notin \mathcal{A}$.

To illustrate this, let us extend P_3 from Page 15 by integrity constraints as follows.

$$\begin{array}{lcl} P_3 \cup \{\leftarrow a\} & = & \{a \leftarrow \sim b,\ b \leftarrow \sim a\} \cup \{\leftarrow a\} \\ P_3 \cup \{\leftarrow \sim a\} & = & \{a \leftarrow \sim b,\ b \leftarrow \sim a\} \cup \{\leftarrow \sim a\} \end{array}$$

From the two stable models of P_3, the first program only admits $\{b\}$, while the second one yields $\{a\}$. The same stable models are obtained from Program $P_3 \cup \{x \leftarrow a, \sim x\}$ and $P_3 \cup \{x \leftarrow \sim a, \sim x\}$, respectively.

In general, the addition of integrity constraints to a logic program can neither produce new stable models nor alter existing ones; rather it can only lead to their elimination.

Choice rules A choice rule is of the form[2]

$$\{a_1, \ldots, a_m\} \leftarrow a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o \tag{2.3}$$

where $0 \leq m \leq n \leq o$ and each a_i is an atom for $1 \leq i \leq o$.

The idea of a choice rule is to express choices over subsets of atoms. Any subset of its head atoms can be included in a stable model, provided the body literals are satisfied. Thus, for instance, the program $P = \{\ a \leftarrow,\ \{b\} \leftarrow a\}$ has two stable models, $\{a\}$ and $\{a, b\}$. For another example, at a grocery store you may or may not buy pizza, wine, or corn.

```
{ buy(pizza), buy(wine), buy(corn) } :- at(grocery).
```

A choice rule of form (2.3) can be translated into $2m + 1$ rules

$$\begin{array}{lcl} a' & \leftarrow & a_{m+1}, \ldots, a_n, \sim a_{n+1}, \ldots, \sim a_o \\ a_1 & \leftarrow & a', \sim \overline{a_1} \quad \ldots \quad a_m \leftarrow a', \sim \overline{a_m} \\ \overline{a_1} & \leftarrow & \sim a_1 \quad \ldots \quad \overline{a_m} \leftarrow \sim a_m \end{array}$$

by introducing new atoms $a', \overline{a_1}, \ldots, \overline{a_m}$. Applying this transformation to the choice rule $\{b\} \leftarrow a$ in our example program P yields

$$\begin{array}{lcl} a & \leftarrow & \\ & & \begin{array}{lcl} b' & \leftarrow & a \\ b & \leftarrow & b', \sim \overline{b} \\ \overline{b} & \leftarrow & \sim b \end{array} \end{array}$$

This program has two stable models, $\{a, b', \overline{b}\}$ and $\{a, b', b\}$, whose intersections with the atoms in the original program correspond to the stable models indicated above.

Cardinality rules A cardinality rule is of the form

$$a_0 \leftarrow l\ \{\ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\ \} \tag{2.4}$$

where $0 \leq m \leq n$ and each a_i is an atom for $0 \leq i \leq n$; l is a non-negative integer.

Cardinality rules allow for controlling the cardinality of subsets of atoms via the lower bound l. That is, the head atom belongs to a stable model, if the latter satisfies at least l body literals. For example, Program $P = \{\ a \leftarrow,\ c \leftarrow 1\ \{a, b\}\ \}$ has the stable model $\{a, c\}$. Here is a less artificial example of a cardinality rule, describing that one passes Course 42, provided one passes two out of three assignments.

[2]The inclusion of default negated literals among the head literals of plain choice rules is without effect. No matter whether a literal $\sim a$ is chosen or not, it cannot contribute to a stable model.

```
pass(c42) :- 2 { pass(a1),  pass(a2),  pass(a3) }.
```

Next, we generalize cardinality rules for expressing more general forms of cardinality constraints, and we provide semantics via translations into simpler forms of programs.

At first, we consider cardinality rules with upper bounds.

$$a_0 \leftarrow l\ \{\ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\ \}\ u \tag{2.5}$$

Such rules extend the syntax of cardinality rules in (2.4) by adding another non-negative integer, u, serving as an upper bound on the cardinality of the satisfied body literals. The single constraint in the body of (2.5) is commonly referred to as a *cardinality constraint*.

A cardinality rule with an upper bound can be expressed by the following three rules (introducing new symbols b and c).

$$\begin{array}{rcl} a_0 & \leftarrow & b, \sim c \\ b & \leftarrow & l\ \{\ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\ \} \\ c & \leftarrow & u{+}1\ \{\ a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\ \} \end{array} \tag{2.6}$$

So far, all cardinality constraints occurred in rule bodies only. We next consider rules with cardinality constraints as heads. Such a rule is of the form

$$l\ \{a_1, \ldots, a_m, \sim a_{m+1}, \ldots, \sim a_n\}\ u \leftarrow a_{n+1}, \ldots, a_o, \sim a_{o+1}, \ldots, \sim a_p \tag{2.7}$$

where $0 \leq m \leq n \leq o \leq p$ and each a_i is an atom for $1 \leq i \leq p$; l and u are non-negative integers. For example, we can express that vertex 42 must be colored with exactly one color, among red, green, and blue, as follows.

```
1 { color(v42,red), color(v42,green), color(v42,blue) } 1 :- vertex(v42).
```

Conditional literals A conditional literal is of the form $\ell : \ell_1 : \cdots : \ell_n$ for $0 \leq i \leq n$. The purpose of this simple yet powerful language construct is to govern the instantiation of the "head literal" ℓ through the literals $\ell_1, \ldots, \ell_n$. In this respect, a conditional literal $\ell : \ell_1 : \cdots : \ell_n$ can be regarded as the list of elements in the set $\{\ell \mid \ell_1, \ldots, \ell_n\}$.

For example, given three facts `color(red)`, `color(green)`, and `color(blue)`, the conditional literal in the cardinality constraint

```
1 { color(v42,C) : color(C) } 1 :- vertex(v42).
```

expands to the cardinality constraint

```
1 { color(v42,red), color(v42,green), color(v42,blue) } 1 :- vertex(v42).
```

However, the final form of the expanded conditional literal is context-dependent. For instance, given the above facts, the integrity constraint

```
:- color(v42,C) : color(C).
```

results in

```
:- color(v42,red), color(v42,green), color(v42,blue).
```

Similarly, conditional literals can be used in optimization statements (see Section 2.3.3) and disjunctive rule heads (see Section 2.3.5). A sophisticated use of conditional literals in various contexts is shown in Listing 4.7 on Page 64 as well as Listing 8.10 on Page 163.

Marina, Willem, Bob, Tina, Bert, Jane, and Alyssa are invited at a wedding. Write a program to identify the possible allocations for them knowing that:

- there are three tables;
- each table should have at least two people;
- each table cannot have more than three people;
- Marina does not want to seat with Willem.