Answer Set Programming

This material is based on a 2012 lecture by Gerhard Lakemeyer, in turn derived from a 2009 paper by Ilkka Niemela.

Negation as failure

From a procedural point of view, negation as failure is an important component used extensively in logic programming (e.g. Prolog).

But from a *declarative* point of view, what does it all mean?

- What are the entailments of a KB that includes negation as failure?
- What does $I \models \mathrm{KB}$ mean when the KB includes negation as failure?

In this section, we examine one very popular way to understand negation as failure, in terms of what is called *stable models*.

This has lead to a new form of logic programming called *answer set programming* that uses negation as failure more flexibly than in Prolog.

(References)

Minimal models

Suppose KB is a set of (propositional) positive Horn clauses.

Example KB1:

$$\begin{array}{lll} p & \Leftarrow & \\ q & \Leftarrow & p \\ r & \Leftarrow & p \wedge q \\ t & \Leftarrow & r \wedge s \\ s & \Leftarrow & s \end{array}$$

Let us denote an interpretation I by the set of atoms satisfied by I.
E.g., let $I = \{p, q, r, t, u\}$. Then $I \models \mathrm{KB1}$.

An interpretation I is called a *minimal model* of a KB iff $I \models \mathrm{KB}$ and for all I' such that $I' \subsetneq I$, $I' \not\models \mathrm{KB}$.

(Intuitively: A minimal model is one that is as false as possible.)

Theorem: For any set of positive clauses KB, KB has a unique minimal model I. Furthermore, $p \in I$ iff $\mathrm{KB} \models p$.

Adding negation as failure

Imagine an extended KB consisting of "rules" of the following form:

$$p \Leftarrow p_1 \wedge \ldots \wedge p_n \wedge \textbf{not}(q_1) \wedge \ldots \wedge \textbf{not}(q_m). \qquad (n \geq 0, m \geq 0)$$

Intuitively: infer p if every p_i can be inferred and none of the q_j can.

Example KB2:

$$\begin{array}{lcl} p & \Leftarrow & p \\ q & \Leftarrow & \textbf{not}(p) \end{array}$$

Can start by trying to define interpretations for KBs like this by treating the **not** as ordinary logical negation.

However, this would not allow us to conclude q, since this $\text{KB2} \not\models q$.

Two minimal models: $I = \{q\}$ and $I' = \{p\}$.

- The first minimal model is fine and leads to the conclusion we want.
- The second one is strange: it has p true, but for no good reason!

Eliminating ungrounded interpretations

Consider an interpretation I and a KB consisting of rules.

Define: The *reduct* of KB wrt I is the set of (positive) clauses defined by:

1. remove any rule that contains a **not**(q) where $I \models q$;
2. delete all **not**(q) terms in the remaining rules.

Define: I is called a *stable model* of a KB iff I is the minimal model of the reduct of KB wrt I.

Return to example KB2:

$$\begin{array}{lcl} p & \Leftarrow & p \\ q & \Leftarrow & \textbf{not}(p) \end{array}$$

Then $I = \{q\}$ is stable, but $I' = \{p\}$ is not, as desired.

The reduct wrt I is $\{[p, \neg p], [q]\}$ whose minimal model is $\{q\}$.

The reduct wrt I' is $\{[p, \neg p]\}$ whose minimal model is $\{\}$.

More examples

if I = {q) then reduct is {[q],[p V ~q]} and MM is {q,p} not stable
if I = {q,p} then reduct is {[q]} and MM {q} not stable

Example KB3:

$$\begin{array}{lll} p & \Leftarrow & \textbf{not}(p) \wedge q \\ q & \Leftarrow & \end{array}$$

KB3 has no stable model. (Go through all four candidates.)

Example KB4:

$$\begin{array}{lll} p & \Leftarrow & r \wedge \textbf{not}(q) \\ q & \Leftarrow & \textbf{not}(p) \\ r & \Leftarrow & \textbf{not}(s) \\ s & \Leftarrow & \textbf{not}(p) \end{array}$$

KB4 has two stable models $\{r, p\}$ and $\{s, q\}$.

reduct for {r,p} is {[p V ~r], [r]} so MM {p,r} stable!

The interpretation $\{p, s\}$ is not stable, for example, since its reduct is $\{[p, \neg r]\}$ whose minimal model is $\{\}$.

Constraints

Constraints are rules whose heads are empty:

$$\Leftarrow p_1 \wedge \ldots \wedge p_n \wedge \textbf{not}(q_1) \wedge \ldots \wedge \textbf{not}(q_m).$$

We say that such a constraint is satisfied by an interpretation I iff it is not the case that $\{p_1, \ldots p_n\} \subseteq I$ and $\{q_1, \ldots q_m\} \cap I = \{\}$.

(Intuitively: the body of the rule should come out false.)

Example KB5:

$$\begin{array}{lcl} p & \Leftarrow & r \wedge \textbf{not}(q) \\ q & \Leftarrow & \textbf{not}(p) \\ r & \Leftarrow & \textbf{not}(s) \\ s & \Leftarrow & \textbf{not}(p) \\ & \Leftarrow & s \wedge \textbf{not}(p) \\ & \Leftarrow & r \wedge s \wedge \textbf{not}(q) \end{array}$$

Now $\{r, p\}$ is the only stable model.

More on constraints

A constraint

$$\Leftarrow p_1 \wedge \ldots \wedge p_n \wedge \textbf{not}(q_1) \wedge \ldots \wedge \textbf{not}(q_m).$$

can be represented by ordinary rules: introduce two new atoms bad and bad' and replace the above constraint by

$$\begin{array}{lcl} bad' & \Leftarrow & \textbf{not}(bad') \wedge bad \\ bad & \Leftarrow & p_1 \wedge \ldots \wedge p_n \wedge \textbf{not}(q_1) \wedge \ldots \wedge \textbf{not}(q_m)\ . \end{array}$$

(See KB3.)

Theorem: Let KB be a set of extended clauses and C a set of constraints. Then the stable models of $\mathrm{KB} \cup C$ (as above) are exactly those stable models of KB which satisfy all the constraints in C.

Variables and constants

First-order rules and constraints with variables and constants can be reduced to the propositional case by appealing to the finite Herbrand base.

Example KB6:

$$
\begin{array}{lcl}
F(a) & \Leftarrow & \\
F(b) & \Leftarrow & \\
H(x,y,z) & \Leftarrow & F(x) \wedge G(y) \wedge G(z) \wedge \textbf{not}(F(x)) \\
H(x,x,x) & \Leftarrow & F(x)
\end{array}
$$

Example KB6′ (Herbrand base):

$$
\begin{array}{lcl}
F(a) & \Leftarrow & \\
F(b) & \Leftarrow & \\
H(a,a,a) & \Leftarrow & F(a) \wedge G(a) \wedge G(a) \wedge \textbf{not}(F(a)) \\
H(a,a,b) & \Leftarrow & F(a) \wedge G(a) \wedge G(b) \wedge \textbf{not}(F(a)) \\
\ldots & & \\
H(b,b,b) & \Leftarrow & F(b)
\end{array}
$$

Unique stable model $\{F(a), F(b), H(a,a,a), H(b,b,b)\}$

Answer Set Programming

Typically in ASP, the reasoning is not concerned with finding what is true in all stable models, but on calculating the stable models themselves.

Example: the Graph k-Colourability Problem

$$
\begin{array}{lcl}
Vertex(v) & \Leftarrow & \text{(a rule for each vertex)} \\
Edge(v, v') & \Leftarrow & \text{(a rule for each edge)} \\
Col(c) & \Leftarrow & \text{(a rule for each of the } k \text{ colours)}
\end{array}
$$

$$
\begin{array}{lcl}
HasCol(x, y) & \Leftarrow & Vertex(x) \wedge Col(y) \wedge \textbf{not}(OtherColour(x,\ y)) \\
OtherColour(x, y) & \Leftarrow & Vertex(x) \wedge Col(y) \wedge Col(z) \wedge\ y \neq z \wedge HasCol(x, z) \\
 & \Leftarrow & Edge(x, y) \wedge Col(z) \wedge HasCol(x, z) \wedge HasCol(y, z)
\end{array}
$$

- The atomic rules define the graph and the allowed colours.
- The next two rules generate candidate solutions: one colour per vertex.
- The final constraint ensures adjacent vertices have different colours.

Each stable model specifies a legal colouring of the vertices: $HasCol(v, c)$.

Extensions and state of the art

Stable models are not restricted to Horn clauses with negation as failure.

Can allow disjunctions in the head:

$$(p \vee q) \quad \Leftarrow$$

has two stable models $\{p\}$ and $\{q\}$.

Can also have real logical negation (over and above negation as failure), but need to modify the definition of minimal model.

Today very efficient ASP solvers exist such as `smodels` (Helsinki), `dlv` (Vienna), or `clingo` (Potsdam) used in a number of application areas involving search and constraints.

Pro: much faster than Prolog, with more flexible uses

Con: restricted to finite Herbrand base

The input file format for `clingo`

The syntax is similar to Prolog:

```
% Three vertices, two edges, two colours
vertex(a).   vertex(b).   vertex(c).
edge(a,b).   edge(a,c).
col(blue).   col(red).

% The rules of graph colouring
hascol(X,Y) :- vertex(X), col(Y), not othercol(X,Y).
othercol(X,Y) :- vertex(X), col(Y), col(Z), Y != Z, hascol(X,Z).
:- edge(X,Y), col(Z), hascol(X,Z), hascol(Y,Z).

% Only display the hascol predicate in the stable models
#hide.
#show hascol/2.
```

Running `clingo`

To run the file `color.cl` from the previous slide:

```
> clingo 0 color.cl              % enumerate all stable models
Answer: 1
hascol(a,red) hascol(c,blue) hascol(b,blue)
Answer: 2
hascol(c,red) hascol(b,red) hascol(a,blue)
SATISFIABLE

Models       : 2
Time         : 0.000
  Prepare    : 0.000
  Prepro.    : 0.000
  Solving    : 0.000
```

Clingo can be obtained from `http://potassco.sourceforge.net/`