Negation

Datalog with Negation

Okay. Let us add "**not**" to the Datalog language (Datalog¬).

E.g.,

$$\begin{aligned} \mathit{cousin}\,(X,\ Y) \leftarrow\ & \mathit{grandparent}\,(P,\ X), \\ & \mathit{grandparent}\,(P,\ Y), \\ & X \neq Y, \\ & \textsf{not}\ \mathit{sibling}\,(X,\ Y). \end{aligned}$$

We only allow use of "**not**" on the right-hand side of the '←'.

The intuitive meaning of "**not**" is quite clear.

How to handle it formally is far from clear.

- What are the models of a Datalog¬ database?
- What should the proof procedure be for Datalog¬?

This "**not**" is *not* logical negation ('¬')!

Safeness
Extended for Datalog¬

We require that Datalog¬ programs be *safe*.

We need to extend the definition of *safeness* for Datalog¬:

Any variable that appears either in the *head* atom of the rule (on the left-hand side) *or* in a negated atom must also appear in a non-negated atom in the *body* (on the right-hand side). Thus,

$$\begin{aligned} h\,(X_1,\ \ldots,\ X_k)\ \leftarrow\ & b_1\,(Y_1,\ \ldots,\ Y_{j_1}), \ldots, b_m\,(Y_{j_{m-1}},\ \ldots,\ Y_{j_m}), \\ & \mathsf{not}\ d_1\,(Z_1,\ \ldots,\ Z_{j_1}), \ldots, \mathsf{not}\ d_n\,(Z_{j_{n-1}},\ \ldots,\ Z_{j_n}). \end{aligned}$$

is safe if

$$(\{X_1, \ldots, X_k\} \cup \{Z_1, \ldots, Z_{j_n}\}) \subseteq \{Y_1, \ldots, Y_{j_m}\}$$

Non-Monotonicity

Non-classical Logic!

Adding a new fact many require that we retract other things that we used to know.

$$\begin{array}{ll} \mathcal{P}\text{:} & a \leftarrow b, \mathsf{not}\ c. \\ & b. \end{array}$$

From $\mathcal{P}$, a follows.

$$\begin{array}{ll} \mathcal{P}'\text{:} & a \leftarrow b, \mathsf{not}\ c. \\ & b. \\ & c. \end{array}$$

However, from $\mathcal{P}'$, a does not follow. In fact, we want to say that $\neg a$ follows.

Classical logic is monotonic. Thus this is a change from classical logic.

This also means that what we have in mind for "not" really *is* different from classical negation ('$\neg$').

Stratification

No cycles through "not"

The Grandmother database is *statically stratified*, even with the predicate *cousin*.

A program is *statically stratified* iff the predicates can be ordered such that no predicate employs another predicate negated that appears before it in the ordered list.

$$\textit{integer}\ (\textit{0}).$$
$$\textit{integer}\ (I) \leftarrow \textit{integer}\ (J),\ I \text{ is } J + 1.$$

$$\textit{even}\ (\textit{0}).$$
$$\textit{even}\ (I) \leftarrow \textit{integer}\ (I),\ I > 0,\ J \text{ is } I - 1,\ \textsf{not}\ \textit{even}\ (J).$$

$$\textit{odd}\ (I) \leftarrow \textit{integer}\ (I),\ \textsf{not}\ \textit{even}\ (I).$$

This odd-even program is clearly not statically stratified. However, it is *locally stratified*.

A program is *locally stratified* iff for any ground atom A (e.g., *even* (7)), it is not possible for the negation of atom A (e.g, not *even* (7)) to appear in a resolution path from A.

In other words, no "proof" of A relies on not A.

The Perfect Model

For Stratified Datalog¬ Programs

Just as there is one minimum model for a Datalog program, there exists one special model named the *perfect model* for each Datalog¬ program.

Let $\mathbf{P}$ denote the perfect model of program $\mathcal{P}$. The interpretation in which A is assigned *true* when $A \in \mathbf{P}$ and is assigned *false* when $A \notin \mathbf{P}$ is a model of $\mathcal{P}$ (in which the **not**'s are treated as logical ¬'s), and is, in a sense, minimal.

Negation-as-finite-failure (NAFF) remains a *sound* proof strategy for stratified datalog¬ programs.

Non-mon Negation in Datalog¬

Extends Expressiveness

Modeling

- Can ask queries with negative components.
- Can express many views (e.g., *cousin*) that we cannot in Datalog.
- Can model databases more succiently

Towards capturing SQL

- Of course, we now can do `except`.
- Can express aggregation using `not`.

NULLs and full-fledged arithmetic in SQL are still a problem.

Negation

Example: Game of Peggly

The game of Peggly is played by two players with a pile of k coins.

- The players alternate turns.
- On a player's turn, the player removes one, two, or three coins.
- If only one coin remains, the player whose turn it is must take it.
- The player to take the last coin loses. (And thus the other player is the winner.)

Generic.

$$\mathit{win}\ (X) \leftarrow \mathit{move}\ (X,\ Y),\ \mathsf{not}\ \mathit{win}\ (Y).$$

Peggly Rules.

$$\mathit{move}\ (X,\ Y) \leftarrow X \geq 1,\ Y \text{ is } X - 1.$$
$$\mathit{move}\ (X,\ Y) \leftarrow X \geq 2,\ Y \text{ is } X - 2.$$
$$\mathit{move}\ (X,\ Y) \leftarrow X \geq 3,\ Y \text{ is } X - 3.$$

$$\mathit{win}\ (\mathit{0}).$$

Non-mon Negation in Datalog¬
Computationally Expensive!

Why? To prove a **not**, one must show that *every* possible proof path fails.

---

state(11) wins because state(9) loses.
state(9) loses because
    state(8) wins
    state(8) wins because state(5) loses.
    state(5) loses because
        state(4) wins
        state(4) wins because state(1) loses.
        state(1) loses because
            that is all.
        AND
        state(3) wins
        state(3) wins because state(1) loses.
        state(1) has been shown to lose.
        AND
        state(2) wins
        state(2) wins because state(1) loses.
        state(1) has been shown to lose.
        AND
        that is all.
    AND
    state(7) wins
    state(7) wins because state(5) loses.
    state(5) has been shown to lose.
    AND
    state(6) wins
    state(6) wins because state(5) loses.
    state(5) has been shown to lose.
    AND
    that is all.

Non-Stratified

Of course there are Datalog¬ programs (databases) that are not even locally stratified.

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

Do we ever need a non-stratified Datalog¬ programs?

Unfortunately, there are natural cases.

Also, the decision problem to determine whether an arbitrary Datalog¬ program is locally stratified is undecidable.

For non-stratified Datalog¬ programs:

- What is the semantics?

  Well, we have choices...
- What is the proof procedure?

  NAFF no longer works.