Adding Negation Datalog^{s,not}

- If we extend Datalog with negation in rule bodies, we will have RA and more

  E.g. $P(x, y) \leftarrow R(x, y),\ \mathit{not}\ S(x, y)$ (*)

  This would capture the difference of RA: $P = R \smallsetminus S$

- Notice that we are not using $\neg$, the classical negation of FOPL in the body

  not will have a slightly different meaning (semantics)

- In RDBs we apply the CWA: If we do not have the (positive) fact p explicitly given, we say p is false, and its negation true

  A form of a non-classical "procedural negation", different from classical $\neg$ used in classical logic and mathematics

  The truth of this new negation is based on "not finding p"

- This is the idea behind *not*

- In (*) $P(a, b)$ becomes true when (a, b) is in R but not in S
- So, *not* is weaker that classical negation:

  $\neg p$ " $\Rightarrow$ " *not* p, but not the other way around
- In classical logic, if we cannot prove $P(a, b)$, we cannot assume $\neg P(a, b)$ is true; it may be undetermined

  $P(a, b)$ may be true in some models and false in others

  But with Datalog and its extensions, we do not consider all kinds of models
- This form of weak procedural negation is used in Datalog with negation, as an extension of the CWA, from the EDB to the IDB
- All this has to be properly established through the definition of the intended semantics

Example: EDB

$$\mathcal{D} = \{P(a,b), P(a,a), P(c,b), Q(a,b), Q(c,c), S(a,a), T(a,b)\}$$

Intensional database with view definitions in Datalog with negation:

$$R(x,y) \leftarrow P(x,y),\ \mathit{not}\ Q(x,y)$$
$$R(x,y) \leftarrow T(x,y), R(x,z),\ \mathit{not}\ S(x,z)$$

P, Q, S, T are base, extensional tables, and R is intensional predicate, recursively defined

It uses both negation and recursion (and implicit disjunction)

The rules as safe: (related to the notion of safe query of RDBs)

- Every variable appearing in a negative body literal also appears in a positive body literal (in the same body); and
- Every variable appearing in a head also appears in the rule body

Example: These rules are safe:

$$P(x,y) \leftarrow S(x), Q(y), \mathit{not}\ S(x,y),\ T(y,z)$$
$$P(x,y) \leftarrow S(x,y), \mathit{not}\ R(x,y)$$

Here all the variables in negative literals refer to values that appear in tables (intentional or not)

These are not safe:

$$P(x,y) \leftarrow S(x), \mathit{not}\ S(x,y),\ T(x,z)$$
$$P(x,y,z) \leftarrow S(x,y), \mathit{not}\ R(x,y)$$

In the first one, what is the meaning of the negative literal?

In the second case, any value from the possibly infinite domain could be used for z, generating an infinite relation

All this in the spirit of safe or domain independent queries of RC

From now on, we accept only programs with safe rules

- We will also assume that Datalog programs with negation are stratified (or have stratified negation)

  Denoted $\text{Datalog}^{s,not}$

- Informally for the moment, this means that there is no recursion via negation

  Not the same as having both negation and recursion: the program on page 44 is stratified

- The semantics of a $\text{Datalog}^{s,not}$ program is also a fix-point semantics

- Compute the intended model, the so-called "standard model"

  Use the same procedure applied to Datalog programs: bottom-up and with forward-propagation

  Treating negation when it pops up as usual difference of RA

Example: (cont.) $\mathcal{D} = \{P(a,b), P(a,a), P(c,b), Q(a,b), Q(c,c), S(a,a), T(a,b)\}$

$R(x,y) \leftarrow P(x,y),\ \mathit{not}\ Q(x,y)$

$R(x,y) \leftarrow T(x,y), R(x,z),\ \mathit{not}\ S(x,z)$

Computation of the standard model $\mathcal{M}$:

1. Propagate values into R using first rule: $R = \{(a,a),(c,b)\}$ (partial computation)

   That rule is used once (only extensional tables in body)

   Bodies are evaluated using relational algebra at every step

2. Use second rule with partial contents of R in body

   $(T \bowtie R)(x,y,z) = \{(\underline{a}, b, \underline{a})\}$, but $(a,a) \in S$

   No new tuples for R at this step; update
   $R = \{(a,a),(c,b)\} \cup \emptyset$

   A fix-point is reached: $\mathcal{M} = \mathcal{D} \cup \{R(a,a),\ R(c,b)\}$

Negation and recursion do not interact: Each predicate has been completely computed (at a previous step) when affected by a negation

Example: EDB: $\mathcal{D} = \{Q(a), Q(b), Q(c), H(b), T(a,b)\}$

Program Π:

$$\begin{array}{lcl} P(x) & \leftarrow & Q(x), \mathit{not}\ R(x) \\ R(x) & \leftarrow & S(x) \\ R(x) & \leftarrow & H(x), \mathit{not}\ S(x) \\ S(x) & \leftarrow & T(x,y), \mathit{not}\ U(y) \\ S(x) & \leftarrow & U(x) \end{array}$$

Stratification of predicates: $S_1 < S_2 < S_3 < S_4$

- $S_1 = \{Q, H, T, U\}$
- $S_2 = \{S\}$ S has to be in a stratum above U, because it is defined by a negated U
- $S_3 = \{R\}$ (idem)
- $S_4 = \{P\}$ (idem)

The upward computation follows the strata

Extensions of predicates at a given stratum are completely computed before moving to the next stratum above

A negated atom, *not* Q, can be invoked by a body of a rule defining a predicate $P \in S_i$ only if it $Q \in S_j$, with $j < i$

Here:

- $\mathcal{M}_1 = \mathcal{D}$ It gives empty extension to U
- $\mathcal{M}_2 = \{S(a)\}$
- $\mathcal{M}_3 = \{R(a), R(b)\}$
- $\mathcal{M}_4 = \{P(c)\}$

The standard model:

$$\begin{aligned}\mathcal{M}(\Pi) &= \mathcal{M}_1 \cup \mathcal{M}_2 \cup \mathcal{M}_3 \cup \mathcal{M}_4 \\ &= \{Q(a), Q(b), Q(c), H(b), T(a,b), S(a), R(a), R(b), P(c)\}\end{aligned}$$

This semantics of stratified programs extends the semantics of (negation-free) Datalog programs

If the program does not have negation the standard model becomes the minimal model

Always welcome: a smooth extension without a radical change of semantics

Example: Beware, recursion may happen through several rules

$P(x, y) \leftarrow R(x, y), not\ S(x)$
$R(x, y) \leftarrow P(y, x)$

Recursive, still stratified

Recursion can be detected through a directed graph with predicates as nodes, with edges from body predicates to the head predicate

If there are loops (cycles), there is recursion

We can annotate edges with, say n, when the body predicate appears negated

A stratified program should have no cycles with an edge annotated with n

Example 1: We can express usual queries in this Datalog^{s,not}

Salaries	Name	Salary
	J. Page	5,000
	V. Smith	3,000
	M. Stowe	7,000
	L. Stark	4,000

Positions	Name	Position
	J. Page	manager
	V. Smith	secretary
	M. Stowe	manager
	K. Stein	accountant

- Referential IC: ψ: $\forall x \forall y(\mathit{Salaries}(x,y) \rightarrow \exists z \mathit{Positions}(x,z))$
- Not satisfied by DB D: $D \not\models \psi$
- Want to catch inconsistencies posing a violation query (view):
  In RC: $\mathcal{Q}^{\psi}(x)$: $\exists y(\mathit{Salaries}(x,y) \wedge \neg \exists z \mathit{Positions}(x,z))$
- In Datalog, stratified Datalog?
- $\mathit{Viol}(x) \longleftarrow \mathit{Salaries}(x,y),\ \mathit{not}\ \mathit{Positions}(x,\underset{\substack{\uparrow\\ \text{not safe}}}{z}))$
- Better introduce auxiliary predicate, and use two rules:
  $\mathit{Viol}(x) \longleftarrow \mathit{Salaries}(x,y),\ \mathit{not}\ \mathit{Aux}(x))$
  $\mathit{Aux}(x) \longleftarrow \mathit{Positions}(x,z)$
- $\mathit{Viol}[D] = \{\langle L.Stark \rangle\} = \mathcal{Q}^{\psi}[D]$ (verify by bottom-up evaluation)

Example 2: (example 1 cont.)

- Above we posed an open query, i.e. with open variables
- A query may also be Boolean, i.e. only true or false in a DB
- In the previous example, we may be interested in knowing if there is a violation or not
- As before (and always), we introduce and define a query-answer predicate

  In this case it becomes a propositional (i. e. Boolean) variable
- $yes \longleftarrow Salaries(x, y),\ not\ Aux(x))$

  (plus the other rule as above)

  All variables in the body are implicitly existentially quantified

  In (safe) RC this would be the Boolean query:

  $$\exists x \exists y (Salaries(x, y) \ \wedge \ \neg Aux(x))$$
- The query is true if atom *yes* belongs to the minimal or standard model, and false otherwise

Example 3: Commonsense Reasoning in Datalog

$$
\begin{array}{rcl}
\mathit{Flies}(x) & \longleftarrow & \mathit{Bird}(x), \mathit{not}\ \mathit{Abnormal}(x) \\
\mathit{Bird}(x) & \longleftarrow & \mathit{Canary}(x) \\
\mathit{Bird}(x) & \longleftarrow & \mathit{Penguin}(x) \\
\mathit{Abnormal}(x) & \longleftarrow & \mathit{Penguin}(x) \\
\mathit{Canary}(\mathit{tweety}). & & \mathit{Penguin}(\mathit{pewee}).
\end{array}
$$

- Query: $\mathit{Flies}(\mathit{tweety})?$
- Check that the standard model is:
  $\mathcal{M} = \{\mathit{Canary}(\mathit{tweety}), \mathit{Penguin}(\mathit{pewee}), \mathit{Abnormal}(\mathit{pewee}), \mathit{Bird}(\mathit{pewee}), \mathit{Bird}(\mathit{tweety}), \mathit{Flies}(\mathit{tweety})\}$
- Tweety does fly!
- First is a commonsense (default) rule: "birds normally fly"
- Unless there are explicitly stated exceptions
- Model minimality makes exceptions minimal!
  CWA: W/o explicit evidence of abnormality, there isn't
- Very useful in KR!

Monotonicity vs. Non-Monotonicity

- The previous example shows something quite relevant
- Negation in Datalog produces a non-monotonic behavior:

  For a program $\Pi \cup \mathcal{D}$ certain consequences may be invalidated when the DB grows: For a ground atom A

  $$A \in \mathcal{M}(\Pi \cup \mathcal{D}) \text{ and } \mathcal{D} \subsetneq \mathcal{D}' \not\Longrightarrow A \in \mathcal{M}(\Pi \cup \mathcal{D}')$$

- Counterexample: In the previous example consider:

  $\mathcal{D} = \{Canary(tweety), Penguin(pewee)\}$

  $\mathcal{D}' = \{Canary(tweety), Penguin(pewee), Abnormal(tweety)\}$

  Now Tweety does not fly: $Flies(tweety) \notin \mathcal{M}(\Pi \cup \mathcal{D}')$

- Commonsense reasoning is intrinsically non-monotonic!
- Datalog (w/o negation) is monotonic (in the same sense and for the same reason as conjunctive queries in DBs):

  $$A \in \underline{\mathcal{M}}(\Pi \cup \mathcal{D}) \text{ and } \mathcal{D} \subsetneq \mathcal{D}' \Longrightarrow A \in \underline{\mathcal{M}}(\Pi \cup \mathcal{D}')$$

Unstratified Datalog?

Example: $\mathcal{D} = \{Q(1), Q(2)\}$ $\quad P(x) \leftarrow Q(x),\ \mathit{not}\ P(x)$

- P is defined by recursion via negation
- Program is not stratified!
- Bottom-up computation of P:
  1. $P = \emptyset$
  2. $P = \{1, 2\}$
  3. $P = \emptyset$
  4. etc., etc.
- Infinite loop! No fixpoint!
- Unstratified programs not considered as query languages in DBs

  Not allowed in SQL3; only Recursive Stratified Datalog
- However, they are useful in Knowledge Representation (coming ...)