Deductive Databases and Logic Programming

(Sommer 2017)

Chapter 8: Negation

- Motivation, Differences to Logical Negation
- Syntax, Supported Models, Clark's Completion
- Stratification, Perfect Model
- Bottom-Up Computation
- Conditional Facts, Well-Founded Model
- Stable Models

Objectives

After completing this chapter, you should be able to:

- explain the difference between negation in logic programming and negation in classical logic
- explain why stratification is helpful, check a given program for the stratification condition.
- compute supported, perfect, well-founded, and stable models of a given program.
- explain the well-founded semantics with conditional facts and elementary program transformations.

Overview

1. Motivation, Differences to classical logic)
2. Syntax, Supported Models
3. Stratification, Perfect Model
4. Conditional Facts, Well-Founded Model
5. Stable Models

Example (Small Library)

book		
BID	Author	Title
Ul189	Ullman	Princ. of DBS and KBS
Llo87	Lloyd	Found. of Logic Progr.

borrowed
BID
Ul189

$$\texttt{available(Author, Title)} \leftarrow \texttt{book(BID, Author, Title)} \land \texttt{not borrowed(BID)}.$$

available	
Lloyd	Found. of Logic Progr.

Motivation (1)

- Queries or view definitions as in the above example are possible in SQL, but cannot be expressed with definite Horn clauses (classical Datalog).
  - ◇ A good query language should be relationally complete, i.e. it should be possible to translate every relational algebra expression into that language.
  - ◇ This goal is reached for Datalog with negation ($\text{Datalog}^{\text{neg}}$) (even without recursion).
- Prolog has an operator `not` (also written `\+`).

Motivation (2)

- Set difference is a natural operation. If it misses in a query language, the user will pose several queries and compute the set difference himself/herself.

  If a query language computes sets, it should be closed under the usual set operations. I.e. a set operation applied to the result of two queries should be expressable as a single query. (One could also require other simple operations, such as counting.) For relations, it should be closed under relational algebra operations. This is not quite the same as relational completeness, because this closure condition holds also e.g. for recursive queries (not expressible in relational algebra).

- It was defined above which facts are false in the minimal model. Up to now, knowledge about false facts cannot be used within in the program.

Not vs. Classical Negation (1)

- The negation operator `not` in Prolog / Datalog^{neg} is not logical negation $\neg$ known from classical logic.
- From the formulas

```
book('Ull89', 'Ullman', 'Princ. of DBS and KBS').
book('Llo87', 'Lloyd', 'Found. of Logic Progr.').
borrowed('Ull89').
available(Author, Title) ← book(BID, Author, Title) ∧
                           ¬borrowed(BID).
```

one cannot conclude that 'Llo87' is not borrowed.

The given formulas specify only the positive information. This was also the motivation for defining the minimal model.

Not vs. Classical Negation (2)

- Therefore, also this is not a logical consequence:

  `available('Lloyd', 'Found. of Logic Progr.')`

- This is a difference to Horn clause Datalog: There, all `answer`-facts in the minimal model are logical consequences of the given program.
- Negative facts must be assumed "by default" if the corresponding positive fact is not provable.
- In order to prove `not` A, Prolog first tries to prove A. If this fails, `not` A is considered "proven": "Negation as (Finite) Failure" / "Default Negation".

Not vs. Classical Negation (3)

- Classical logic is monotonic: If ψ follows from Φ, it also follows from $\Phi \cup \{\varphi'\}$ for any φ'.

  If one has more preconditions, one can derive more (or at least the same).

- This important property does not hold in logic programming: If one adds `borrowed('Llo87')` to the given formulas, one can no longer conclude

  `available('Lloyd', 'Found. of Logic Progr.')`

- Thus, "Nonmonotonic Logic" is used to explain negation in logic programming.

Not vs. Classical Negation (4)

- Not all classical equivalences hold in logic programming: For instance,

  $$\begin{aligned}\mathtt{available(Author, Title)} \leftarrow{} & \mathtt{book(BID, Author, Title)} \land \\ & \neg\mathtt{borrowed(BID)}.\end{aligned}$$

  is logically equivalent to

  $$\begin{aligned}\mathtt{available(Author, Title)} \lor{} & \neg\mathtt{book(BID, Author, Title)} \lor \\ & \neg\neg\mathtt{borrowed(BID)}.\end{aligned}$$

  and thus to

  $$\begin{aligned}\mathtt{borrowed(BID)} \leftarrow{} & \mathtt{book(BID, Author, Title)} \land \\ & \neg\mathtt{available(Author, Title)}.\end{aligned}$$

Not vs. Classical Negation (5)

- In logic programming (with `not` instead of $\neg$), the two formulas have a completely different semantics:

  $$\begin{array}{l}\texttt{borrowed(BID)} \leftarrow \texttt{book(BID, Author, Title)} \wedge \\ \qquad\qquad\qquad\qquad \texttt{not available(Author, Title).}\end{array}$$

- Because no `available`-facts can be derived, Prolog now concludes that also 'Llo87' is borrowed.

- In logic programming, rules can be used in only one direction. The distinction between head and body is important.

  The contraposition of a rule is not used.

Not vs. Classical Negation (6)

- To understand `not`, it is helpful to assume that for every predicate p there is a new, system-defined predicate $\texttt{not\_}p$.

  One can also use modal logic (with an operator "I know that").

- Then exchanging the head literal and a negated body literal is no longer logical contraposition, since $\texttt{not\_}p$ is not necessarily $\neg p$.

- It is not even astonishing that some negation semantics make neither p nor $\texttt{not\_}p$ true for difficult programs like $p \leftarrow \texttt{not}\ p$.

Why not Classical Logic?

NOT is useful/necessary:

- Already the specification of finite relations (as in relational databases) is quite complicated in first order logic.
- The transitive closure cannot be defined in first order logic.

  The well-known rules entail what must be true. But one cannot make sure in classical logic that every other fact is false, i.e. one cannot give an $\leftrightarrow$-definition of `path` that works for any given `edge`-relation.

Overview

1. Motivation, Differences to classical logic)
2. Syntax, Supported Models
3. Stratification, Perfect Model
4. Conditional Facts, Well-Founded Model
5. Stable Models

Syntax

- Now two types of body literals are allowed:
  - Postive body literals (atomic formulas, as usual):

    $$p(t_1, \ldots, t_n)$$

  - Negative body literals (default negation of an atomic formula):

    $$\mathtt{not}\ p(t_1, \ldots, t_n)$$

- The default negation operator **not** cannot be used in the head of a rule.

  This corresponds to the above view that "**not**_p" is a system defined predicate. One cannot introduce rules that define this predicate.

SLDNF-Resolution (1)

- SLDNF-Resolution (SLD-resolution with negation as failure) is a generalization of SLD-resolution to programs with negative body literals.

  Some authors think that it is more precise to say "finite failure".

- As in SLD-resolution, a tree is constructed, where the nodes are marked with goals (conjunctions of positive and negative literals).

  Seen as a refutation proof, one can also view the goals as disjunction of the opposite literals.

- If the selected literal is a positive literal, child nodes are constructed as in SLD-resolution.

SLDNF-Resolution (2)

- If the selected literal is a negative literal, SLDNF-resolution calls itself recursively with the corresponding positive literal as query.

  I.e. a new tree is constructed, the root marked the the positive literal.

  ◇ If this tree is finite and contains no success node (empty goal), the negative literal is considered proven, and the calling node gets a single child node with the negative literal removed.

  ◇ If the tree contains a success node, the calling node is a failure node (without child nodes).

SLDNF-Resolution (3)

- Most authors require that a negative literal can only be selected if it is ground.
- The reason is the probably unexpected local quantification, see next slides.
- If the goal is not empty, but the selection function cannot select a literal (because only nonground negative literals are left), evaluation "flounders".

  This is an error condition.

- Most Prolog systems do not obey this restriction.

Range Restriction (1)

- `not_`p can only be called with the binding pattern `bb...b`.
- I.e. every variable of the rule must occur in a positive body literal.

  With further restrictions if built-in predicates are used.

- Many Prolog systems evaluate also negative literals with variables. But then the usual quantification is inverted:
  - ◇ `not` p(`X`) is successful if p(`X`) fails for all `X`.
  - ◇ p(`X`) is successful if it succeeds for at least one `X`.

Range Restriction (2)

- Consider the following example:

$$p(\mathtt{X}) \leftarrow \mathbf{not}\ r(\mathtt{X}) \wedge q(\mathtt{X}).$$
$$q(a).$$
$$r(b).$$

- Most Prolog systems will answer the query "$p(\mathtt{X})$" with "no", since already $\mathbf{not}\ r(\mathtt{X})$ fails.

  In this case, there are actually two different variables named "$\mathtt{X}$", since $\mathtt{X}$ within $\mathbf{not}$ is implicitly $\exists$-quantified.

- If one exchanges the two body literals, every Prolog systems answers withe same query with $\mathtt{X} = a$.

Range Restriction (3)

Remark (Anonymous Variables):

- Anonymous variables in negated body literals can be useful.
- Suppose that the table `borrowed` is extended:

borrowed	
BID	User
U1189	Brass

- Formally, the following rule would not be range-restricted, but Prolog would work as expected:

$$\texttt{available(Author, Title)} \leftarrow \texttt{book(BID, Author, Title)} \land$$
$$\texttt{not borrowed(BID, \_)}.$$

Range Restriction (4)

Remark, continued:

- Probably, most deductive database systems permit negative body literals with anonymous variables.

  However, these variables are $\forall$-quantified in the body, whereas all other variables that occur only in the body are $\exists$-quantified.

  One can also say that anonymous variables are $\exists$-quantified immediately in front of the atomic formula, i.e. inside the negation.

- This is also consistent with the idea that anonymous variables project away unnecessary columns.

- Exercise: If anonymous variables were not allowed in negated literals, how would one define `available`?

Exercise

- Let the following EDB-relations be given:
  - `lecture_hall(RoomNo, Capacity).`
  - `reservation(RoomNo, Day, From, To, Course).`
- Which lecture halls are free on tuesdays, 8^{30}–10^{00}?
- What is the largest capacity of a lecture hall?
- Is there a time at which all lecture halls are used?

  If there is such a time at all, also one of the times in `From` satisfies this condition (Proof: Go back from the given time, when all lecture halls are used, to the nearest start of a reservation.). Thus, it is not necessary to check all possible times.

Clark's Completion (1)

- The first approach to define a semantics of negation in logic programming non-operationally was (probably) Clark's Completion (also called CDB: "completed database") [1978].
- Basically, the idea was to turn "$\leftarrow$" into "$\leftrightarrow$.
- E.g., if the only rule about p is

$$p(X) \leftarrow q(X) \wedge r(X)$$

the definition of p in the CDB is (equivalent to)

$$\forall X\colon\; p(X) \leftrightarrow q(X) \wedge r(X)$$

Clark's Completion (2)

- When variables occur only in the body, e.g. Y in

$$p(X) \leftarrow q(X, Y)$$

it is normally not important whether

- ◇ it is universally quantified over the entire rule:

$$\forall X, Y\colon\ p(X) \leftarrow q(X, Y)$$

- ◇ or existentially over the body (equivalent):

$$\forall X\colon\ p(X) \leftarrow \exists Y\colon q(X, Y)$$

- But for the CDB only the second version works:

$$\forall X\colon\ p(X) \leftrightarrow \exists Y\colon q(X, Y)$$

Clark's Completion (3)

- If there are several rules about one predicate, the rule bodies must be connected disjunctively.
- E.g. consider

$$p(a, X) \leftarrow q(X).$$
$$p(b, X) \leftarrow r(X).$$

- The rule heads must be normalized: New variables are introduced for the arguments of the head, and equated with the original arguments in the body:

$$\begin{aligned} \forall Y_1, Y_2\colon p(Y_1, Y_2) \leftrightarrow {} & (\exists X\colon Y_1 = a \land Y_2 = X \land q(X)) \lor \\ & (\exists X\colon Y_1 = b \land Y_2 = X \land r(X)) \end{aligned}$$

Clark's Completion (4)

- E.g., consider this program:

$$\begin{array}{l} p(X) \leftarrow q(X) \wedge \texttt{not}\ r(X). \\ q(a). \\ q(b). \\ r(b). \end{array}$$

- Clark's completion (as follows) implies e.g. $p(a)$:

$$\begin{array}{l} \forall X\ p(X) \leftrightarrow q(X) \wedge \neg r(X). \\ \forall X\ q(X) \leftrightarrow X = a \vee X = b. \\ \forall X\ r(X) \leftrightarrow X = b. \end{array}$$

  In addition it contains an equality theory that includes, e.g., $a \neq b$ (unique names assumption, UNA).

Clark's Completion (5)

- Consider the program

$$p \leftarrow p.$$

- The definition of p in Clark's Completion is

$$p \leftrightarrow p$$

  i.e. p can be true or false.

- SLDNF resolution can prove neither p nor `not` p.

  It always gets into an infinite loop.

- However, if one uses a deductive database with bottom-up evaluation, it is clear that p is false.

  This motivates the search for stronger negation semantics, see below.

Clark's Completion (6)

- Consider the program

$$p \leftarrow \texttt{not}\ p.$$

- The definition of p in Clark's Completion is

$$p \leftrightarrow \neg p$$

  which is inconsistent.

- SLDNF resolution can prove neither p nor $\texttt{not}\ p$.
- However, if the program contains other, unrelated predicates, SLDNF resolution would give reasonable positive and negative answers for them, while Clark's completion implies everything.

Clark's Completion (7)

- Of course, the rule $p \leftarrow$ **not** p is strange and contradictory: p is provable iff p is not provable.

  In classical logic $p \leftarrow \neg p$ is simply equivalent to p, i.e. p is true. However, as explained above, when negation is used, logic programming rules do not behave as the corresponding formulas in classical logic.

- However, such a case can be hidden in a large program, and be totally unrelated to a given query.
- In order to support goal-directed query evaluation procedures, such cases must be excluded or the semantics must "localize" the consistency problem.

Overview

1. Motivation, Differences to classical logic)
2. Syntax, Supported Models
3. Stratification, Perfect Model
4. Conditional Facts, Well-Founded Model
5. Stable Models

Stratification (1)

- In order to avoid the $p \leftarrow \texttt{not}\ p$ problem, the class of stratified programs is introduced.
- Note that negative body literals $\texttt{not}\ p(t_1, \ldots, t_n)$ can be easily evaluated if the complete extension of p was already computed previously.

  Variables among the arguments are already bound to a concrete value because of the range restriction. Thus, one only has to check whether the argument tuple is contained in the extension of p.
- This means that p must not depend on a predicate that depends on $\texttt{not}\ p$.

  In short: Recursion through negation is excluded.

Stratification (2)

Definition (Level Mapping of the Predicates):

- A level mapping of the predicates $\mathcal{P}$ is a mapping

$$\ell\colon \mathcal{P} \to \mathbb{N}_0.$$

- The domain of this mapping is extended to atomic formulas through

$$\ell(p(t_1, \ldots, t_n)) := \ell(p).$$

Note:

- The purpose of this level mapping is to define an evaluation sequence: One starts with predicates of level 0, continues with level 1, and so on.

Stratification (3)

Definition (Stratified Program):

- A program P is stratified if and only if there is a level mapping ℓ such that for each rule

$$A \leftarrow B_1 \wedge \cdots \wedge B_m \wedge \mathtt{not}\ C_1 \wedge \cdots \wedge \mathtt{not}\ C_n$$

  the following holds:

  ◇ $\ell(B_i) \leq \ell(A)$ for $i = 1, \ldots, m$, and

  ◇ $\ell(C_j) < \ell(A)$ for $i = 1, \ldots, n$.

- Such a level mapping is called a stratification of P.

Stratification (4)

- One can compute a stratification as follows:
  - ◇ Let k be the number of different predicates in the program.
  - ◇ Assign level 0 to every predicate.
  - ◇ Whenever the condition is violated for a rule, and the level of the predicate in the head is less than k, increment it.
  - ◇ If the level of a predicate reaches k, the program is not stratified. Otherwise, when a stable state is reached, this is a valid stratification.

Stratification (5)

- The predicate dependency graph for programs with negation is defined as follows:
  - Nodes: Predicates that occur in the program.

    Here **not** p does *not* count as a predicate on its own.
  - There is a positive edge from q to p iff there is a rule of the form $p(\ldots) \leftarrow \ldots \wedge q(\ldots) \wedge \ldots$
  - There is a negative edge from q to p iff there is a rule of the form $p(\ldots) \leftarrow \ldots \wedge \mathbf{not}\ q(\ldots) \wedge \ldots$.
- A program is stratified if and only if there is no cycle that contains at least one negative edge.

Perfect Model (1)

- For defining when an interpretation is a model of a program one treats **not** like classical negation $\neg$.
- Thus, an interpretation $\mathcal{I}$ is a model of a program P iff for every rule

$$A \leftarrow B_1 \wedge \cdots \wedge B_m \wedge \textbf{not } C_1 \wedge \cdots \wedge \textbf{not } C_n$$

and every variable assignment $\mathcal{A}$ the following holds:

◇ If $(\mathcal{I}, \mathcal{A}) \models B_i$ for $i = 1, \ldots, m$ and $(\mathcal{I}, \mathcal{A}) \not\models C_j$ for $j = 1, \ldots, n$, then $(\mathcal{I}, \mathcal{A}) \models A$.

If one identifies again a Herbrand model $\mathcal{I}$ with its set of true facts, a ground literal $\textbf{not } p(c_1, \ldots, c_n)$ is true in $\mathcal{I}$ if and only if $p(c_1, \ldots, c_n) \notin \mathcal{I}$.

Perfect Model (2)

Problem:

- With negation, the minimal Herbrand model is no longer unique:

  $$p \leftarrow \mathbf{not}\ q.$$

  This program has two minimal models (it is logically equivalent to $p \vee q$):

  - $\mathcal{I}_1 = \{p\}$.
  - $\mathcal{I}_2 = \{q\}$.

- Of these, only $\mathcal{I}_1$ is intuitively right (intended model): Since there are no rules about q, one cannot prove q. Thus, it should be false.

Perfect Model (3)

Idea:

- Predicate minimization with priorities:
  - ◇ It is natural to compute the predicate extensions in the sequence given by the level mapping.
  - ◇ Then predicates with lower level are minimized with higher priority.

    They can choose first and they want to have a minimal extension.
  - ◇ In the example $p \leftarrow \mathtt{not}\ q$ (e.g., $\ell(q)=0$, $\ell(p)=1$) q is minimized with higher priority, i.e. it is more important to make q false than to make p false. Therefore, one chooses $\mathcal{I}_1 = \{p\}$.

Perfect Model (4)

Definition (Prioritized Minimal Model):

- Let a level mapping ℓ for a program P be given.
- A Herbrand model $\mathcal{I}_1$ of P is preferable to a Herbrand model $\mathcal{I}_2$ ($\mathcal{I}_1 \prec_\ell \mathcal{I}_2$) iff there is $i \in \mathbb{N}_0$ with
  - ◇ $\mathcal{I}_1(p) = \mathcal{I}_2(p)$ for all predicates p with $\ell(p) < i$.
  - ◇ $\mathcal{I}_1(p) \subseteq \mathcal{I}_2(p)$ for all predicates p with $\ell(p) = i$.
  - ◇ $\mathcal{I}_1(p) \neq \mathcal{I}_2(p)$ for at least one p with $\ell(p) = i$.
- $\mathcal{I}_1 \preceq_\ell \mathcal{I}_2$ iff $\mathcal{I}_1 \prec_\ell \mathcal{I}_2$ or $\mathcal{I}_1 = \mathcal{I}_2$.

Perfect Model (5)

Theorem/Definition:

- Every stratified program P has exactly one minimal model $\mathcal{I}_0$ with respect to $\preceq_\ell$.
- This model $\mathcal{I}_0$ is called the perfect model of P.
- The perfect model does not depend on the exact stratification. If ℓ und ℓ' are two stratifications for P, the minimal model with respect to $\preceq_\ell$ is also minimal with respect to $\preceq_{\ell'}$.

  Actually, the original definition of the perfect model does not use a level mapping but the priority relation between the predicates given by the rules of the program.

Bottom-Up Evaluation (1)

- One first applies rules about predicates of level 0. These do not contain negation.
- Then one applies the rules about predicates of level 1. These refer negatively only to predicates of level 0. But the extensions of these predicates are already known. And so on.
- When the rules are applied in a sequence obtained from topologically sorting the predicate dependency graph, one automatically gets this order compatible with the predicate levels.

Bottom-Up Evaluation (2)

Definition (Generalized T_P-Operator):

$$
\begin{aligned}
T_{P,\mathcal{J}}(\mathcal{I}) := \{F \in \mathcal{B}_\Sigma \mid\ & \text{There is a rule} \\
& A \leftarrow B_1 \wedge \cdots \wedge B_m \\
& \qquad \wedge\, \mathtt{not}\ C_1 \wedge \cdots \wedge \mathtt{not}\ C_n \\
& \text{in } P \text{ and ground substitution } \theta, \\
& \text{such that} \\
& \quad \theta(A) = F, \\
& \quad \theta(B_i) \in \mathcal{I} \cup \mathcal{J} \text{ for } i = 1, \ldots, m, \\
& \quad \theta(C_i) \notin \mathcal{J} \text{ for } i = 1, \ldots, n\}.
\end{aligned}
$$

Bottom-Up Evaluation (3)

Iterated Fixpoint Computation:

- Let ℓ be a stratification of P with maximal level k.
- Let P_i be the rules about predicates of level i, i.e.

  $P_i := \{A \leftarrow B_1 \wedge \cdots \wedge B_m \wedge \mathbf{not}\ C_1 \wedge \cdots \wedge \mathbf{not}\ C_n \ \in P \mid \ell(A) = i\}.$
- Let $\mathcal{I}_0 := \emptyset$ and $\mathcal{I}_{i+1} := \mathcal{I}_i \cup \mathit{lfp}(T_{P_i,\mathcal{I}_i})$.
- Then $\mathcal{I}_{k+1}$ is the perfect model of P.

Overview

Non-Stratified Negation (1)

- Some practically useful programs are not stratified.
- This happens e.g., if there is a "state" argument, and when one defines a predicate for the next state, one uses it negatively for the previous state:

$$\texttt{odd}(\texttt{X}) \leftarrow \texttt{succ}(\texttt{Y}, \texttt{X}) \wedge \texttt{not odd}(\texttt{Y}).$$

$$\texttt{succ}(0, 1).$$
$$\texttt{succ}(1, 2).$$
$$\vdots$$
$$\texttt{succ}(n-1, n).$$

Since succ is acyclic, an odd-fact does not depend on itself negatively. But if one looks only at the predicates, there is the negative cycle. It depends on the data whether a program is "dynamically stratified".

Non-Stratified Negation (2)

- A similar program defines the winning states of a game in which a player loses if he cannot make another move:

$$\mathtt{win(X)} \leftarrow \mathtt{move(X,Y)} \land \mathtt{not\ win(Y)}.$$

E.g., in the Nim game, there are three piles of objects (e.g. matches). A move is to take any number of objects (at least one) from a single pile. The player who takes the last object wins (often it is played in the opposite way). [https://en.wikipedia.org/wiki/Nim].

- A winning state is one for which there is a strategy to win the game.

No matter what the opponent does.

Non-Stratified Negation (3)

- Negation is only a special case of aggregation functions, it simply means: $\texttt{count}(\ldots) = 0$.
- The "bill of materials" problem, which was one of the applications that motivated recursive queries in databases, uses unstratified aggregation:
  - ◇ One is given the prices of elementary parts,
  - ◇ and the parts lists of modules, which are composed recursively to still larger modules.
  - ◇ One has to compute the prices of all modules (the end products are the root modules).

Interpretations, Models

- An extended Herbrand-interpretation (EH-interp.) is a set $\mathcal{I}$ of positive and negative ground literals.

  So this is a four-valued logic: p is true in $\mathcal{I}$ iff $p \in \mathcal{I}$ and $\texttt{not}\ p \notin \mathcal{I}$. It is false iff $\texttt{not}\ p \in \mathcal{I}$ and $p \notin \mathcal{I}$. It is undefined iff $p \notin \mathcal{I}$ and $\texttt{not}\ p \notin \mathcal{I}$. It is inconsistent iff $p \in \mathcal{I}$ and $\texttt{not}\ p \in \mathcal{I}$.

- An EH-interpretation is three-valued (consistent) if it does not contain a ground fact and its negation.

- An interpretation $\mathcal{I}$ is a model of a logic program P if for every ground instance of a rule in P, if each body literal is contained in $\mathcal{I}$, then also the head is contained in $\mathcal{I}$.

Well-Founded Semantics (1)

- The well-founded semantics (WFS) defines a single three-valued EH-interpretation, the well-founded model, for each program.
- For the problematic program

$$p \leftarrow \mathtt{not}\ p.$$

  the well-founded model contains neither p nor $\mathtt{not}\ p$.

  I.e. p has the third truth value "undefined". One can view this as a kind of error value.

Well-Founded Semantics (2)

- The well-founded model (WFM) can be defined as least fixed point of a modified T_P operator.
- Given a three-valued EH-interpretation $\mathcal{I}$, this permits to deduce positive and negative literals.
- The deduction of positive literals is as usual: For each ground instance

$$A \leftarrow B_1 \wedge \cdots \wedge B_m \wedge \texttt{not}\ C_1 \wedge \cdots \wedge \texttt{not}\ C_n$$

of a rule in P, if the given interpretation $\mathcal{I}$ contains all the (positive and negative) body literals, then $A \in T_P(\mathcal{I})$.

Well-Founded Semantics (3)

- Negative literals are deduced based on the notion of an unfounded set: Given a three-valued EH-interpretation $\mathcal{I}$ and a program P, a set of facts $\mathcal{J}$ is an unfounded set iff for each ground instance

  $$A \leftarrow B_1 \wedge \cdots \wedge B_m \wedge \texttt{not}\ C_1 \wedge \cdots \wedge \texttt{not}\ C_n$$

  of a rule in P with $A \in \mathcal{J}$
  - ◇ There is $1 \leq i \leq m$ with $\texttt{not}\ B_i \in \mathcal{I}$, or
  - ◇ there is $1 \leq i \leq n$ with $C_i \in \mathcal{I}$, or
  - ◇ there is $1 \leq i \leq m$ with $B_i \in \mathcal{J}$

  (called "witness of unusability of the rule").

Well-Founded Semantics (4)

- It is obvious that the union of two unfounded sets is again an unfounded set.
- Thus, there is a unique maximal unfounded set.
- Now the T_P-operator for the well-founded model adds the negative counterpart (the default negation) of all elements of the maximal unfounded set.

  Plus the derived positive ground literals (see Slide 8-51).
- This T_P operator is monotonic with respect to $\subseteq$ (information order).
- Thus, the least fixed point exists. This is the WFM.

Well-Founded Semantics (5)

Example:

- Program P: $s \leftarrow \mathtt{not}\ r.$
  $r \leftarrow q.$
  $q \leftarrow r.$
  $q \leftarrow \mathtt{not}\ p.$
  $p.$
- $\mathcal{I}_0 := \emptyset.$
- $\mathcal{I}_1 := \{p\}.$
- Now $\{q, r\}$ is an unfounded set with respect to $\mathcal{I}_1$.
  $\mathcal{I}_2 := \{p, \mathtt{not}\ q, \mathtt{not}\ r\}.$
- $\mathcal{I}_3 := \{p, \mathtt{not}\ q, \mathtt{not}\ r, s\}$ (well-founded model of P).

Well-Founded Semantics (6)

Theorem:

- For stratified programs P, the well-founded model of P is the perfect model of P.

Note/Theorem:

- The above definition of “model” only required that when the body is true, the head must be true.
- The well-founded model also satisfies that when the body is undefined (no negation of a body literal is in the model), the head is at least undefined (the negation of the head is also not in the model).

Problem

- For stratified programs, the semantics is clear, but stratified programs are not enough in practice.
- There are about 15 proposals for the semantics of nonmonotonic negation. The WFS is one of them.

  Actually, the well-founded semantics and the stable model semantics "survived", the others are only interesting for specialists. The stable model semantics lead to a new programming formalism, "answer set programming".

- Which one(s) are natural and free of surprises?
- Are there good semantics we do not know yet?

Abstract Semantics

Definition:

- A semantics is a mapping $\mathcal{S}$, which assigns to every program P a set of EH-interpretations which are models of P.
- $\mathcal{S}(P) = \mathcal{S}(ground(P))$.

  I.e. it suffices to consider ground programs.
- If a ground literal A does not occur in $ground(P)$, then $\mathbf{not}\ A \in \mathcal{I}$ and $A \notin \mathcal{I}$ for every $\mathcal{I} \in \mathcal{S}(P)$.

  I.e. in this obvious case, A must be false.

Program Transformations (1)

Definition:

- A program-transformation is a relation $\mapsto$ between ground logic programs.

  This is a relation, not a function: Since a transformation might be applied to different rules or predicates in the program, the same "input" program can be related to several alternative "output" programs.

- A semantics $\mathcal{S}$ allows a transformation $\mapsto$ iff

$$P_1 \mapsto P_2 \implies \mathcal{S}(P_1) = \mathcal{S}(P_2).$$

  I.e. the transformation does not change the meaning of the program: The set of models (selected by the semantics) before and after the transformation is the same.

Program Transformations (2)

Deletion of Tautologies:

- $P_1 \mapsto_T P_2$ iff P_1 contains a rule of the form

$$A \leftarrow \ldots \wedge A \wedge \ldots,$$

and P_2 is the result of deleting this rule from P_1.

Example:

- P_1: $q \leftarrow \mathtt{not}\ p.$
  $p \leftarrow p \wedge q.$
- P_2: $q \leftarrow \mathtt{not}\ p.$

Program Transformations (3)

Unfolding (Partial Evaluation):

- Replace a positive body literal B by the bodies of all rules about B.

- P_1:

$$\begin{aligned} p &\leftarrow q \wedge \mathbf{not}\ r. \\ q &\leftarrow s \wedge \mathbf{not}\ t. \\ q &\leftarrow u. \end{aligned}$$

- P_2:

$$\begin{aligned} p &\leftarrow s \wedge \mathbf{not}\ t \wedge \mathbf{not}\ r. \\ p &\leftarrow u \wedge \mathbf{not}\ r. \\ q &\leftarrow s \wedge \mathbf{not}\ t. \\ q &\leftarrow u. \end{aligned}$$

Program Transformations (4)

Deletion of Nonminimal (Subsumed) Rules:

- A rule $A \leftarrow L_1 \wedge \cdots \wedge L_n$ can be deleted if there is another rule $A \leftarrow L_{i_1} \wedge \cdots \wedge L_{i_k}$ such that $\{L_{i_1}, \ldots, L_{i_k}\} \subset \{L_1, \ldots, L_n\}$.

  We treat rule bodies here as sets. My habilitation thesis uses an explicit operation for reordering body literals.

Example:

- P_1: $p \leftarrow p \wedge \mathbf{not}\ q.$
  $p.$

  One could remove the tautological rule or unfold and get $p \leftarrow \mathbf{not}\ q$.

- P_2: $p.$

Program Transformations (5)

Normal Form:

P_0 is a normal form of P wrt $\mapsto$ iff

- $P \mapsto^* P_0$ and
- there is no P_1 with $P_0 \mapsto P_1$.

Confluence:

$\mapsto$ is confluent iff for all P_1, P_2, P_3:

- If $P_1 \mapsto^* P_2$ and $P_1 \mapsto^* P_3$,
- then there is P_4 with $P_2 \mapsto^* P_4$ and $P_3 \mapsto^* P_4$.

Program Transformations (6)

Theorem:

- The rewriting system $\mapsto$ consisting of the above three transformations is terminating, i.e. every program has a normal form.
- The rewriting system $\mapsto$ is also confluent.
- Therefore, every program has a unique normal form.

Definition:

- The normal form of P is called the weak residual program of P.

Conditional Facts (1)

Conditional Fact:

- Ground rule with only negative body literals:

$$A \leftarrow \mathbf{not}\ B_1 \wedge \cdots \wedge \mathbf{not}\ B_n.$$

- The weak residual program is a set of conditional facts.
- It can be computed with a variant of the usual T_P-operator which simply delays the evaluation of negative body literals.

  The range restriction ensures that all variables in the rule will be bound if we insert conditional facts for the positive body literals.

Conditional Facts (2)

Direct Consequence Operator T_P:

$$p(a) \leftarrow \mathtt{not}\ s(b) \land \mathtt{not}\ r(b).$$

$$\boxed{p(\mathtt{X}) \leftarrow q_1(\mathtt{X}) \land q_2(\mathtt{X},\mathtt{Y}) \land \mathtt{not}\ r(\mathtt{Y}).}$$

$$q_1(a) \qquad q_2(a,b) \leftarrow \mathtt{not}\ s(b).$$

Theorem:

- $\mathit{lfp}(T_P)$ (without nonminimal cond. facts) is exactly the normal form of $\mathit{ground}(P)$.

Conditional Facts (3)

Example:

- $\texttt{odd(X)} \leftarrow \texttt{succ(Y,X)} \wedge \texttt{not odd(Y)}.$

  $\texttt{succ}(0, 1).$
  $\texttt{succ}(1, 2).$
  $\ldots$
  $\texttt{succ}(n - 1, n).$

Normal Form:

- $\texttt{odd}(1) \leftarrow \texttt{not odd}(0).$ $\texttt{succ}(0, 1).$
  $\texttt{odd}(2) \leftarrow \texttt{not odd}(1).$ $\texttt{succ}(1, 2).$
  $\ldots$ $\ldots$
  $\texttt{odd}(n) \leftarrow \texttt{not odd}(n - 1).$ $\texttt{succ}(n - 1, n).$

Relation to Minimal Models

Order Among the EH-Interpretations:

$\mathcal{I}_1 \prec \mathcal{I}_2$ iff

- $\mathcal{I}_1 \subset \mathcal{I}_2$, but
- $\mathcal{I}_1$ and $\mathcal{I}_2$ contain the same negative literals.

Theorem:

- A semantics $\mathcal{S}$ allows unfolding, elimination of tautologies and of nonminimal rules iff
- $\mathcal{S}(P_1) = \mathcal{S}(P_2)$ for all programs P_1 and P_2, which have the same set of $\prec$-minimal EH-models.

WFS-Characterization (1)

Positive Reduction:

- Replace a rule of the form

$$A \leftarrow L_1 \wedge \cdots \wedge L_{i-1} \wedge \texttt{not}\ B \wedge L_{i+1} \wedge \cdots \wedge L_n,$$

where B occurs in no rule head, by

$$A \leftarrow L_1 \wedge \cdots \wedge L_{i-1} \wedge L_{i+1} \wedge \cdots \wedge L_n.$$

WFS-Characterization (2)

Negative Reduction:

- Delete a rule of the form

$$A \leftarrow L_1 \wedge \cdots \wedge \texttt{not}\ B \wedge \cdots \wedge L_n,$$

where $B \leftarrow \mathit{true}$ is given as a fact.

Theorem

- Also the rewriting system extended by these two transformations is terminating and confluent.

WFS-Characterization (3)

Residual Program:

- The normal form of a program P with respect to all five transformations is called the residual program $res(P)$ of P.

Example (continued on next slide):

- $\texttt{odd}(\texttt{X}) \leftarrow \texttt{succ}(\texttt{Y}, \texttt{X}) \land \texttt{not odd}(\texttt{Y}).$

  $\texttt{succ}(0, 1).$
  $\texttt{succ}(1, 2).$
  $\ldots$
  $\texttt{succ}(n-1, n).$

WFS-Characterization (4)

Derivable Conditional Facts:

- odd(1) $\leftarrow$ not odd(0).
  odd(2) $\leftarrow$ not odd(1).
  odd(3) $\leftarrow$ not odd(2).
  ...

Residual Program:

- odd(1). succc(0, 1).
  succ(1, 2).
  odd(3). succ(2, 3).
  ... ...
  succ($n-1, n$).

WFS-Characterization (5)

Example

- $p \leftarrow \mathbf{not}\ p.$
- This program cannot be further reduced, it is its own residual program.

  The well-founded semantics leaves p undefined in this case. One can understand the undefined truth value as a kind of localized error indicator.

Theorem

- The well-founded semantics allows the above five transformations.

WFS-Characterization (6)

Theorem:

- The well-founded model of P can be directly read from the residual program $res(P)$:
  - A is true in the well-founded model iff $res(P)$ contains the fact $A \leftarrow$ true.
  - A is false in the well-founded model iff $res(P)$ contains no rule about A.
  - All other ground atoms are undefined in the well-founded model.

WFS-Characterization (7)

Theorem:

- Let $\mathcal{S}$ be any semantics that permits the above five transformation.
- Then for all programs P and all $\mathcal{I} \in \mathcal{S}(P)$:
  - If A is true in the well-founded model of P, then $A \in \mathcal{I}$.
  - If A is false in the well-founded model of P, i.e. $\mathbf{not}\ A$ is contained in it, then $\mathbf{not}\ A \in \mathcal{I}$.
- I.e.: The well-founded semantics is the weakest semantics that permits the five transformations.