Implementation Techniques

Acknowledgements to Angele, Gehrke, STI
Word of Caution

- There is not the one silver bullet
- In actual systems, different techniques are combined
Many further techniques

- Truth maintenance
- Tabling
- Tail elimination
- Compilation into SAT solvers (especially for EDLP)
- Optimization of information passing (comparable to join ordering)
- ….

Plus extensions for
- External predicates, connection to relational databases
- Combinations with first order logics
- Arithmetics
- ….

What may help in one case may cause harm in another one!
- Forward Chaining
  - Naive Evaluation
  - Semi-naive evaluation

- Backward Chaining
  - SLDNF / Prolog
  - Dynamic Filtering

- Magic Sets

- Alternating Fixpoint
Example knowledge base

- The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.

- Prove that Col. West is a criminal
... it is a crime for an American to sell weapons to hostile nations:
\[
\text{American}(x) \land \text{Weapon}(y) \land \text{Sells}(x,y,z) \land \text{Hostile}(z) \Rightarrow \text{Criminal}(x)
\]

Nono ... has some missiles, i.e., \(\exists x \text{ Owns}(\text{Nono},x) \land \text{Missile}(x)\):
\[
\text{Owns}(\text{Nono},M_1) \land \text{Missile}(M_1)
\]

... all of its missiles were sold to it by Colonel West
\[
\text{Missile}(x) \land \text{Owns}(\text{Nono},x) \Rightarrow \text{Sells}(\text{West},x,\text{Nono})
\]

Missiles are weapons:
\[
\text{Missile}(x) \Rightarrow \text{Weapon}(x)
\]

An enemy of America counts as "hostile``:
\[
\text{Enemy}(x,\text{America}) \Rightarrow \text{Hostile}(x)
\]

West, who is American ...
\[
\text{American}(\text{West})
\]

The country Nono, an enemy of America ...
\[
\text{Enemy}(\text{Nono},\text{America})
\]
function FOL-FC-Ask(KB, α) returns a substitution or false

repeat until new is empty

new ← {}

for each sentence r in KB do

    (p₁ ∧ ... ∧ pₙ ⇒ q) ← STANDARDIZE-Apart(r)

    for each θ such that (p₁ ∧ ... ∧ pₙ)θ = (p₁' ∧ ... ∧ pₙ')θ

        for some p₁', ..., pₙ' in KB

            q' ← SUBST(θ, q)

            if q' is not a renaming of a sentence already in KB or new then do

                add q' to new

                φ ← UNIFY(q', α)

                if φ is not fail then return φ

        add new to KB

return false
Forward chaining proof

\begin{align*}
\text{American(\text{West})} & \quad \text{Missile(MI)} & \quad \text{Owns(Nono,MI)} & \quad \text{Enemy(Nono,\text{America})}
\end{align*}
Forward chaining proof

Diagram:

```
American(West)  Weapon(M1)  Sells(West,M1,Nono)  Hostile(Nono)
       |            |                |                     |
       |            |                |                     |
       |            |                |                     |
       |            |                |                     |
Missile(M1)    Owns(Nono,M1)  Enemy(Nono,America)
```
Forward chaining proof

```
Criminal(West)

Weapon(MI)  Sells(West,MI,Nono)  Hostile(Nono)

American(West)  Missile(MI)  Owns(Nono,MI)

Enemy(Nono,America)
```
Properties of forward chaining

- Sound and complete for first-order definite Horn clauses, which means that it computes all entailed facts correctly.

- **Datalog** = first-order definite clauses + no functions

- FC terminates for Datalog in finite number of iterations

- May not terminate in general if $\alpha$ is not entailed
  - This is unavoidable: entailment with definite clauses is semi-decidable
Efficiency of forward chaining

- **Incremental forward chaining**: no need to match a rule on iteration $k$ if a premise wasn’t added on iteration $k-1$
  - match each rule whose premise contains a newly added positive literal

- **Magic Sets**: rewriting the rule set, using information from the goal, so that only relevant bindings are considered during forward chaining

\[ Magic\_Criminal(x) \land American(x) \land Weapon(y) \land Sells(x,y,z) \land Hostile(z) \implies Criminal(x) \]
add $Magic\_Criminal(West)$ to the KB

- Matching itself can be expensive
  - **Database indexing** allows $O(1)$ retrieval of known facts: for a given fact it is possible to construct indices on all possible queries that unify with it
  - **Subsumption lattice** can get very large; the costs of storing and maintaining the indices must not outweigh the cost for facts retrieval.

- **Forward chaining is widely used in deductive databases**
function FOL-BC-Ask(KB, goals, θ) returns a set of substitutions

  inputs: KB, a knowledge base
  goals, a list of conjuncts forming a query
  θ, the current substitution, initially the empty substitution { }

  local variables: ans, a set of substitutions, initially empty

  if goals is empty then return {θ}
  q' ← SUBST(θ, FIRST(goals))
  for each r in KB where STANDARDIZE-Apart(r) = (p₁ ∧ ... ∧ pₙ ⇒ q)
    and θ' ← UNIFY(q, q') succeeds
    ans ← FOL-BC-Ask(KB, [p₁, ..., pₙ|REST(goals)], COMPOSE(θ, θ')) ∪ ans
  return ans
Backward chaining example

Criminal(West)
Backward chaining example
Backward chaining example
Backward chaining example

Diagram:

- Criminal(West)
- {x/West}
- American(West)
- Weapon(y)
- Sells(x,y,z)
- Hostile(z)
- Missile(y)
Backward chaining example
Backward chaining example
Backward chaining example
Properties of backward chaining

- Depth-first recursive proof search: space is linear in size of proof
- Incomplete due to infinite loops
  - can be fixed by applying breadth-first search
- Inefficient due to repeated sub-goals (both success and failure)
  - can be fixed using caching of previous results (extra space)
- Widely used for logic programming
Resolution strategies

- **Unit preference**
  - clauses with just one literal are preferred
  - **Unit resolution**
    - incomplete in general, complete for Horn KB

- **Set of support**
  - at least one of the clauses makes part from the set of support
  - complete if the remainder of the sentences are jointly satisfiable
  - Using the negated query as set-of-support

- **Input resolution**
  - at least one of the clauses makes part from the initial KB or the query
  - Complete for Horn, incomplete in the general case
  - **Linear resolution:** P and Q can be resolved if P is in the original KB or P is an ancestor of Q in the proof tree; complete

- **Subsumption:** all sentences subsumed by others in the KB are eliminated
- In each resolution step unification is applied!
  - **green**: goal
  - **red**: KB rules
Comparison Forward/Backward Chaining

- Advantage Forward Chaining:
  - Computing join is more efficient than nested-loop implicit in backward chaining

- Advantage Backward Chaining:
  - Avoiding the computation of the whole fixpoint
Summary

- Proof algorithms, are only semi-decidable i.e., might not terminate for non-entailed queries
- Propositionalization – instantiating quantifiers: slow
- Unification makes the instantiation step unnecessary
- Complete for definite clauses; semi-decidable
  - Forward-chaining
  - Backward-chaining
  - decidable for Datalog
- Strategies for reducing the search space of a resolution system required
OPTIMIZATIONS
Find components of trike!
Are we running low on any parts needed to build a trike?
What is total component and assembly cost to build trike at today's part prices?
Datalog Query that Does the Job

\[
\text{Comp}(\text{Part}, \text{Subpt}) :- \text{Assembly}(\text{Part}, \text{Subpt}, \text{Qty}). \\
\text{Comp}(\text{Part}, \text{Subpt}) :- \text{Assembly}(\text{Part}, \text{Part2}, \text{Qty}), \text{Comp}(\text{Part2}, \text{Subpt}).
\]
Example

For any instance of Assembly, we compute all Comp tuples by repeatedly applying two rules.

Actually: we can apply Rule 1 just once, then apply Rule 2 repeatedly.

- Rule1 ~ projection
- Rule2 ~ cross-product with equality join

\[
\text{Comp}(\text{Part}, \text{Subpt}) \leftarrow \text{Assembly}(\text{Part}, \text{Subpt}, \text{Qty}).
\]
\[
\text{Comp}(\text{Part}, \text{Subpt}) \leftarrow \text{Assembly}(\text{Part}, \text{Part2}, \text{Qty}), \ \text{Comp}(\text{Part2}, \text{Subpt}).
\]
### Assembly Instance

<table>
<thead>
<tr>
<th>Part</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>trike</td>
<td>3</td>
</tr>
<tr>
<td>trike</td>
<td>1</td>
</tr>
<tr>
<td>frame</td>
<td>1</td>
</tr>
<tr>
<td>frame</td>
<td>1</td>
</tr>
<tr>
<td>wheel</td>
<td>2</td>
</tr>
<tr>
<td>wheel</td>
<td>1</td>
</tr>
<tr>
<td>tire</td>
<td>1</td>
</tr>
<tr>
<td>tire</td>
<td>1</td>
</tr>
<tr>
<td>rim</td>
<td>1</td>
</tr>
<tr>
<td>tube</td>
<td>1</td>
</tr>
</tbody>
</table>

### Comp Tuples by Applying Rule 2 Once

<table>
<thead>
<tr>
<th>Part1</th>
<th>Part2</th>
</tr>
</thead>
<tbody>
<tr>
<td>trike</td>
<td>spoke</td>
</tr>
<tr>
<td>trike</td>
<td>tire</td>
</tr>
<tr>
<td>trike</td>
<td>seat</td>
</tr>
<tr>
<td>trike</td>
<td>pedal</td>
</tr>
<tr>
<td>wheel</td>
<td>rim</td>
</tr>
<tr>
<td>wheel</td>
<td>tube</td>
</tr>
<tr>
<td>trike</td>
<td>rim</td>
</tr>
<tr>
<td>trike</td>
<td>tube</td>
</tr>
</tbody>
</table>

### Comp Tuples by Applying Rule 2 Twice

<table>
<thead>
<tr>
<th>Part1</th>
<th>Part2</th>
</tr>
</thead>
<tbody>
<tr>
<td>trike</td>
<td>spoke</td>
</tr>
<tr>
<td>trike</td>
<td>tire</td>
</tr>
<tr>
<td>trike</td>
<td>seat</td>
</tr>
<tr>
<td>trike</td>
<td>pedal</td>
</tr>
<tr>
<td>wheel</td>
<td>rim</td>
</tr>
<tr>
<td>wheel</td>
<td>tube</td>
</tr>
</tbody>
</table>

**Comp**(Part, Subpt) :- Assembly(Part, Part2, Qty), Comp(Part2, Subpt).
Example

For any instance of Assembly, we can compute all Comp tuples by repeatedly applying the two rules. (Actually, we can apply Rule 1 just once, then apply Rule 2 repeatedly.)

<table>
<thead>
<tr>
<th>trike</th>
<th>spoke</th>
</tr>
</thead>
<tbody>
<tr>
<td>trike</td>
<td>tire</td>
</tr>
<tr>
<td>trike</td>
<td>seat</td>
</tr>
<tr>
<td>trike</td>
<td>pedal</td>
</tr>
<tr>
<td>wheel</td>
<td>rim</td>
</tr>
<tr>
<td>wheel</td>
<td>tube</td>
</tr>
</tbody>
</table>

Comp tuples got by applying Rule 2 once

<table>
<thead>
<tr>
<th>trike</th>
<th>spoke</th>
</tr>
</thead>
<tbody>
<tr>
<td>trike</td>
<td>tire</td>
</tr>
<tr>
<td>trike</td>
<td>seat</td>
</tr>
<tr>
<td>trike</td>
<td>pedal</td>
</tr>
<tr>
<td>wheel</td>
<td>rim</td>
</tr>
<tr>
<td>wheel</td>
<td>tube</td>
</tr>
<tr>
<td>trike</td>
<td>rim</td>
</tr>
<tr>
<td>trike</td>
<td>tube</td>
</tr>
</tbody>
</table>

Comp tuples got by applying Rule 2 twice
Evaluation of Datalog Programs

- Avoid Repeated inferences:
- Avoid Unnecessary inferences:
Query Optimization #1.
Avoid Repeated inferences:

When recursive rules are repeatedly applied in naïve way, we make same inferences in several iterations.
### Assembly instance

<table>
<thead>
<tr>
<th>Part</th>
<th>Subpart</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>trike</td>
<td>wheel</td>
<td>3</td>
</tr>
<tr>
<td>trike</td>
<td>frame</td>
<td>1</td>
</tr>
<tr>
<td>frame</td>
<td>seat</td>
<td>1</td>
</tr>
<tr>
<td>frame</td>
<td>pedal</td>
<td>1</td>
</tr>
<tr>
<td>wheel</td>
<td>spoke</td>
<td>2</td>
</tr>
<tr>
<td>wheel</td>
<td>tire</td>
<td>1</td>
</tr>
<tr>
<td>tire</td>
<td>rim</td>
<td>1</td>
</tr>
<tr>
<td>tire</td>
<td>tube</td>
<td>1</td>
</tr>
</tbody>
</table>

### Comp tuples by applying Rule 2 once

<table>
<thead>
<tr>
<th>Part</th>
<th>Subpart</th>
</tr>
</thead>
<tbody>
<tr>
<td>trike</td>
<td>spoke</td>
</tr>
<tr>
<td>trike</td>
<td>tire</td>
</tr>
<tr>
<td>trike</td>
<td>seat</td>
</tr>
<tr>
<td>trike</td>
<td>pedal</td>
</tr>
<tr>
<td>wheel</td>
<td>rim</td>
</tr>
<tr>
<td>wheel</td>
<td>tube</td>
</tr>
</tbody>
</table>

### Comp tuples by applying Rule 2 twice

<table>
<thead>
<tr>
<th>Part</th>
<th>Subpart</th>
</tr>
</thead>
<tbody>
<tr>
<td>trike</td>
<td>spoke</td>
</tr>
<tr>
<td>trike</td>
<td>tire</td>
</tr>
<tr>
<td>trike</td>
<td>seat</td>
</tr>
<tr>
<td>trike</td>
<td>pedal</td>
</tr>
<tr>
<td>wheel</td>
<td>rim</td>
</tr>
<tr>
<td>wheel</td>
<td>tube</td>
</tr>
</tbody>
</table>

\[
\text{Comp(Part, Subpt) :- Assembly(Part, Part2, Qty), Comp(Part2, Subpt).}
\]
Avoiding Repeated Inferences

- **Semi-naive Fixpoint Evaluation:**
  - Ensure that when rule is applied, at least one of body facts used was generated in most recent iteration.
  - Such new inference could not have been carried out in earlier iterations.
Avoiding Repeated Inferences

- **Idea:** For each recursive table $P$, use table $\text{delta}_P$ to store $P$ tuples generated in previous iteration.
  - 1. Rewrite program to use delta tables
  - 2. Update delta tables between iterations.

```
Comp(Part, Subpt) :- Assembly(Part, Part2, Qty), Comp(Part2, Subpt).

Comp(Part, Subpt) :- Assembly(Part, Part2, Qty), delta_Comp(Part2, Subpt).
```
Query Optimization #2.
**Unnecessary inferences:**

- If we just want to find components of a particular part, say *wheel*, then first computing general fixpoint of Comp program and then at end selecting tuples with *wheel* in the first column is wasteful.

- This would compute many irrelevant facts.
Avoiding Unnecessary Inferences

SameLev(S1,S2) :- Assembly(P1,S1,Q1),
                  Assembly(P1,S2,Q2).

SameLev(S1,S2) :- Assembly(P1,S1,Q1),
                  SameLev(P1,P2),
                  Assembly(P2,S2,Q2).

```
  trike
   3
     1
    / 
   /   
wheel  frame
   
    2  1
   /    /
spoke tire seat pedal
   
    1  1
   /    /
rim    tube
```

1  1

1  1

1  1

1  1
Avoiding Unnecessary Inferences

Tuple \((S_1, S_2)\) in \(\text{SameLev}\) if there is path up from \(S_1\) to some node and down to \(S_2\) with same number of up and down edges.

\[
\text{SameLev}(S_1, S_2) \leftarrow \text{Assembly}(P_1, S_1, Q_1), \text{Assembly}(P_1, S_2, Q_2).
\]

\[
\text{SameLev}(S_1, S_2) \leftarrow \text{Assembly}(P_1, S_1, Q_1), \text{SameLev}(P_1, P_2), \text{Assembly}(P_2, S_2, Q_2).
\]
Avoiding Unnecessary Inferences

- Want all SameLev tuples with \texttt{spoke} in first column.
- \textbf{Intuition:} “Push” this selection into fixpoint computation.
- How do that?

\[ \text{SameLev}(S1,S2) :- \]
\[ \text{Assembly}(P1,S1,Q1), \]
\[ \text{SameLev}(P1,P2), \text{Assembly}(P2,S2,Q2). \]

\[ \text{SameLev}(\text{spoke},S2) :- \]
\[ \text{Assembly}(P1,\text{spoke},Q1), \]
\[ \text{SameLev}(P1?=\text{spoke}?,P2), \]
\[ \text{Assembly}(P2,S2,Q2). \]
Avoiding Unnecessary Inferences

- **Intuition:** “Push” this selection with *spoke* into fixpoint computation.

\[
\text{SameLev}(S1,S2) :- \\
\text{Assembly}(P1,S1,Q1), \\
\text{SameLev}(P1,P2), \text{Assembly}(P2,S2,Q2).
\]

\[
\text{SameLev}(\text{spoke},S2) :- \\
\text{Assembly}(P1,\text{spoke},Q1), \text{SameLev}(P1,P'), \\
\text{Assembly}(P2,S2,Q2).
\]

\[
\text{SameLev}(\text{spoke},\text{seat}) :- \\
\text{Assembly}(\text{wheel},\text{spoke},2), \\
\text{SameLev}(\text{wheel},\text{frame}), \\
\text{Assembly}(\text{frame},\text{seat},1).
\]

- Other SameLev tuples are needed to compute all such tuples with *spoke*, e.g. *wheel*.
“Magic Sets” Idea

1. Define “filter” table that computes all relevant values

2. Restrict computation of SameLev to infer only tuples with relevant value in first column.
- **Relevant** values: contains all tuples \( m \) for which we have to compute all same-level tuples with \( m \) in first column to answer query.

- Put differently, **relevant** values are all Same-Level tuples whose first field contains value on path from spoke up to root.

- We call it Magic-SameLevel (Magic-SL)
### “Magic Sets” in Example

- **Idea:** Define “filter” table that computes all relevant values: Collect all parents of `spoke`.

```
Magic_SL(P1) :- Magic_SL(S1), Assembly(P1,S1,Q1).
Magic_SL(spoke) :- .
```

Make Magic table as Magic-SameLevel.
"Magic Sets" Idea

- **Idea:** Use "filter" table to restrict the computation of SameLev.

```
Magic_SL(P1) :- Magic_SL(S1), Assembly(P1,S1,Q1).
Magic(spoke).

SameLev(S1,S2) :- Magic_SL(S1), Assembly(P1,S1,Q1),
Assembly(P1,S2,Q2).

SameLev(S1,S2) :- Magic_SL(S1), Assembly(P1,S1,Q1),
SameLev(P1,P2), Assembly(P2,S2,Q2).
```
**“Magic Sets” Idea**

- **Idea:** Define “filter” table that computes all relevant values, and restrict the computation of SameLev correspondingly.

\[
\text{Magic}_{\text{SL}}(P_1) :\text{ Magic}_{\text{SL}}(S_1), \text{ Assembly}(P_1,S_1,Q_1), \text{ Magic}\text{(spoke)}.
\]

\[
\text{SameLev}(S_1,S_2) :\text{ Magic}_{\text{SL}}(S_1), \text{ Assembly}(P_1,S_1,Q_1), \text{ Assembly}(P_1,S_2,Q_2).
\]

\[
\text{SameLev}(S_1,S_2) :\text{ Magic}_{\text{SL}}(S_1), \text{ Assembly}(P_1,S_1,Q_1), \text{ SameLev}(P_1,P_2), \text{ Assembly}(P_2,S_2,Q_2).
\]
1. Generate an “adorned” program
   Program is rewritten to make pattern of bound and free arguments in query explicit

2. Add magic filters of form “Magic_P”
   for each rule in adorned program add a Magic condition to body that acts as filter on set of tuples generated (predicate P to restrict these rules)

3. Define new rules to define filter tables
   Define new rules to define filter tables of form Magic_P
Step 1: Generating Adorned Rules

Adorned program for query pattern SameLev\(^{bf}\), assuming left-to-right order of rule evaluation:

\[
\text{SameLev}^{bf}\ (S1, S2) \leftarrow \text{Assembly}(P1, S1, Q1), \text{Assembly}(P1, S2, Q2).
\]

\[
\text{SameLev}^{bf}\ (S1, S2) \leftarrow \text{Assembly}(P1, S1, Q1),
\]

\[
\text{SameLev}^{bf}\ (P1, P2), \text{Assembly}(P2, S2, Q2).
\]

- Argument of (a given body occurrence of) SameLev is:
  - \(b\) if it appears to the left in body,
  - or if it is a \(b\) argument of head of rule,
  - Otherwise it is free.
- Assembly not adorned because explicitly stored table.
Step 2: Add Magic Filters

For every rule in adorned program add a ‘magic filter’ predicate

\[
\text{SameLev}^{bf} (S1,S2) :- \text{Magic\_SL} (S1), \text{Assembly}(P1,S1,Q1), \text{Assembly}(P1,S2,Q2).
\]

\[
\text{SameLev}^{bf} (S1,S2) :- \text{Magic\_SL} (S1), \text{Assembly}(P1,S1,Q1), \text{SameLev}^{bf} (P1,P2), \text{Assembly}(P2,S2,Q2).
\]

Filter predicate: copy of head of rule, Magic prefix, and delete free variable
Rule for Magic_P is generated from each occurrence of recursive P in body of rule:

- Delete everything to right of P
- Add prefix “Magic” and delete free columns of P
- Move P, with these changes, into head of rule
Step 3: Defining Magic Table

Rule for Magic_P is generated from each occurrence O of recursive P in body of rule:

- Delete everything to right of P

\[
\text{SameLev}^{bf}(S1,S2) \leftarrow \text{Magic}_SL(S1), \text{Assembly}(P1,S1,Q1), \\
\text{SameLev}^{bf}(P1,P2), \text{Assembly}(P2,S2,Q2).
\]

- Add prefix “Magic” and delete free columns of P

\[
\text{Magic-SameLev}^{bf}(S1,S2) \leftarrow \text{Magic}_SL(S1), \text{Assembly}(P1,S1,Q1), \\
\text{Magic-SameLev}^{bf}(P1__)\
\]

- Move P, with these changes, into head of rule

\[
\text{Magic}_SL(P1) \leftarrow \text{Magic}_SL(S1), \text{Assembly}(P1,S1,Q1).
\]
Step 3: Defining Magic Tables

Rule for Magic_P is generated from each occurrence of P in body of such rule:

\[
\text{SameLev}^{bf} (S1,S2) :\text{ Magic}_S(L(S1), \text{ Assembly}(P1,S1,Q1), \\
\text{ SameLev}^{bf} (P1,P2), \text{ Assembly}(P2,S2,Q2)).
\]

\[
\text{Magic}_S(L(P1) :\text{ Magic}_S(L(S1), \text{ Assembly}(P1,S1,Q1)).
\]
“Magic Sets” Idea

- Define “filter” table that computes all relevant values:
- Restrict computation of SameLev

```prolog
Magic_SL(P1) :- Magic_SL(S1), Assembly(P1,S1,Q1).
Magic(spoke).

SameLev(S1,S2) :- Magic_SL(S1), Assembly(P1,S1,Q1),
    Assembly(P1,S2,Q2).
SameLev(S1,S2) :- Magic_SL(S1), Assembly(P1,S1,Q1),
    SameLev(P1,P2), Assembly(P2,S2,Q2).
```
DYNAMIC FILTERING
Dynamic Filtering (Kifer)

\[ q(f(a), a), q(a,c), q(f(c),c), r(a, c), r(c, d), r(e,c) \]

\[ p(X, Y) \leftarrow q(f(X), X) \land r(X, Y) \]

\[ \leftarrow p(X, c) \]
Dynamic Filtering: Problem Negation

\[ p(X, Y) \leftarrow q(f(X), X) \land r(X, Y) \]

\[ \leftarrow q(X, c) \land \neg p(X, c) \]

\[ a, a \]

\[ X, c \]

\[ f(c), c \]

\[ f(a), a \]

\[ q(f(a), a) \]

\[ q(a,c) \]

\[ q(f(c), c) \]

\[ r(a, c) \]

\[ r(c,d) \]

\[ r(e,f) \]
Dynamic Filtering: Stratification

\[ p(X, Y) \leftarrow q(f(X), X) \land r(X, Y) \]

Level 2

\[ f(c) \leftarrow q(X, c) \land \text{not } p(X, c) \]

Level 1

\[ X, c \]
\[ a, c \]
\[ f(a), a \]
\[ q(f(a), a) \]
\[ q(a, c) \]
\[ q(f(c), c) \]

Level 0

\[ a \]
\[ f(c) \]
\[ a, c \]
\[ f(c), c \]
\[ a, a \]
\[ r(a, c) \]
\[ r(c, d) \]
\[ r(e, f) \]

X, c
ALTERNATING FIXPOINT
What about the following?

person(somebody).

woman(X) <- person(X) and not man(X).
man(X) <- person(X) and not woman(X).

If there exists some $n$:

$T^{n+1}(I) = T^{n-1}(I)$ and

$T^{n+2}(I) = T^{n}(I)$

Then the true facts are defined by the smaller sets of $T^{n-1}(I)$ and $T^{n}(I)$

Unknown facts are given by set difference.
Alternating Fixed Point: Example

person(sb).
woman(X) <- person(X) and not man(X).
man(X) <- person(X) and not woman(X).

T(I={person{sb}}) = {woman(sb), man(sb), person(sb)}
T(T(I)) = {person(sb)}
T(T(T(I)))) = {woman(sb), man(sb), person(sb)}
T(T(T(T(I)))) = {person(sb)}

True Facts = {person(sb)}
Unknown Facts = {woman(sb), man(sb)}

Very expensive 😞