Herbrand-Universe/Herbrand-Base (1)

- Mentioned several times on the previous slides: Set of all facts **constructible** from the "syntactic material at hand"
- Intuitively reasonable idea, but imprecise formulation:
  - What constitutes "the syntactic material"?
  - What does it mean "to construct"?
- Fortunately, there are precise definitions for the terms available in mathematical logic:
  - Given: A deductive database $D = (R, F)$
  - Set of all relation symbols (with arities) occurring in $D$:
    - **Signature** $\text{Rel}_D$ of $D$
  - Set of all constants occurring in $D$:
    - **Herbrand-Universe** $U_D$ of $D$
  - Set of all ground literals composed from relation symbols in $\text{Rel}_D$ and constants in $U_D$, respecting the arities of the relation symbols:
    - **Herbrand-Base** $H_D$ of $D$
- Origin of this notion: **Jacques Herbrand**, French logician who created a „School of Semantics“ in logic based solely on syntactic concepts
Herbrand-Universe/Herbrand-Base (2)

Corresponding Herbrand-Universe: \( U_D = \{a, b, c\} \)

Herbrand-Base \( H_D \) for \( Rel_D \) and \( U_D \):

- \( s(a,a) \)
- \( r(a) \)
- \( q(a) \)
- \( p(a) \)
- \( s(a,b) \)
- \( r(b) \)
- \( q(b) \)
- \( p(b) \)
- \( s(a,c) \)
- \( r(c) \)
- \( q(c) \)
- \( p(c) \)
- \( s(b,a) \)
- \( s(b,b) \)
- \( s(b,c) \)
- \( s(c,a) \)
- \( s(c,b) \)
- \( s(c,c) \)

Corresponding signature: \( Rel_D = \{r/1, s/2, q/1, p/1\} \)
Jacques Herbrand

Born: 12.2.1908 in Paris
Died: 27.7.1931 near La Bérarde, Isère

• 1925 aged 17 accepted for École Normale Supérieure as top student of his year
• 1929 aged 21 doctorate in mathematics, afterwards military service
• 1931 Rockefeller grant for studying with leading mathematicians of the time in Germany:
  John v. Neumann (Berlin), Emil Artin (Hamburg), Emmy Noether (Göttingen)
• July 1931: Killed in a mountaineering accident in the French Alps when returning from Germany
Semantics of Unstratifiable Rules

• During the 1980s, there has been an intense scientific debate about a „proper“ way of dealing with the semantics of unstratifiable deductive databases.

• One „camp“ in this debate, following an approach that has been called the „stable model semantics“ since, proposed to determine so-called „possible worlds“, i.e., two-valued assignments of truth values to facts in the Herbrand base, which are in a certain sense consistent with the derivations possible with the rules in $R$ if the resp. truth values in the „world“ under consideration are known before. Databases with more than one such „possible worlds“ were discarded as meaningless due to this ambiguity.

• We will follow a different trend of research, called the „well-founded semantics“. Here, a three-valued logic is employed (similar to the way we discussed up till now), such that at the end decisions about undefined cases are made on a fact-by-fact basis.

• In turn, there have been various approaches to define (and compute) truth value assignments in line with the „well-founded“ approach. In this chapter, we will follow the conditional fixpoint method (introduced by F. Bry in 1989) and the alternating fixpoint method (introduced by van Gelder in 1988).
General attitude

If there are any facts for which an unstratifiable set of rules cannot determine a definite truth value . . .

. . . then do not assign any semantics to the entire DB.

"stable model" semantics
(Gelfond/Lifschitz, 1988)

. . . then make use of 3-valued logic and assign an undefined truth value to the "problematic" facts only.

"well-founded" semantics
(van Gelder/Ross/Schlipf, 1988)
Implementing Well-Founded Models

How to compute the well-founded semantics?

"alternating fixpoint computation" (van Gelder, 1989)

"conditional fixpoint computation" (Bry, 1989; Dung/Kanchansut, 1989)

“doubled program" 
Kemp/Srivasta/Stuckey, 1995

"program remainder" 
Brass/Zukowski/Freitag, 1997
CFP-Approach in the Example

1st phase: Fixpoint iteration without evaluating negative literals

p(X) ← s(X,Y), not p(Y).

s(1,1).
s(2,3).
s(3,4).

p(1) ← not p(1).
p(2) ← not p(3).
p(3) ← not p(4).

2nd phase: Iterated application of two reduction steps until no further change occurs

"Conditional facts"

true
false
undefined
Second motivating example, this time without recursion

1st phase: Fixpoint iteration without evaluating negative literals

Transfer of condition parts of "inserted" conditional facts
CFP-Method: Expansion (2)

- Partial evaluation of positive literals: **without** transferring conditions

\[ q(X) \leftarrow s(X), \text{not} \ t(X). \]
\[ s(a). \]
\[ q(a) \leftarrow \text{not} \ t(a). \]

- Partial evaluation of positive literals: **with** transferring conditions

\[ p(X) \leftarrow q(X), \text{not} \ r(X). \]
\[ q(a) \leftarrow \text{not} \ t(a). \]
\[ p(a) \leftarrow \text{not} \ t(a), \text{not} \ r(a). \]
CFP-Method: Reduction (1)

2\textsuperscript{nd} phase: Reduction of results from the 1\textsuperscript{st} phase by eliminating definitely true literals

\[ F_2 \]

\[
\begin{align*}
q(a) & \leftarrow \text{not } t(a). \\
q(b) & \leftarrow \text{not } t(b). \\
r(a) & \leftarrow \text{not } w(a). \\
r(b) & \leftarrow \text{not } w(b). \\
p(a) & \leftarrow \text{not } t(a), \text{not } r(a). \\
p(b) & \leftarrow \text{not } t(b), \text{not } r(b). \\
s(a). \\
s(b). \\
t(c). \\
w(b). \\
w(c).
\end{align*}
\]

Facts in the Herbrand-Base which cannot become true ever (because of \( F_2 \)):
CFP-Method: Reduction (2)

2\textsuperscript{nd} phase: Further reduction by eliminating "unproductive" conditional facts

p(X) ← q(X), \textbf{not} r(X).
q(X) ← s(X), \textbf{not} t(X).
r(X) ← s(X), \textbf{not} w(X).

s(a).
s(b). w(b).
t(c). w(c).

\[2^\text{nd phase:} \quad \text{Further reduction by eliminating "unproductive" conditional facts}\]

\[F'\]

q(a).
q(b).
r(a).
r(b) ← \textbf{not} w(b).
p(a) ← \textbf{not} r(a).
p(b) ← \textbf{not} r(b).
s(a).
s(b).
w(b).
t(c). w(c).

\[F''\]

q(a).
q(b).
r(a).
p(b) ← \textbf{not} r(b).
s(a).
s(b).
w(b).
t(c). w(c).

p(a) und r(b) have now been identified as false, too!
CFP-Method: Reduction (3)

\[
\begin{align*}
p(X) & \leftarrow q(X), \ \text{not} \ r(X). \\
q(X) & \leftarrow s(X), \ \text{not} \ t(X). \\
r(X) & \leftarrow s(X), \ \text{not} \ w(X).
\end{align*}
\]

"Fixpoint" of the reduction phase = $F_{\text{pos}^*}$

Complement of $F_{\text{pos}^*} = F_{\text{neg}^*}$

No facts with *undefined* truth value!
Optimization of Expansion

\[ p(X) \leftarrow s(X,Y), \ p(Y), \ not \ q(Y). \]
\[ q(X) \leftarrow s(X,Y), \ not \ q(X). \]

\[ s(1,2). \]
\[ s(2,3). \]
\[ s(3,4). \]

Danger of (potentially exponential) **growth of length** of bodies of conditional facts in phase 1!

\[ p(1) \leftarrow p(2), \ not \ q(2). \]
\[ p(2) \leftarrow p(3), \ not \ q(3). \]
\[ p(3) \leftarrow p(4), \ not \ q(4). \]
\[ q(1) \leftarrow \ not \ q(1). \]
\[ q(2) \leftarrow \ not \ q(2). \]
\[ q(3) \leftarrow \ not \ q(3). \]

Delay evaluation of certain "critical" positive literals, too!

Brass/Zukowski/Freitag, 1997
The CFP-semantics is based on different derivation operators, too, for which least fixpoints are computed (via iteration).

However, the CFP-method is much more complex than simple fixpoint iteration or even as iterated FPI – nevertheless (or better: because of this complexity) a formal definition of these operators and the overall principle is very important, though tedious.

CFP operates on conditional facts, i.e., on variable-free rules, the body literals of which are all negated, or the body of which consists of the single literal true.

CFP works in two phases:

- In an expansion phase all conditional facts are determined which are derivable from the set of base facts by applying all rules in the rule set R. Negation is not yet „touched“ in this phase, but delayed to the second phase.

- Then this intermediate result is reduced in a reduction phase by eliminating literals in conditional facts or entire conditional facts from the fact set at hand.

- Finally, a truth value assignment is performed, assigning true, undefined, or false to each of the facts in the Herbrand-base of the DB based on the result of the two evaluation phases.
CFP-Method: Principle (2)

- Before starting the 1st phase (expansion), all base facts of the resp. DB are transformed into conditional facts, the body of which consists just of true.

- Both phases can be viewed as separate fixpoint iterations with different operators.
The "preparation step", in which all base facts are transformed into conditional facts by adding the "artificial" body $\leftarrow \text{true}$ before starting expansion, is not really necessary.

However, doing so simplifies formalization of the method significantly as we do not have to distinguish between conditional and unconditional facts any more,
How does CFP deal with stratifiable databases (with recursion)?

CFP does not need any layers/strata, but applies the expansion phase to all rules simultaneously, however without touching negation (true-bodies are omitted here):

Will be explained immediately!
CFP-Semantics for Stratifiable Databases (2)

\[
\begin{align*}
s(X) & \leftarrow t(X,Y), \text{not} p(X,Y). \\
p(X,Y) & \leftarrow q(X,Y), \text{not} q(Y,X). \\
p(X,Y) & \leftarrow q(X,Z), p(Z,Y). \\
q(X,Y) & \leftarrow r(X,Y,Z).
\end{align*}
\]

\[
\begin{align*}
s(1) & \leftarrow \text{not} p(1,4). \\
p(1,2) & \leftarrow \text{not} q(2,1). \\
p(3,2) & \leftarrow \text{not} q(2,3). \\
p(2,2) & \leftarrow \text{not} q(2,2). \\
q(1,2) & \leftarrow \text{not} p(1,4). \\
q(2,3) & \leftarrow \text{not} q(2,1). \\
q(3,1) & \leftarrow \text{not} q(2,3).
\end{align*}
\]

\[
\begin{align*}
r(1,2,3). \\
r(2,3,4). \\
r(3,1,2). \\
t(1,4). \\
p(1,2). \\
p(3,2). \\
p(2,2). \\
q(1,2). \\
q(2,3). \\
q(3,1).
\end{align*}
\]

Reduction eliminates all negative literals
Semantics for Different Classes of DDB

- In both stratifiable examples (recursive and non-recursive) the CFP-method always delivered the same result as iterated fixpoint computation (IFP) based on a stratification, even though in a different manner.

- This observation holds for each database!
  - CFP- and IFP-semantics coincide for stratifiable database.
  - CFP extends IFP conservatively for unstratifiable databases.
  - This is true in particular for (semi-)positive DBs, too, where all three methods deliver the same results as simple fixpoint iteration.
  - Proofs for these claims can be found in the original research papers introducing the resp. methods (which are not at all trivial!).

![Diagram showing relationships between different classes of databases]

- Unstratifiable
- Iterated FP-method
- "Simple" FPI
- Stratifiable
- Semi-positive
- CFP-method
Which Databases are "reasonable"?

A database can be regarded as meaningful without any doubt, if (even in presence of unstratifiable rules) no fact is assigned the truth value undefined. Certainly not meaningful are only such DBs, for which all facts have to be assigned undefined.
A More "Realistic" Unstratifiable Example (1)

- **Finally**: Example of a rule set for which unstratifiability arises quite "naturally" and which might appear in many real applications.

- **Intuitive semantics**: 
  Quality control of complex objects (Air planes, computers etc.)
  - `is_ok(T)`: Part T is in an acceptable state
  - `has_been_tested(T)`: Part T has been tested explicitly
  - `component_of(X,Y)`: Part Y is a component of part X
  - `is_complex_part(T)`: Part T is composed from components
  - `has_bad_component(T)`: Part T has at least one component, which is not ok.

\[
\begin{align*}
\text{is}_\text{ok}(T) & \leftarrow \text{has}_\text{been}_\text{tested}(T) . \\
\text{is}_\text{ok}(T) & \leftarrow \text{is}_\text{complex}_\text{part}(T), \text{not} \ \text{has}_\text{bad}_\text{component}(T) . \\
\text{has}_\text{bad}_\text{component}(T) & \leftarrow \text{component}_\text{of}(T, T') , \text{not} \ \text{is}_\text{ok}(T') . \\
\text{is}_\text{complex}_\text{part}(T) & \leftarrow \text{component}_\text{of}(T, _). 
\end{align*}
\]
is_ok(T) ← has Been tested(T).

is_ok(T) ←
  is_complex_part(T), not has_bad_component(T).

has_bad_component(T) ←
  component_of(T, T'), not is_ok(T').

is_complex_part(T) ←
  component_of(T, _).

Component_of:

- \( p_1 \)
- \( p_2 \)
- \( p_3 \)
- \( p_4 \)
- \( p_5 \)
- \( p_6 \)
- \( p_7 \)
- \( p_8 \)
- \( p_9 \)
- \( p_{10} \)

- \( p_1 \) tested (thus, ok)
- \( p_4 \) bad component
- \( p_2 \) ok recursively
Formalization of CFP – Expansion Phase

- In phase 1 (expansion), rules are applied to conditional facts. When doing so,
  - only positive body literals are replaced, however, and
  - in the bodies of the substituted conditional facts are transferred.

- First, we define once again an operator formalizing the application of individual rules:

\[ T'_{\text{cond}}(R_i) = \{ \sigma \mid \forall 1 \leq j \leq n : (B_j \leftarrow D_j) \sigma \in CF \text{ holds} \} \]

- Name of this operator: \( T'_{\text{cond}} \) – Attention! The following definition is not yet final! It still contains a serious error in reasoning (thus, a ' with the T)!

- Transferred conditions
- Not yet "evaluated" negative body literals
- Conjunctions of literals
In the "Normal Case"
1st Problem: Redundant true-Literals

If conditional facts with "artificial" bodies are used for replacing positive literals, logically redundant true-literals will occur which will have to be absorbed by the "real" literals, if any, due to a simple law of propositional logic:

true and A ≡ A
2nd Problem: Duplicates in Conditions

In our running example, this second problem does not occur, but a fitting example can be easily constructed:

\[
p(X) \leftarrow q(X), v(X), \text{not } r(X).
\]

According to the Law of Idempotence in logic, duplicates can be eliminated – and this should be done, because the multiplicity of literals may grow and grow due to recursion:

\[
A \land A \equiv A
\]
"Pruning" is a notion from gardening, where it means cutting of shrubs or other plants.

We will introduce an operator 'prune', which cuts a given conjunction of literals in such a way that

- no duplicates do occur anymore and
- each occurrence of true is eliminated (at most until at least one literal remains).

For 'prune' we don’t present any formal definition – its meaning should be evident.

Thus, we are able to present an improved version of the operator for single rules applied to conditional facts as follows:

\[
T_{\text{cond}}[R_1](CF) \overset{\text{def}}{=} \{ A \leftarrow \text{prune}[(D_1, \ldots, D_n, \neg C_1, \ldots, \neg C_m) \sigma] \mid \\
\sigma \text{ is a consistent variable substitution, such that} \\
\forall 1 \leq j \leq n : (B_j \leftarrow D_j) \sigma \in CF \text{ holds} \}
\]
Expansion Phase: Summary

- The **expansion phase** is organized overall like a quite "normal" fixpoint iteration similar to the way we organized it for positive rules.

- All rules are applied to a set of conditional facts using a **collective** $T_{\text{cond}}$:

  $$T_{\text{cond}}[R](CF) = \text{def} \bigcup_{R_i \in R} T_{\text{cond}}[R_i](CF)$$

- Iteration can be done if we "carry along" all results previously obtained. For doing so, a **cumulative** $T_{\text{cond}}^*$-operator is used:

  $$T_{\text{cond}}^*[R](CF) = \text{def} T_{\text{cond}}[R](CF) \cup CF$$

- For a given transformed set of base facts $F_{\text{cond}}$, the result of the first phase of the CPF-approach is again the **least fixpoint** of $T_{\text{cond}}^*[R]$ containing all of $F_{\text{cond}}$:

  $$F_{\text{cond}}^* = \text{def} \lim_{i \to \infty} T_{\text{cond}}^*[R]^i(F_{\text{cond}})$$
The reduction phase requires two basic operators:
- One operator for eliminating negative literals from conditions, and
- another operator for eliminating entire conditional facts from the input set.

Negative literals can be regarded as satisfied over a given set of conditional facts, if no head of any conditional fact corresponds to the positive part of that literal.

As a useful tool for formalizing this step (and others to come), we introduce an operator for extracting all head literals from a set of conditional facts $CF$:

$$\text{heads}(CF) = \{ A \mid (A \leftarrow B) \in CF \}$$

If for a negative literal not $L$ the positive part $L$ is not in heads $(CF)$, then the resp. negative literal can be replaced by true and subsequently eliminated by "prune".

An operator performing this step will be called $\text{Red}_{\text{true}}$ (for "reduce").
Formalization of the Reduction Phase (2)

- Red\textsubscript{true} can be formalized as follows:

\[
\text{Red}\textsubscript{true}(CF) = \{ A \leftarrow \text{prune} [C_{red}^1, \ldots, C_{red}^n] \mid (A \leftarrow \text{not } C_1, \ldots, \text{not } C_n) \in CF \}
\]

with

\[
C_{red}^i = \begin{cases} 
\text{true}, & \text{if } C_i \not\in \text{heads}(CF) \\
\text{not } C_i, & \text{else}
\end{cases}
\]

for \(1 \leq i \leq n\)

- For better understanding of this (non-trivial) construct, consider the corresponding natural language version of this definition again:

If for a negative literal \text{not } L the positive part L is \text{not} in \text{heads} (CF), then this negative literal can be replaced by \text{true} and subsequently eliminated by "prune".
Example for the First Reduction Operator

$$\text{heads(CF)} = \{q(a), q(b), r(a), r(b), p(a), p(b), s(a), s(b), t(c), w(b), w(c)\}$$
If for a negative literal not L it is already clear that the positive part L is satisfied (because there is an "unconditional fact" L ← true), we can replace it by false.

Conditional facts the body of which contains false can never evaluate to true and thus should be eliminated from the current set of conditional facts.

These considerations motivate a second reduction operator Red\textsubscript{false}:

\[
\text{Red}_{\text{false}}(CF) = _{\text{def}} CF - CF_{\text{false}}
\]

where \( CF_{\text{false}} = _{\text{def}} \{ A ← \text{not } C_1, \ldots, \text{not } C_n \mid
(A ← \text{not } C_1, \ldots, \text{not } C_n) \in CF \text{ and } \exists 1 \leq i \leq n: C_i ← \text{true} \in CF \}\)

Both reduction operators have to be applied sequentially:
Example for the Second Reduction Operator

Conditional facts to be eliminated:

- \(r(b) \leftarrow \text{not } w(b).
- \(p(a) \leftarrow \text{not } r(a).

Will be eliminated during the next iteration by the first operator.

```
<table>
<thead>
<tr>
<th>CF</th>
<th>CF_false</th>
</tr>
</thead>
<tbody>
<tr>
<td>q(a) \leftarrow true.</td>
<td>r(b) \leftarrow not w(b).</td>
</tr>
<tr>
<td>q(b) \leftarrow true.</td>
<td>p(a) \leftarrow not r(a).</td>
</tr>
<tr>
<td>r(a) \leftarrow true.</td>
<td></td>
</tr>
<tr>
<td>r(b) \leftarrow not w(b).</td>
<td></td>
</tr>
<tr>
<td>p(a) \leftarrow not r(a).</td>
<td></td>
</tr>
<tr>
<td>p(b) \leftarrow not r(b).</td>
<td></td>
</tr>
<tr>
<td>s(a) \leftarrow true.</td>
<td></td>
</tr>
<tr>
<td>s(b) \leftarrow true.</td>
<td></td>
</tr>
<tr>
<td>w(b) \leftarrow true.</td>
<td></td>
</tr>
<tr>
<td>w(c) \leftarrow true.</td>
<td></td>
</tr>
<tr>
<td>t(c) \leftarrow true.</td>
<td></td>
</tr>
</tbody>
</table>
```
Formalization of the Entire CFP-Semantics

- In the reduction phase all reduction steps will be applied in alternation as long as the set of facts under reduction does not shrink anymore: Least fixpoint of Red

\[ F_{\text{red}}^* = \text{def} \lim_{i \to \infty} \text{Red}^i (F_{\text{cond}}^*) \]

Reduction starts from the result of the expansion phase

- The semantics of a deductive database \( D = (R,F) \) is based on three truth values true, false, and undefined, determined from the final result of the "double iteration" as follows:

\[ F_{\text{pos}}^* = \text{def} \{ A \mid A \leftarrow \text{true} \in F_{\text{red}}^* \} \]
\[ F_{\text{undef}}^* = \text{def} \text{heads}(F_{\text{red}}^*) - F_{\text{pos}}^* \]
\[ F_{\text{neg}}^* = \text{def} H_D - \text{heads}(F_{\text{red}}^*) \]

Heads of all finally "unconditional" facts
Heads of all "surviving" conditional facts
Complement of both sets
Important Original Sources

- **van Emden/Kowalski**: "The semantics of predicate logic as a programming language", Journal of the ACM 23(4):733-742, 1976
- **Gelfond/Lifschitz**: "The stable model semantics for logic programming", ICLP 1988: 1070-1080
- **Bry**: "Logic programming as Constructivism: A Formalization and its Application in Databases", PODS 1989:34-50
- **Dung/Kanchansut**: "A Fixpoint Approach to Declarative Semantics of Logic Programs", NACL 1989:604-625
Deficiencies of CFP

- “conditional”-Facts: problematic data structure
  - Representation within relational context?
    \[
    p(1) \leftarrow \textbf{not} \ p(2).
    \]
  - Indexing conditional facts?
    \[
    p(3) \leftarrow \textbf{not} \ q(4), \textbf{not} \ p(2).
    \]
  - Exponential growth of conditions possible during the expansion phase
    \[
    p(1) \leftarrow \textbf{not} \ q(1),
    p(1) \leftarrow \textbf{not} \ q(2)
    p(1) \leftarrow \textbf{not} \ q(1), \textbf{not} \ q(2)
    \ldots
    \]

- Instance-oriented rule execution not suited for the DB context

\[
\begin{array}{c}
p(2) \leftarrow \textbf{not} \ p(3).
p(3) \leftarrow \textbf{not} \ p(4).
\end{array}
\Rightarrow
\begin{array}{c}
p(2) \leftarrow \textbf{not} \ p(3).
p(3) \leftarrow \textbf{not} \ p(4).
\end{array}
\]
Alternating Fixpoint (1)

The two negative literals \( \text{not } Q(A) \) and \( \text{not } Q(B) \) have to be interpreted correctly.

There are two possible views for \( \llbracket R \rrbracket \):

- **View of the pessimist:**
  All negative literals are assumed to be **false** – perform a fixpoint computation under this assumption.

- **View of the optimist:**
  All negative literals are assumed to be **true** – perform a fixpoint computation under this assumption.
Thus, \( Q(B) \) is always derivable under every interpretation of negation.

Consequently, \( \text{not } Q(B) \) ought to be considered to be false even for the optimistic view.

Additional condition for the fixpoint computation of the optimist!
Alternating Fixpoint (3)

1st optimistic FPI

\[
\begin{align*}
q(a) & \leftarrow r(a). \\
q(b) & \leftarrow r(b). \\
p(a) & \leftarrow s(a,a), \not q(a). \\
p(a) & \leftarrow s(a,b), \not q(b). \\
p(b) & \leftarrow s(b,a), \not q(a). \\
p(b) & \leftarrow s(b,b), \not q(b).
\end{align*}
\]

AN FPI DERIVES Q(B) AND P(A) BUT Q(A) AND P(B) CANNOT BE DERIVED UNDER THIS SLOPPY INTERPRETATION OF NEGATION.

CONSEQUENTLY, NOT Q(A) OUGHT TO BE CONSIDERED TO BE TRUE EVEN FOR THE PESSIMISTIC VIEW.

Additional condition for the fixpoint computation of the pessimist!
Alternating Fixpoint (4)

2nd pessimistic FPI

\[
\begin{align*}
q(a) &\leftarrow r(a). \\
q(b) &\leftarrow r(b). \\
p(a) &\leftarrow s(a,a), \quad \text{not } q(a). \\
p(a) &\leftarrow s(a,b), \quad \text{not } q(b). \\
p(b) &\leftarrow s(b,a), \quad \text{not } q(a). \\
p(b) &\leftarrow s(b,b), \quad \text{not } q(b).
\end{align*}
\]

Assuming \( \neg q(b) \equiv \text{false} \) and \( \neg q(a) \equiv \text{true} \), solely \( q(b) \) is derivable by a FPI for the pessimistic view.

No further additional condition for the optimist!
Assuming \( \neg Q(B) \equiv \text{false} \) and \( \neg Q(A) \equiv \text{true} \), \( Q(B) \) remains to be derivable by an FPI in the optimistic view but not \( P(A) \) anymore.

No further additional condition for the pessimist!
Alternating Fixpoint Iterations (1)

least fixpoint computation with fixed reference for negation
complementation

$H_D$
Alternating Fixpoint Iterations (2)

```
def⁻ poss⁻ 
poss⁺ def⁺
```

```
def⁻ poss⁻ 
poss⁺ def⁺
```

\[ i = i + 1 \]

- true
- unknown
- false
Alternating Fixpoint: General view

local fixpoint computation

global fixpoint computation
Doubled Program Approach

The original AFP works on negative conclusions (very large set in a database context).

The doubled program approach avoids the explicit representation of negative conclusions and works on definitely and possibly true facts, only:
Doubled Program Approach

\[ p(X) : \neg s(X,Y), \neg p(Y). \]

\[ s(1,2), s(2,3), s(3,4) \]

definitely true conclusions DEF

\[ dt_p(X) : \neg s(X,Y), \neg ndf_p(Y). \]

\[ ndf_p(X) : \neg s(X,Y), \neg dt_p(Y). \]

possibly true conclusions POSS
(nfd – not definitely false)

\[ lfp(R^p, POSS_1) \]
\[ ndf_p(1), ndf_p(2), ndf_p(3). \]
\[ lfp(R^d, DEF_1) \]
\[ dt_p(3) \]
\[ lfp(R^p, POSS_2) \]
\[ ndf_p(1). \]
\[ lfp(R^d, DEF_2) \]
\[ dt_p(1). dt_p(3). \]

\[ POSS_2 \setminus DT_2 = \text{undefined} \]

\[ DT_2 = \text{true} \]
Properties of the DP-Approach

Advantages:

- Data representation is purely relational
  - directly applicable within the DB context
  - supports DB indexing of the respective relations

- set-oriented rule application

Disadvantages:

- Possibly introducing redundant derivations due to rule doubling:

Possible Solution: [Behrend’05]