Logic Programming

View Evaluation

Michael Genesereth
Computer Science Department
Stanford University
Method
- Start with dataset
- Apply rules repeatedly to produce closure
- Repeat up the stratum hierarchy
- Evaluate query on the result

Disadvantages
- Generates large numbers of irrelevant conclusions
- Does not work with infinite extensions
Method
Start with query to be answered
Apply rules repeatedly to reduce to subqueries
Continue until reaching data level
Match base level subgoals against dataset

Disadvantages
Slightly harder to understand
Sometimes recomputes subgoals
Susceptible to *avoidable* infinite loops
Top-Down Processing of Ground Goals and Rules

Unification

Top-Down Processing of Goals and Rules with Variables
Ground Goals and Rules
Given a query, a dataset, and a ruleset, do the following.

(1) If the predicate in the query is a base predicate, succeed if and only if query is in dataset.

(2) If the query is a negation, evaluate target and succeed if and only if fail to prove.

(3) If the query is a conjunction, succeed iff succeed on all conjuncts.

(4) If the predicate in the query is a view predicate, evaluate the body of each rule defining that predicate and succeed if and only if succeeds on at least one rule.
<table>
<thead>
<tr>
<th>Dataset</th>
<th>Ruleset</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(a)</td>
<td>s(c) :- p(a) &amp; q(b)</td>
</tr>
<tr>
<td>p(b)</td>
<td>s(c) :- p(b) &amp; t(c)</td>
</tr>
<tr>
<td>p(c)</td>
<td>s(c) :- p(c) &amp; ~q(c)</td>
</tr>
<tr>
<td>q(d)</td>
<td>t(c) :- p(a) &amp; p(d)</td>
</tr>
</tbody>
</table>
Dataset
- p(a)
- p(b)
- p(c)
- q(d)

Ruleset
- s(c) :- p(a) & q(b)
- s(c) :- p(b) & t(c)
- s(c) :- p(c) & ~q(c)
- t(c) :- p(a) & p(d)

Top Down Evaluation

- s(c)?
  - p(a) & q(b)?
    - X
  - p(b) & t(c)?
  - p(c) & ~q(c)?
    - X
  - p(a) & p(d)?
    - X

Success
Unification
Unification is the process of determining whether two expressions can be unified, i.e. made identical by appropriate substitutions for their variables.

Example: \( p(a, y) \) and \( p(x, b) \) can be unified. If we replace \( x \) by \( a \) and \( y \) by \( b \), we end up with \( p(a, b) \) in both cases.
A substitution is a finite set of pairs of variables and terms, called replacements.

\{X \leftarrow a, Y \leftarrow f(b), Z \leftarrow V\}

Domain: \{x, y, z\}
Range: \{a, f(b), V\}

NB: Domain elements must be variables.
NB: Replacements may contain variables.
The result of applying a substitution $\sigma$ to an expression $\varphi$ is the expression $\varphi\sigma$ obtained from $\varphi$ by replacing every occurrence of every variable in the substitution by its replacement.

\[
\begin{align*}
q(X,Y) \{X\leftarrow a, Y\leftarrow f(b), Z\leftarrow V\} &= q(a, f(b)) \\
q(X,X) \{X\leftarrow a, Y\leftarrow f(b), Z\leftarrow V\} &= q(a,a) \\
q(X,W) \{X\leftarrow a, Y\leftarrow f(b), Z\leftarrow V\} &= q(a,W) \\
q(Z,V) \{X\leftarrow a, Y\leftarrow f(b), Z\leftarrow V\} &= q(V,V)
\end{align*}
\]
Cascaded Substitutions

\[ r(X, Y, Z)\{x\leftarrow a, y\leftarrow f(U), Z\leftarrow V\} = r(a, f(U), V) \]

\[ r(a, f(U), V)\{U\leftarrow d, V\leftarrow e, Z\leftarrow g\} = r(a, f(d), e) \]

\[ r(X, Y, Z)\{X\leftarrow a, Y\leftarrow f(d), Z\leftarrow e, U\leftarrow d, V\leftarrow e\} = r(a, f(d), e) \]
The *composition* of substitution $\sigma$ and $\tau$ is the substitution (written $\text{compose}(\sigma, \tau)$ or, more simply, $\sigma \tau$) obtained by

1. applying $\tau$ to the replacements in $\sigma$
2. adding to $\sigma$ pairs from $\tau$ with different variables
3. deleting any assignments of a variable to itself.

\[
\{X \leftarrow a, \ Y \leftarrow U, \ Z \leftarrow V\} \{U \leftarrow d, \ V \leftarrow e, \ Z \leftarrow g\} \\
= \{X \leftarrow a, \ Y \leftarrow d, \ Z \leftarrow e\} \{U \leftarrow d, \ V \leftarrow e, \ Z \leftarrow g\} \\
= \{X \leftarrow a, \ Y \leftarrow d, \ Z \leftarrow e, \ U \leftarrow d, \ V \leftarrow e\}
\]
A substitution $\sigma$ is a *unifier* for an expression $\varphi$ and an expression $\psi$ if and only if $\varphi\sigma = \psi\sigma$.

\[
p(X, Y)\{x\leftarrow a, \ y\leftarrow b, \ v\leftarrow b\} = p(a, b)
p(a, V)\{x\leftarrow a, \ y\leftarrow b, \ v\leftarrow b\} = p(a, b)
\]

If two expressions have a unifier, they are said to be *unifiable*. Otherwise, they are *nonunifiable*.

\[
p(X, X)
p(a, b)
\]
Non-Uniqueness of Unification

Unifier 1:
\[
\begin{align*}
  p(X, Y) & \{X \leftarrow a, Y \leftarrow b, V \leftarrow b\} = p(a, b) \\
  p(a, V) & \{X \leftarrow a, Y \leftarrow b, V \leftarrow b\} = p(a, b)
\end{align*}
\]

Unifier 2:
\[
\begin{align*}
  p(X, Y) & \{X \leftarrow a, Y \leftarrow f(W), V \leftarrow f(W)\} = p(a, f(W)) \\
  p(a, V) & \{X \leftarrow a, Y \leftarrow f(W), V \leftarrow f(W)\} = p(a, f(W))
\end{align*}
\]

Unifier 3:
\[
\begin{align*}
  p(X, Y) & \{X \leftarrow a, Y \leftarrow V\} = p(a, V) \\
  p(a, V) & \{X \leftarrow a, Y \leftarrow V\} = p(a, V)
\end{align*}
\]
A substitution $\sigma$ is a *most general unifier* (*mgu*) of two expressions if and only if it is *as general as or more general than* any other unifier.

Theorem: If two expressions are unifiable, then they have an mgu that is unique up to variable permutation.

\[
\begin{align*}
  p(X,Y)\{X\leftarrow a, Y\leftarrow V\} &= p(a,V) \\
  p(a,V)\{X\leftarrow a, Y\leftarrow V\} &= p(a,V) \\
  p(X,Y)\{X\leftarrow a, V\leftarrow Y\} &= p(a,Y) \\
  p(a,V)\{X\leftarrow a, V\leftarrow Y\} &= p(a,Y)
\end{align*}
\]
One good thing about our language is that there is a simple and inexpensive procedure for computing a most general unifier of any two expressions if it exists.
Each expression is treated as a sequence of its immediate subexpressions.

Linear Version:

\[ p(a, f(b, c), d) \]

Structured Version:
(1) If two expressions being compared are identical, succeed.

(2) If neither is a variable and at least one is a constant, fail.

(3) If one of the expressions is a variable, proceed as described shortly.

(4) If both expressions are sequences, iterate across the expressions, comparing each subexpression as described above.
If one of the expressions is a variable, check whether the variable has a binding in the current substitution.

(a) If so, try to unify the binding with the other expression.

(b) If no binding, check whether the other expression contains the variable. If the variable occurs within the expression, fail. Otherwise, set the substitution to the composition of the old substitution and a new substitution in which variable is bound to the other expression.
Example

Call: \( p(x, b), p(a, Y), \{\} \)

Call: \( p, p, \{\} \)
Exit: \( \{\} \)

Call: \( x, a, \{\} \)
Exit: \( \{\} \{x \leftarrow a\} = \{x \leftarrow a\} \)

Call: \( b, Y, \{x \leftarrow a\} \)
Exit: \( \{x \leftarrow a\} \{Y \leftarrow b\} = \{x \leftarrow a, Y \leftarrow b\} \)

Exit: \( \{x \leftarrow a, Y \leftarrow b\} \)
Call: \( p(x, x), p(a, y), \{\} \)

Call: \( p, p, \{\} \)
Exit: \( \{\} \)

Call: \( x, a, \{\} \)
Exit: \( \{\} \{x \leftarrow a\} = \{x \leftarrow a\} \)

Call: \( x, y, \{x \leftarrow a\} \)
  Call: \( a, y, \{x \leftarrow a\} \)
  Exit: \( \{x \leftarrow a\} \{y \leftarrow a\} = \{x \leftarrow a, y \leftarrow a\} \)
Exit: \( \{x \leftarrow a, y \leftarrow a\} \)

Exit: \( \{x \leftarrow a, y \leftarrow a\} \)
Call: p(x, x), p(a, b), {}

Call: p, p, {}
Exit: {}

Call: x, a, {}
Exit: {}{x←a} = {x←a}

Call: x, b, {x←a}
  Call: a, b, {x←a}
  Exit: false
Exit: false

Exit: false
Example

Call: p(X, X), p(Y, f(Y)), {}

Call: p, p, {}
Exit: {}

Call: X, Y, {}
Exit: {}{X←Y} = {X←Y}

Call: X, f(Y), {X←Y}
  Call: Y, f(Y), {X←Y}
  Exit: false
Exit: false

Exit: false
Reason

Circularity Problem:

\{X \leftarrow f(Y), Y \leftarrow f(Y)\}

Unification Problem:

\begin{align*}
    p(X, X)\{X \leftarrow f(Y), Y \leftarrow f(Y)\} &= p(f(Y), f(Y)) \\
    p(Y, f(Y))\{X \leftarrow f(Y), Y \leftarrow f(Y)\} &= p(f(Y), f(f(Y)))
\end{align*}

Before assigning a variable to an expression, first check that the variable does not occur within that expression.

This is called the occur check test.

*Prolog does not do the occur check (and is proud of it).*

*But it can give incorrect answers as a result.*
General Goals and Rules
Procedure without variables uses *equality* tests.

\[
p(a,b) \\
p(b,c) \\
s(a,c) :- p(a,b) \& p(b,c) \\
\]

\[
s(a,c)?
\]

Procedure with variables uses *unification*.

\[
p(a,b) \\
p(b,c) \\
s(X,Z) :- p(X,Y) \& p(Y,Z) \\
\]

\[
s(a,c)?
\]
Given an atom with a base relation and a substitution:

(a) Compare the goal to each factoid in our dataset.

(b) If there is an extension of the given substitution that unifies the goal and the factoid, add to our list of answers.

(c) Once all relevant factoids examined, return answers.
Goal: \( p(X, Y) \)
Substitution: \( \{X \leftarrow a\} \)
Dataset: \( \{p(a, b), p(a, c), p(b, c)\} \)
Result: \( \{\{X \leftarrow a, Y \leftarrow b\}, \{X \leftarrow a, Y \leftarrow c\}\} \)
Step 2 - Negations

Given a negation and a substitution:

(a) Execute the procedure on the target of the negation and the given substitution.

(b) If the result is empty, return a singleton list containing the given substitution, indicating success.

(c) Otherwise, return the empty list of answers, indicating failure.
Example 2 - Negations

Goal: \(~p(X,Y)\)
Substitution: \{X\leftarrow a, Y\leftarrow d\}
Dataset: \{p(a,b), p(a,c), p(b,c)\}
Result: \[[\{X\leftarrow a, Y\leftarrow d\}]\]

Goal: \(~p(X,Y)\)
Substitution: \{X\leftarrow a, Y\leftarrow c\}
Dataset: \{p(a,b), p(a,c), p(b,c)\}
Result: \[\[\]\]
Given a conjunction and a substitution:

(a) Execute our procedure on the first conjunct and the given substitution to get a list of answers.

(b) Iterate through the list of substitutions, calling the procedure recursively on the remaining conjuncts with each substitution in turn.

(c) Collect the answers from recursive calls and return.
Example 3 - Conjunctions

Goal: \( p(X, Y) \land p(Y, Z) \)
Substitution: \( \{X \leftarrow a\} \)
Dataset: \{p(a, b), p(a, c), p(b, c)\} 

Call: \( p(X, Y) \), \( \{X \leftarrow a\} \)
Result: \[ \{X \leftarrow a, Y \leftarrow b\}, \{X \leftarrow a, Y \leftarrow c\} \] 

Call: \( p(Y, Z) \), \( \{X \leftarrow a, Y \leftarrow b\} \)
Result: \[ \{X \leftarrow a, Y \leftarrow b, Z \leftarrow c\} \] 

Call: \( p(Y, Z) \), \( \{X \leftarrow a, Y \leftarrow c\} \)
Result: [] 

Overall Result: \[ \{X \leftarrow a, Y \leftarrow b, Z \leftarrow c\} \]
Given atom with view relation and a substitution:
(a) Iterate through the rules in our program.
(b) *Copy each rule, replacing variables with new variables.*
(c) Try to unify the given goal and the new rule head.
(d) Call the procedure recursively on the body of the rule.
(e) Return substitutions from all successful cases.
Example 4 - Atoms with View Relations

Goal: \( q(X,Y) \)
Substitution: \( \{ X \leftarrow a \} \)
Rule: \( q(X,Z) :- p(X,Y) \& p(Y,Z) \)
Dataset: \( \{ p(a,b), p(a,c), p(b,c) \} \)

Copy of rule: \( q(U,W) :- p(U,V) \& p(V,W) \)

Unification: \( q(U,W) \quad q(X,Y) \quad \{ X \leftarrow a \} \)
Result: \( \{ U \leftarrow a, W \leftarrow Y, X \leftarrow a \} \)

New Goal: \( p(U,V) \& p(V,W) \)
New Substitution: \( \{ U \leftarrow a, W \leftarrow Y, X \leftarrow a \} \)

Result: \( \{ \{ U \leftarrow a, W \leftarrow c, X \leftarrow a, V \leftarrow b, Y \leftarrow c \} \} \)
Compound terms compound the difficulty.

Rule
\[ s(X, f(Y, Z)) :\text{ } :- \text{ } p(X, g(Y)) \text{ } \& \text{ } p(Y, X) \]

Query
\[ s(h(X), X) \]

Subgoal
\[ p(h(f(Y, Z)), g(Y)) \text{ } \& \text{ } p(Y, h(f(Y, Z))) \]
Multiple substitutions
  Different substitutions used for goals and rules
  *Good: Rules are not copied*

Evaluation of conjuncts is pipelined
  Once each answer to a conjunct is computed, the other conjuncts are checked immediately; then other answers generated and checked.
  *Good: Saves work when only few answers needed.*
  *Good: Avoids problems due to infinite answer sets.*

Upshot: This is complicated. Don't try this at home. Leave it to the professionals.
Facts and Rules

\[
p(a, b)
p(b, c)
s(X, Z) :- p(X, Y) \& p(Y, Z)
\]

Trace

- Call: \( s(X, Z) \)
  - Call: \( p(X, Y) \)
  - Exit: \( p(a, b) \)
  - Call: \( p(b, Z) \)
  - Exit: \( p(b, c) \)
- Exit: \( s(a, c) \)
Facts and Rules

\[ p(a,b) \]
\[ p(b,c) \]
\[ s(X,Z) : \text{if} \ p(X,Y) \ & \ p(Y,Z) \]

Trace

Call: \( s(X,Z) \)  
| Call: \( p(X,Y) \)  
| Exit: \( p(a,b) \)  
| Call: \( p(b,Z) \)  
| Exit: \( p(b,c) \)  
Exit: \( s(a,c) \)

Redo: \( s(X,Z) \)  
| Redo: \( p(b,Z) \)  
| Fail: \( p(b,Z) \)  
| Redo: \( p(X,Y) \)  
| Exit: \( p(b,c) \)  
| Call: \( p(c,Z) \)  
| Fail: \( p(c,Z) \)  
Fail: \( s(X,Z) \)
Summary
Comparison of Evaluation Strategies

**Bottom-Up Evaluation**
- Easy to understand
- Computes all results
- Computes subresults just once
- Wasteful when want just one or a few answers, not all
- Problematic on logic programs with infinite models

**Top-Down Evaluation**
- Less waste when want one or a few answers
- More complicated
- Subqueries evaluated multiple times
- Possibility of infinite loops on programs w/ finite models
Bottom-Up Evaluation
   Can be focussed using Magic Sets

Top-Down Evaluation
   Top-Down can avoid duplication through caching
   Infinite Loops can be avoided using iterative deepening

The arms race continues.
Sierra
Sierra is browser-based IDE (interactive development environment) for Epilog.

- Saving and loading files
- Viewing and Editing datasets
- Querying datasets
- Transformation tools for datasets
- Interpreter (for view definitions, action definitions)
- Trace capability (useful for debugging rules)
- Analysis tools (error checking and optimizing rules)

http://epilog.stanford.edu/homepage/sierra.php
Lambda

\[p(a,b)\]
\[p(b,c)\]
\[p(c,d)\]
\[p(d,e)\]

Query

Pattern \(\text{goal}(X,Z)\)
Query \(p(X,Y) \& p(Y,Z)\)

Transform

Condition \(p(X,Y)\)
Conclusion \(-p(X,Y) \& p(Y,X)\)

- \(-p(a,b)\)
- \(p(b,a)\)
- \(-p(b,c)\)
- \(p(c,b)\)
- \(-p(c,d)\)
- \(p(d,c)\)
- \(-p(d,e)\)
- \(p(e,d)\)
Lambda:

\[ p(a,b) \]
\[ p(b,c) \]
\[ p(c,d) \]
\[ p(d,e) \]

Query:

Pattern: \( \text{goal}(x, z) \)

Query: \( p(x, y) \land p(y, z) \)

Library:

\[ \text{anc}(X, Y) :- p(X, Y) \]
\[ \text{anc}(X, Z) :- p(X, Y) \land \text{anc}(Y, Z) \]

Transform:

Condition: \( p(X, Y) \)

Conclusion: \( \neg p(X, Y) \land p(Y, X) \)

Execute: \( \neg p(a,b) \)
\( p(b,a) \)
\( \neg p(b,c) \)
\( p(c,b) \)
\( \neg p(c,d) \)
\( p(d,c) \)
\( \neg p(d,e) \)
\( p(e,d) \)
Lambda

\[ p(a,b) \]
\[ p(b,c) \]
\[ p(c,d) \]
\[ p(d,e) \]

Library

\[ \text{anc}(X,Y) :- p(X,Y) \]
\[ \text{anc}(X,Z) :- p(X,Y) \& \text{anc}(Y,Z) \]

Query

Pattern: \( \text{goal}(X,Z) \)
Query: \( p(X,Y) \& p(Y,Z) \)

Transform

Condition: \( p(X,Y) \)
Conclusion: \( \neg p(X,Y) \& p(Y,X) \)

Expand on update

Execute

Run on clock tick

- \( p(a,b) \)
- \( p(b,a) \)
- \( p(b,c) \)
- \( p(c,b) \)
- \( p(c,d) \)
- \( p(d,c) \)
- \( p(d,e) \)
- \( p(e,d) \)
Lambda

\begin{align*}
p(a, b) \\
p(b, c) \\
p(c, d) \\
p(d, e) \\
\end{align*}

Library

\begin{align*}
\text{anc}(X, Y) &\triangleq p(X, Y) \\
\text{anc}(X, Z) &\triangleq p(X, Y) \land \text{anc}(Y, Z) \\
\end{align*}

Query

Pattern:  \text{goal}(X, Z) \\
Query:  p(X, Y) \land p(Y, Z)

\begin{align*}
goal(a, c) \\
goal(b, d) \\
goal(c, e) \\
\end{align*}

Compute

Query: \text{anc}(b, z)

\begin{align*}
\text{anc}(b, c) \\
\text{anc}(b, d) \\
\text{anc}(b, e) \\
\end{align*}

Transform

Condition:  p(X, Y)

Conclusion:  \neg p(X, Y) \land p(Y, X)

Expand: Yes \\
Execute: Yes

\begin{align*}
\neg p(a, b) \\
p(b, a) \\
\neg p(b, c) \\
p(c, b) \\
\neg p(c, d) \\
p(d, c) \\
\neg p(d, e) \\
p(e, d) \\
\end{align*}
Lambda

Pattern: goal(X, Z)
Query: p(X, Y) & p(Y, Z)

Query: anc(b, Z)

Library

anc(X, Y) :- p(X, Y)
anc(X, Z) :- p(X, Y) & anc(Y, Z)

Transform

Condition: p(X, Y)
Conclusion: ~p(X, Y) & p(Y, X)

Execute: | Expand on update: | Run on clock tick
-p(a, b)
p(b, a)
-p(b, c)
p(c, b)
-p(c, d)
p(d, c)
-p(d, e)
p(e, d)
-p(e, f)
p(f, e)