Resolution Theorem Proving

Robert Keller
7 March 2005
What is this?

- Resolution is a special kind of theorem proving used for:
  - Automated theorem proving and reasoning
  - Answer extraction and databases
  - Prolog language
- Resolution in itself is a complete proof rule.
How it works

- A special stripped-down representation is used: “clausal form”.
- Quantifiers have been eliminated.
- A formula is proved by **refutation**, i.e. showing that its negation is unsatisfiable (as with the tree method).
Two Types of Resolution

- Predicate calculus resolution:
  - Our main objective

- Propositional resolution:
  - Needed to understand predicate resolution
  - Used in algorithms and complexity theory (NP completeness, for example)
Propositional Version of Resolution

- A **literal** is a proposition symbol or its negation.

- A **clause** is a disjunction of literals.

- The **negation** of the formula to be proved is first converted to a **clause set**, effectively a **conjunction** of those clauses.

- The original formula is a theorem iff the clauses in the set are **not simultaneously satisfiable**.
Example

- **Clause set:**
  - \( p \lor \neg q \)
  - \( q \lor r \)
  - \( \neg p \)
  - \( \neg r \)

- **This clause set is unsatisfiable:**
  - There is no assignment of T or F to \( \{p, q, r\} \) which makes all formulas T simultaneously.
Example

• Clause set:
  • \( p \lor \neg q \)
  • \( \neg q \lor \neg r \)
  • \( q \)

• This clause set is satisfiable:
  • \( p = q = T, r = F \) will satisfy them simultaneously.
Equivalence of Clause Sets

• Two clause sets are called **equivalent** if they are satisfied by the same set of assignments.

• In particular, if two clause sets are equivalent, they are either
  • both satisfiable, or
  • both unsatisfiable
Reduced Clause Sets

• A clause set is **reduced** provided:
  • No literal occurs multiple times in any clause.
    • $p \lor \neg q \lor p$ is disallowed in a reduced set.
  • No clause contains a literal and its negation.
    • $p \lor q \lor \neg p$ is disallowed in a reduced set.

• Any clause set $S$ is equivalent to a reduced set $\text{reduce}(S)$:
  • Replace multiple occurrences of a literal with a single occurrence.
  • Drop any clauses containing a literal and its negation.
reduce example

\[\text{reduce}\left(\{p \lor \neg q \lor p, \ p \lor q \lor \neg p \lor q\}\right) = \{p \lor \neg q\}\]
Resolution Method

• Input: A reduced set of clauses.

• Output: A set of clauses equivalent to the input set, such that the original set is unsatisfiable iff the final set contains the null clause $\bot$ (also designated by $\square$).
How Resolution Works

• Do Repeatedly:
  • From the set of clauses, pick a pair from which a new clause is created, called the “resolvent”.

  • Add the resolvent to the set.

• If ⊥ is ever added to the set, the original set of clauses is unsatisfiable.

• Conversely, if the original set of clauses is unsatisfiable, it is possible to eventually derive ⊥.
What is the Resolvent?

• Suppose $p$ is a proposition symbol.
• If the set contains both
  • $p \lor \varphi$
  • $\neg p \lor \psi$
• where $\varphi$ and $\psi$ are formulas (either could be empty), then the resolvent is
  • $\varphi \lor \psi$
Resolution as a Deduction Rule

\[
\begin{align*}
& p \lor \varphi \\
\hline
& \neg p \lor \psi \\
& \varphi \lor \psi
\end{align*}
\]

where \( p \) is any proposition symbol and \( \varphi \) and \( \psi \) are clauses (either could be empty).
Example of Resolvents

• Consider the clauses
  • p ∨ ¬q
  • q ∨ r

• Since q and ¬q occur in different clauses, the resolvent is:
  • p ∨ r
Example of Resolvents

- Consider the clauses
  - \( p \lor r \)
  - \( \neg r \)
- Since \( r \) and \( \neg r \) occur in different clauses, the resolvent is:
  - \( p \)
Example of Resolvents

- Consider the clauses
  - p
  - \( \neg p \)
- Since \( p \) and \( \neg p \) occur in different clauses, the resolvent is:
  - \( \bot \)
Resolution Algorithm (Crude form)

- Input: $S$, the clause set to be tested.

\[
S := \text{reduce}(S); \\
T := \text{reduce}(\text{resolveall}(S)); \\
\text{while}( \neg (T \subseteq S) ) \text{ do} \\
\quad S := S \cup T; \\
\quad T := \text{reduce}(\text{resolveall}(S));
\]

- where $\text{resolveall}(S)$

\[
= \{ \varphi \lor \psi \mid (p \lor \varphi) \in S \land (\neg p \lor \psi) \in S \}\]
## Resolution Algorithm Step

- \{p \lor \neg q, q \lor r, \neg p, \neg r\} original clause set

- resolvents:

\[
\begin{array}{cccc}
  & p \lor \neg q & q \lor r & \neg p & \neg r \\
\hline
p \lor \neg q & & p \lor r & \neg q & \\
q \lor r & & & & q \\
\neg p & & & & \\
\neg r & & & & (symmetric cases)
\end{array}
\]

- union: \{p \lor \neg q, q \lor r, \neg p, \neg r, p \lor r, \neg q, q\}
Resolution Algorithm Step

- set: \( \{ p \lor \neg q, q \lor r, \neg p, \neg r, p \lor r, \neg q, q \} \)

  Resolvents: \( \{ p \lor r, \neg q, q, p, r, \bot \} \)

- Even though the algorithm is not done, we can see now that the original set of clauses is unsatisfiable, since \( \bot \) is present.
Adding the resolvent does not alter satisfiability

- A reduced clause set $\Gamma \cup \{p \lor \varphi, \neg p \lor \psi\}$ is equivalent to $\Gamma \cup \{p \lor \varphi, \neg p \lor \psi, \varphi \lor \psi\}$

- Reason: Suppose $\alpha$ is an assignment that satisfies both $p \lor \varphi$ and $\neg p \lor \psi$.

  - Suppose $\alpha(p) = T$. Then $\alpha$ induces $F$ in $\neg p$. But since $\alpha$ satisfies $\neg p \lor \psi$, $\alpha$ must therefore induce $T$ in $\psi$.

  - The case for $\alpha(p) = F$ is symmetric.

- Converse: Next slide.
Adding the resolvent does not alter satisfiability (converse)

- A reduced clause set \( \Gamma \cup \{p \lor \varphi, \neg p \lor \psi\} \) is equivalent to \( \Gamma \cup \{p \lor \varphi, \neg p \lor \psi, \varphi \lor \psi\} \)

- **Converse**: If \( \alpha \) satisfies \( \varphi \lor \psi \) (which contains neither \( p \) nor \( \neg p \)), then it must satisfy one or the other of \( \varphi \) or \( \psi \).

- If \( \alpha \) satisfies \( \varphi \) then it also satisfies \( p \lor \varphi \). Extend \( \alpha \) to \( \alpha' \) such that \( \alpha(p) = F \). Then \( \alpha \) induces \( T \) in \( \neg p \lor \psi \) also.

- If not, then \( \alpha \) satisfies \( \psi \), so extend \( \alpha \) to \( \alpha' \) such that \( \alpha(p) = T \). Then \( \alpha \) induces \( T \) in \( p \lor \varphi \) and also in \( \neg p \lor \psi \).
What if the Clause Set is Satisfiable?

• \{p \lor \neg q, \neg q \lor \neg r, q\}

• \{p \lor \neg q, \neg q \lor \neg r, q, p, \neg r\}

• Closure: There are no other resolvents, yet \bot has not been derived.
Resolution Algorithm Invariant

- **Input:** $S$, the clause set to be tested.
  
  $\{S = S_0\}$
  
  $S := \text{reduce}(S)$;
  
  $T := \text{reduce}(\text{resolveall}(S))$;
  
  while( $\neg(T \subseteq S)$ )
  
  $S := S \cup T$;
  
  $T := \text{reduce}(\text{resolveall}(S))$;

- **Invariant:**
  
  $\{S$ is unsatisfiable $\equiv S_0$ is unsatisfiable$\}$
Resolution Algorithm Termination

- Closure is always achievable.
- The set of distinct reduced clause sets for a given set of proposition symbols is **finite**.
- At worst, every possible clause (regarding reordering of symbols as equivalent) will be generated.
- How many distinct clauses can there be?
Resolution Algorithm Refinement 1

- Input: S, the clause set to be tested.

\[
S := \text{reduce}(S);
\]
\[
T := \text{reduce}(\text{resolveall}(S));
\]
\[
\text{while}(\bot \notin S \wedge \neg(T \subseteq S))
\]
\[
S := S \cup T;
\]
\[
T := \text{reduce}(\text{resolveall}(S));
\]
Resolution as a Tree

$p \lor \neg q$
$q \lor r$
$\neg r$
$\neg p$

$p$
$q$
$\bot$
Resolution as a “Proof”

1. $p \lor \neg q$  
   Premise
2. $q \lor r$  
   Premise
3. $\neg r$  
   Premise
4. $\neg p$  
   Premise
5. $q$  
   Resolution 2, 3
6. $p$  
   Resolution 1, 5
7. $\bot$  
   Resolution 6, 4
Try these:

1. \( \{\neg p \lor \neg q \lor \neg r, \quad \neg q \lor r, \quad q \lor s, \quad \neg s, \quad p, \quad \} \)

2. \( \{p \lor \neg q \lor r, \quad q \lor \neg r, \quad \neg p \} \)
Sometimes a DAG is more appropriate than a tree for showing all options.

We avoid identifying the two ⊥ nodes, so as not to confuse the two sets of antecedents.
Resolution is a Complete Rule

• The single rule of resolution is **refutation-complete**: If a set of clauses is unsatisfiable, this can be determined using only the resolution rule.

• However, considerable logic went into getting everything into clausal form in the first place, so it is perhaps **unfair** to compare the single rule to the set of natural deduction rules, which cover all logical steps.
Resolution Algorithm Refinement 2

- The idea is to avoid revisiting pairs that were resolved in earlier steps.

- Input: $S$, the clause set to be tested.
  
  $S := \text{reduce}(S)$;
  
  $T := \text{reduce}(\text{resolveall}(S, S)) - S$;
  
  while ($\bot \notin S \land T \neq \emptyset$)
  
    $S := S \cup T$;
  
    $T := \text{reduce}(\text{resolveall}(S, T)) - S$;

- $\text{resolveall}(S, T)$

  $$\{\varphi \lor \psi \mid (p \lor \psi) \in S \land (\neg p \lor \psi) \in T\}$$
  $$\cup \{\varphi \lor \psi \mid (p \lor \psi) \in T \land (\neg p \lor \psi) \in S\}$$
Selective Resolution

• Rather than resolving all pairs of clauses, try to pick pairs that will produce ⊥ in the fewest number of steps.

• Consider using a “non-deterministic” expression of the algorithm (pick clauses to resolve).
How General is the Clausal Form?

• Every propositional formula can be represented in clausal form.

• Examples:
  • $p \lor q$ in clausal form is $\{p \lor q\}$.
  • $p \land q$ in clausal form is $\{p, q\}$.
  • $p \rightarrow q$ in clausal form is $\{\neg p \lor q\}$.

• In general, to get clausal form:
  • Replace $\varphi \rightarrow \psi$ with $(\neg \varphi \lor \psi)$.
  • Push $\neg$ toward proposition symbols:
    • Replace $\neg(\varphi \land \psi)$ with $(\neg \varphi \lor \neg \psi)$.
    • Replace $\neg(\varphi \lor \psi)$ with $(\neg \varphi \land \neg \psi)$.
    • Replace $\neg \neg \varphi$ with $\varphi$.
  • Distribute $\lor$ inward:
    • Replace $\chi \lor (\varphi \land \psi)$ with $(\chi \lor \varphi) \land (\chi \lor \psi)$.
    • Replace $(\varphi \land \psi) \lor \chi$ with $(\varphi \lor \chi) \land (\psi \lor \chi)$. 
Example of Conversion to Clauses

- \neg (p \rightarrow (\neg q \land (r \land \neg s)))
- \neg (\neg p \lor (\neg q \land (r \land \neg s)))
- \neg \neg p \land \neg (\neg q \land (r \land \neg s))
- p \land \neg (\neg q \land (r \land \neg s))
- p \land (\neg \neg q \lor \neg (r \land \neg s))
- p \land (q \lor \neg (r \land \neg s))
- p \land (q \lor \neg r \lor \neg \neg s)
- p \land (q \lor \neg r \lor s)
- \{p, \ q \lor \neg r \lor s\}
Clausal Form from Truth Table

- A clausal form can be extracted from the truth table for any expression, as conjunctive normal form (CNF).

- For the rows for which the value is F, form a conjunction the corresponding literals.

- Overall, we have a disjunction of conjoined literals, representing the negation of the expression.

- Then change the disjunction to a conjunction, the individual conjunctions to disjunctions, and complement each literal (appealing to DeMorgan’s laws).
Example CNF from Truth Table

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>r</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

\[
p'q'r' \lor p'q'r' \lor pq'r: \text{ negate to get } \{p \lor q \lor r, p \lor q \lor \neg r, \neg p \lor q \lor \neg r\}
\]
**Check CNF from Truth Table**

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>r</th>
<th>value</th>
<th>pvqvr</th>
<th>pvqv¬r</th>
<th>¬pvqv¬r</th>
<th>conj.</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

\[ p'q'r' \lor p'q'r \lor pq'r \text{: negate to get } \{pvqvr, pvqv¬r, ¬pvqv¬r\} \]
Common Special Case to Clause Set

• Often we want to prove a sequent such as:
  \[ \varphi_{11} \land \varphi_{12} \lor \ldots \lor \varphi_{1m_1} \rightarrow \psi_1, \]
  \[ \varphi_{21} \land \varphi_{22} \lor \ldots \lor \varphi_{2m_2} \rightarrow \psi_2, \]
  \[ \ldots \]
  \[ \varphi_{n1} \land \varphi_{n2} \lor \ldots \lor \varphi_{nm_n} \rightarrow \psi_n \]
  \[ \models \chi_1 \land \chi_2 \lor \ldots \lor \chi_p \]
  where each symbol represents a literal.

• This can be done by showing that the following clause set is unsatisfiable:

\[ \{ \neg \varphi_{11} \lor \neg \varphi_{12} \lor \ldots \lor \neg \varphi_{1m_1} \lor \psi_1, \]
\[ \neg \varphi_{21} \lor \neg \varphi_{22} \lor \ldots \lor \neg \varphi_{2m_2} \lor \psi_2, \]
\[ \ldots \]
\[ \neg \varphi_{n1} \lor \neg \varphi_{n2} \lor \ldots \lor \neg \varphi_{nm_n} \lor \psi_n, \]
\[ \neg \chi_1 \lor \neg \chi_2 \lor \ldots \lor \neg \chi_p \} \]
Horn Clauses

- A Horn clause is one in which there is at most one non-negated literal:
  - $\neg \varphi_1 \lor \neg \varphi_2 \lor \ldots \lor \neg \varphi_m \lor \psi$
  or
  - $\neg \varphi_1 \lor \neg \varphi_2 \lor \ldots \lor \neg \varphi_m$

- Horn clause are the basis of the Prolog language, where:
  $$\neg \varphi_1 \lor \neg \varphi_2 \lor \ldots \lor \neg \varphi_m \lor \psi$$

is written

$$\psi \vdash \varphi_1, \varphi_2, \ldots, \varphi_m.$$
Prolog uses a special form of resolution to do its work (SLD resolution)

- \{p \lor \neg r \lor \neg s,
  r \lor \neg q,
  s \lor \neg q,
  q,
  \neg p,\}

becomes

- p :- r, s.
  r :- q.
  s :- q.
  q.

?- p.
Resolution for Predicate Logic

- **Predicate Clausal Form:**
  - A literal is an atomic formula or its negation, instead of a proposition symbol or its negation.
  - The variables of each clause are implicitly $\forall$-quantified.
  - The variables of each clause are independent from the other clauses; even if they are the same, they should be thought of as being different (e.g. implicitly rename by indexing with a clause number).
Example: Predicate Clausal Form

- \{p(X), q(X, Y), \neg q(X, X) \lor p(X)\}
  stands for the conjunction

- \(\forall X \ p(X)\)
  \(\land \forall X \ \forall Y \ q(X, Y)\)
  \(\land \forall X \ \forall Y (\neg q(X, X) \lor p(X))\)

which is the same as
- \(\forall X_1 \ p(X_1)\)
  \(\land \forall X_2 \ \forall Y_2 \ q(X_2, Y_2)\)
  \(\land \forall X_3 \ \forall Y_3 (\neg q(X_3, X_3) \lor p(X_3))\)

i.e. the clause set
- \(\{p(X_1), q(X_2, Y_2), \neg q(X_3, X_3) \lor p(X_3)\}\)
How General is This?

- We will see later that it is very general, as far as showing unsatisfiability is concerned.
Examples of Predicate Clausal Form

- $\neg \text{man}(X) \lor \text{mortal}(X)$
- $\text{man}(\text{socrates})$
- $\neg \text{mortal}(\text{socrates})$

- These clauses can be used to prove the syllogism:
  - All men are mortal.
  - Socrates is a man.
  - Therefore Socrates is mortal.
Resolution for Predicate Clauses

- To resolve predicate clauses, it is no longer sufficient to look for just a literal and its negation in two distinct clauses, e.g. \( p(X) \) in
  \[
  \neg q(X, X) \lor p(X) \\
  \neg p(X) \lor r(X, Y)
  \]
- For one thing, the identity of the variables is independent in each.
- For another, the arguments are generally **terms**, not just simple variables:
  \[
  \neg q(X, X) \lor p(f(X)) \\
  \neg p(X) \lor r(g(X), c)
  \]
What Resolution Must Do

• Suppose we have derived three formulas (where c is a constant symbol):
  • p(c)
  • ∀X (p(X) → q(f(X)))
  • ∀X (q(X) → r(X, g(X)))

• We would expect to be able to infer
  • q(f(c))
  • r(f(c), g(f(c)))

• Resolution must be able to handle such things.
Equivalent Clausal Form

• The clausal form of
  • \( p(c) \)
  • \( \forall X \ (p(X) \rightarrow q(f(X))) \)
  • \( \forall X \ (q(X) \rightarrow r(X, g(X))) \)

• is
  • \( \{p(c), \neg p(X) \lor q(f(X)), \neg q(X) \lor r(X, g(X))\} \)

• Resolution has to be able to “make a connection” between \( p(c) \) and \( p(X) \), and between \( q(f(X)) \) and \( q(X) \).
Unification

- The “connection” alluded to on the previous slide is known as unification.

- Two atoms are unifiable if there is a uniform substitution of terms for their variables that makes them identical.

- If such a substitution exists, it is applied to all literals in the formulas prior to resolution.
Unification Examples

• Consider atoms \( p(c), p(X) \) (\( c \) is a constant).
  These terms are **unifiable**, since the substitution \( \{ X \leftarrow c \} \) makes them identical.

• Consider \( q(c, d), q(X, X) \) (\( c \) and \( d \) are constants).
  These terms are **not unifiable**, since distinct constant symbols designate distinct individuals. There is no substitution that will make them identical.
Literals from Different Clauses

- Remember that variables don’t carry across clauses.

- If we are considering whether two literals in different clauses unify, we first must rename the variables so that there is no overlap.
Literals from Different Clauses

• Consider $p(X, f(Y))$ vs. $p(g(Y), f(X))$
• These might appear not to unify, since we would have a conflict $X ← g(Y)$ vs. $X ← Y$.

• However, if we rename the variables in the second clause we get:
  
  $p(X, f(Y))$ vs. $p(g(Z), f(W))$.

• These unify with $X ← g(X)$, $Y ← W$.

• **Note**: Renaming is done only at the start of considering unification of two clauses, and all variables in the clause are renamed uniformly.
### More Unification Examples

<table>
<thead>
<tr>
<th>Term 1</th>
<th>Term 2</th>
<th>Unifiable?</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(X, X)</td>
<td>p(f(Y), f(Z))</td>
<td></td>
</tr>
<tr>
<td>p(X, X)</td>
<td>p(f(Y), g(Y))</td>
<td></td>
</tr>
<tr>
<td>p(X, Y)</td>
<td>p(Z, f(Z))</td>
<td></td>
</tr>
<tr>
<td>p(X, f(X))</td>
<td>p(g(Y), W)</td>
<td></td>
</tr>
<tr>
<td>p(X, f(X))</td>
<td>p(f(Y), Y)</td>
<td></td>
</tr>
</tbody>
</table>
Even More Unification Examples
(assume renaming was already done; how these can arise will be seen later)

<table>
<thead>
<tr>
<th>Term 1</th>
<th>Term 2</th>
<th>Unifiable?</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>p(X, Y)</code></td>
<td><code>p(f(Y), g(Z))</code></td>
<td></td>
</tr>
<tr>
<td><code>p(X, f(X))</code></td>
<td><code>p(f(Y), Y)</code></td>
<td></td>
</tr>
<tr>
<td><code>p(f(X), Y)</code></td>
<td><code>p(X, Y)</code></td>
<td></td>
</tr>
<tr>
<td><code>p(f(X), f(X))</code></td>
<td><code>p(c, c)</code></td>
<td></td>
</tr>
<tr>
<td><code>p(f(X), g(X))</code></td>
<td><code>p(Y, g(Y))</code></td>
<td></td>
</tr>
</tbody>
</table>
Most General Unifiers (mgu)

• If two literals unify at all, they have a “most general unifier”, one which adds no unneeded constraints.

• Example: \( p(X) \) vs. \( p(f(Y)) \) could be unified with the substitution 
  \[ X \leftarrow f(c), \ Y \leftarrow c. \]

• However, this would not be the most general, since we could leave \( Y \) as a variable: 
  \[ X \leftarrow f(Y) \]
  and they would unify with this, which is a “more general” substitution.
Notation for Variable Substitutions

• In general, a substitution consists of a set of bindings of variables to terms, e.g.

\[ \beta = \{X \leftarrow Z, \ Y \leftarrow f(Z, \ c), \ W \leftarrow c\} \]

• If \( \tau \) is a term, then \( \tau_\beta \) denotes the result of making the substitutions \( \beta \) in for variables in \( \tau \), e.g.

\[
\begin{align*}
\tau &= p(X, g(Y, \ W)) \\
\tau_\beta &= p(Z, g(f(Z, \ c), \ c))
\end{align*}
\]
Composing Variable Substitutions

- If $\beta$ and $\gamma$ are substitutions and $\tau$ is a term, then $(\tau\beta)\gamma$ denotes the result of first applying $\beta$ to $\tau$, then $\gamma$ to the result, e.g.
  
  \[\beta = \{X \leftarrow Z, Y \leftarrow f(Z, c), W \leftarrow c\}\]
  \[\gamma = \{Z \leftarrow V\}\]
  \[\tau = p(X, g(Y, W))\]
  \[(\tau\beta)\gamma = p(V, g(f(V, c), c))\]

- The composition $\beta\gamma$ of two substitutions is the substitution such that for all terms $\tau$
  
  \[\tau(\beta\gamma) = (\tau\beta)\gamma.\]
Generality of Substitutions

- Substitution $\beta$ is **at least as general as** substitution $\alpha$ if there is a $\gamma$ such that $\alpha = \beta \gamma$.

- To say that $\beta$ is a “most general unifier” really means that is at least as general as any other unifier.
Find the MGU or indicate non-unifiable

<table>
<thead>
<tr>
<th>Term 1</th>
<th>Term 2</th>
<th>MGU?</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(X, Y)</td>
<td>p(Z, Z)</td>
<td></td>
</tr>
<tr>
<td>p(X, c)</td>
<td>p(Y, Y)</td>
<td></td>
</tr>
<tr>
<td>p(f(X), Y)</td>
<td>p(Y, f(Z))</td>
<td></td>
</tr>
<tr>
<td>p(f(X), Y)</td>
<td>p(X, Y)</td>
<td></td>
</tr>
<tr>
<td>p(f(Z), g(X))</td>
<td>p(Y, g(Y))</td>
<td></td>
</tr>
</tbody>
</table>
Note on Unification in Prolog

• In Prolog, unification is used in goal matching and in the `=`, operator.
• However, Prolog’s unification is slightly **abridged**: it avoids the “occur check”:
  \[ X = f(X) \]
  will unify in Prolog, but not in ordinary unification. In effect, \( X \) gets bound to the infinite term:
  \[ f(f(f(\ldots))) \]