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 agorithms 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:
  - $p \vee \neg q$
  - $q \vee r$
  - $\neg p$
  - $\neg r$
- 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:
  - $p \vee \neg q$
  - $\neg q \vee \neg r$
  - 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.
    - $p \vee \neg q \vee p$ is disallowed in a reduced set.
  - No clause contains a literal and its negation.
    - $p \vee q \vee \neg p$ is disallowed in a reduced set.
- Any clause set S is equivalent to a reduced set reduce(S):
  - Replace multiple occurrences of a literal with a single occurrence.
  - Drop any clauses containing a literal and its negation.

reduce example

$$\text{reduce}(\{p \vee \neg q \vee p,\ \ p \vee q \vee \neg p \vee q\}) =$$

$$\{p \vee \neg q\}$$

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** $\perp$ (also designated by ❑).

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 $\perp$ 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 $\perp$.

What is the Resolvent?

- Suppose p is a proposition symbol.
- If the set contains both
  - $p \vee \varphi$
  - $\neg p \vee \psi$
- where φ and ψ are formulas (either could be empty), then the resolvent is
  - $\varphi \vee \psi$

Resolution as a Deduction Rule

$$\frac{p \vee \varphi \qquad\qquad \neg p \vee \psi}{\varphi \vee \psi}$$

where p is any proposition symbol and φ and ψ are clauses (either could be empty).

Example of Resolvents

- Consider the clauses
  - $p \vee \neg q$
  - $q \vee r$
- Since q and $\neg q$ occur in different clauses, the resolvent is:
  - $p \vee r$

Example of Resolvents

- Consider the clauses
  - $p \vee r$
  - $\neg r$
- Since r and ¬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:
  - $\perp$

Resolution Algorithm (Crude form)

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

  S := reduce(S);
  T := reduce(resolveall(S));

  while($\neg$(T $\subseteq$ S))
  S := S $\cup$ T;
  T := reduce(resolveall(S));

- where resolveall(S)

  $$= \{\varphi \vee \psi \mid (p \vee \varphi) \in S \wedge (\neg p \vee \psi) \in S\}$$

Resolution Algorithm Step

- {p ∨ ¬q, q ∨ r, ¬p, ¬r} original clause set
- resolvents:

	p ∨ ¬q	q ∨ r	¬p	¬r
p ∨ ¬q		p ∨ r	¬q	
q ∨ r				q
¬p				
¬r	(symmetric cases)			

- union: {p ∨ ¬q, q ∨ r, ¬p, ¬r, p ∨ r, ¬q, q}

Resolution Algorithm Step

- set: $\{p \vee \neg q, q \vee r, \neg p, \neg r, p \vee r, \neg q, q\}$

  Resolvents: $\{p \vee r, \neg q, q, p, r, \perp\}$

- Even though the algorithm is note done, we can see now that the original set of clauses is unsatisfiable, since $\perp$ is present.

Adding the resolvent does not alter satisfiability

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

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

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

- The case for $\alpha(p) = F$ is symmetric.
- Converse: Next slide.

Adding the resolvent does not alter satisfiability (converse)

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

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

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

- If not, then α satisfies ψ, so extend α to α' such that $\alpha(p) = T$. Then α induces T in $p \vee \varphi$ and also in $\neg p \vee \psi$.

What if the Clause Set *is* Satisfiable?

- $\{p \vee \neg q, \neg q \vee \neg r, q\}$

- $\{p \vee \neg q, \neg q \vee \neg r, q, p, \neg r\}$

- Closure: There are no other resolvents, yet $\perp$ has not been derived.

Resolution Algorithm Invariant

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

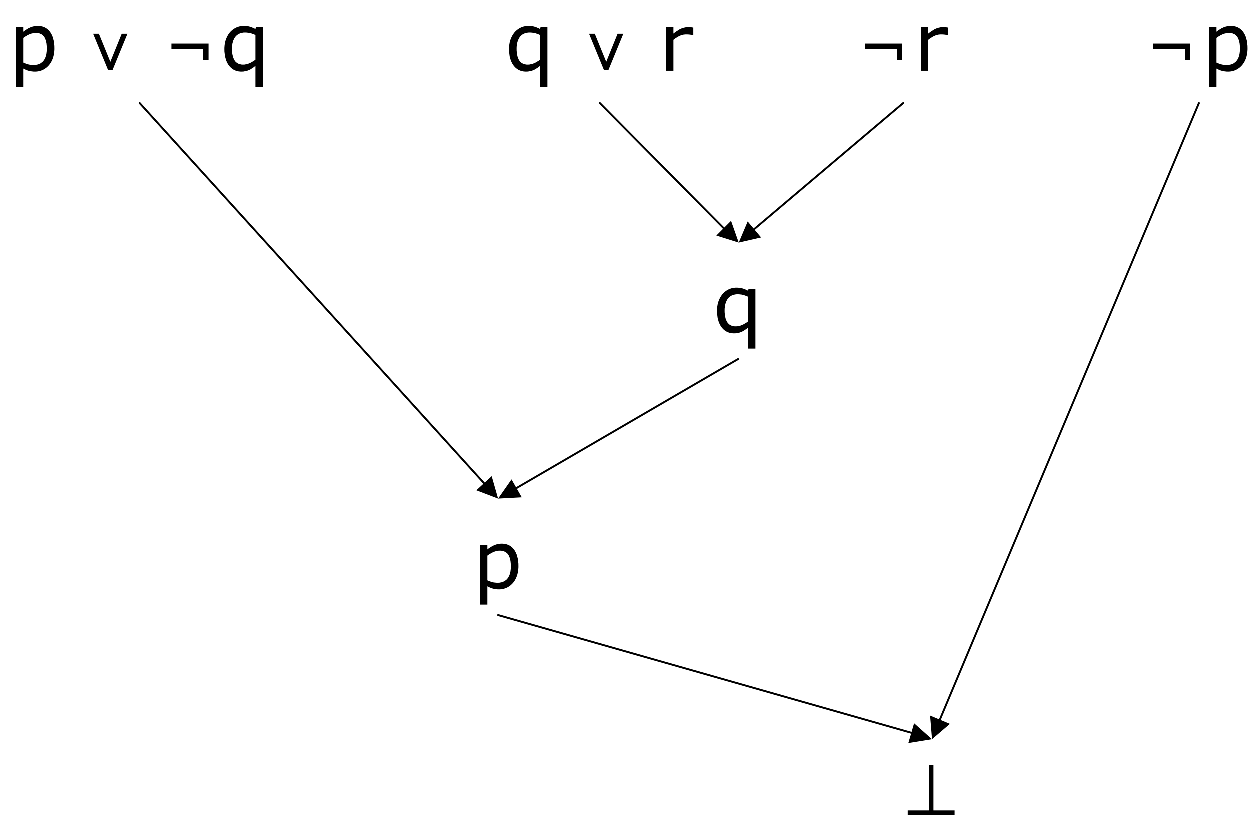
Resolution Algorithm Refinement 1

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

  S := reduce(S);
  T := reduce(resolveall(S));

  while($\perp \notin S \wedge \neg(T \subseteq S)$)
  S := $S \cup T$;
  T := reduce(resolveall(S));

Resolution as a Tree



Resolution as a "Proof"

1.	$p \vee \neg q$	Premise
2.	$q \vee r$	Premise
3.	$\neg r$	Premise
4.	$\neg p$	Premise
5.	q	Resolution 2, 3
6.	p	Resolution 1, 5
7.	$\perp$	Resolution 6, 4

Try these:

- $\{\neg p \vee \neg q \vee \neg r,$
  $\neg q \vee r,$
  $q \vee s,$
  $\neg s,$
  $p,$
  $\}$

- $\{p \vee \neg q \vee r,$
  $q \vee \neg r,$
  $\neg p\}$

Sometimes a DAG is more appropriate than a tree for showing all options

$$
\begin{tikzcd}
p \vee q \arrow[d] \arrow[drrr] & p \vee \neg q \arrow[dl] \arrow[d] & \neg p \vee \neg q \arrow[dl] \arrow[d] & \neg p \vee q \arrow[dl] \arrow[d] \\
p \arrow[dr] & \neg q \arrow[dr] & \neg p \arrow[dl] & q \arrow[dl] \\
 & \perp & \perp &
\end{tikzcd}
$$

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 := reduce(S);
  T := reduce(resolveall(S, S)) - S;
  while($\perp \notin S \wedge T \neq \emptyset$)
          S := $S \cup T$;
          T := reduce(resolveall(S, T)) - S;

- resolveall(S, T)

$$= \{\varphi \vee \psi \mid (p \vee \psi) \in S \wedge (\neg p \vee \psi) \in T\}$$
$$\cup \{\varphi \vee \psi \mid (p \vee \psi) \in T \wedge (\neg p \vee \psi) \in S\}$$

Selective Resolution

- Rather than resolving all pairs of clauses, try to pick pairs that will produce $\perp$ 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:
  - $p \vee q$ in clausal form is $\{p \vee q\}$.
  - $p \wedge q$ in clausal form is $\{p, q\}$.
  - $p \rightarrow q$ in clausal form is $\{\neg p \vee q\}$.
- In general, to get clausal form:
  - Replace $\varphi \rightarrow \psi$ with $(\neg\varphi \vee \psi)$.
  - Push $\neg$ toward proposition symbols:
    - Replace $\neg(\varphi \wedge \psi)$ with $(\neg\varphi \vee \neg\psi)$.
    - Replace $\neg(\varphi \vee \psi)$ with $(\neg\varphi \wedge \neg\psi)$.
    - Replace $\neg\neg\varphi$ with φ.
  - Distribute $\vee$ inward:
    - Replace $\chi \vee (\varphi \wedge \psi)$ with $(\chi \vee \varphi) \wedge (\chi \vee \psi)$.
    - Replace $(\varphi \wedge \psi) \vee \chi$ with $(\varphi \vee \chi) \wedge (\psi \vee \chi)$.

Example of Conversion to Clauses

- $\neg(p \to (\neg q \wedge (r \wedge \neg s)))$
- $\neg(\neg p \vee (\neg q \wedge (r \wedge \neg s)))$
- $\neg\neg p \wedge \neg(\neg q \wedge (r \wedge \neg s))$
- $p \wedge \neg(\neg q \wedge (r \wedge \neg s))$
- $p \wedge (\neg\neg q \vee \neg(r \wedge \neg s))$
- $p \wedge (q \vee \neg(r \wedge \neg s))$
- $p \wedge (q \vee \neg r \vee \neg\neg s)$
- $p \wedge (q \vee \neg r \vee s)$
- $\{p,\quad q \vee \neg r \vee s\}$

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

p	q	r	value
F	F	F	F
F	F	T	F
F	T	F	T
F	T	T	T
T	F	F	T
T	F	T	F
T	T	F	T
T	T	T	T

p’q’r’ ∨ p’q’r ∨ pq’r: negate to get {p∨q∨r, p∨q∨¬r, ¬p∨q∨¬r}

Check CNF from Truth Table

p	q	r	value	p∨q∨r	p∨q∨¬r	¬p∨q∨¬r	conj.
F	F	F	F	F	T	T	F
F	F	T	F	T	F	T	F
F	T	F	T	T	T	T	T
F	T	T	T	T	T	T	T
T	F	F	T	T	T	T	T
T	F	T	F	T	T	F	F
T	T	F	T	T	T	T	T
T	T	T	T	T	T	T	T

p′q′r′ ∨ p′q′r ∨ pq′r: negate to get {p∨q∨r, p∨q∨¬r, ¬p∨q∨¬r}

Common Special Case to Clause Set

- Often we want to prove a sequent such as:
  - $\varphi_{11} \wedge \varphi_{12} \wedge \dots \wedge \varphi_{1m1} \rightarrow \psi_1$,
    $\varphi_{21} \wedge \varphi_{22} \wedge \dots \wedge \varphi_{2m2} \rightarrow \psi_2$,
    . . .
    $\varphi_{n1} \wedge \varphi_{n2} \wedge \dots \wedge \varphi_{nmn} \rightarrow \psi_n$
    $\vdash \chi_1 \wedge \chi_2 \wedge . . . \wedge \chi_p$
    where each symbol represents a literal.
- This can be done by showing that the following clause set is **unsatisfiable**:

$$
\begin{aligned}
\{&\neg\varphi_{11} \vee \neg\varphi_{12} \vee \dots \vee \neg\varphi_{1m1} \vee \psi_1, \\
&\neg\varphi_{21} \vee \neg\varphi_{22} \vee \dots \vee \neg\varphi_{1m2} \vee \psi_2, \\
&. . . \\
&\neg\varphi_{n1} \vee \neg\varphi_{n2} \vee \dots \vee \neg\varphi_{nmn} \vee \psi_n, \\
&\neg\chi_1 \vee \neg\chi_2 \vee . . . \vee \neg\chi_p\}
\end{aligned}
$$

Horn Clauses

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

  or
  - $\neg\varphi_1 \vee \neg\varphi_2 \vee \dots \vee \neg\varphi_m$

- Horn clause are the basis of the Prolog language, where:

$$\neg\varphi_1 \vee \neg\varphi_2 \vee \dots \vee \neg\varphi_m \vee \psi$$

  is written

$$\psi \text{ :- } \varphi_1, \varphi_2, \ldots \varphi_m.$$

Prolog uses a special form of resolution to do its work (SLD resolution)

- $\{p \vee \neg r \vee \neg s,$
  $r \vee \neg q,$
  $s \vee \neg q,$
  $q,$
  $\neg p,$
  $\}$

  becomes

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

```
| ?- [user].
| p :- r, s.
| r :- q.
| s :- q.
| q.
 ^D (eof)

| ?- p.
yes
```

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 ∀-quantified.
  - The variables of each clause are **independent** from the other clauses; even if they are they 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

- $\neg$man(X) $\vee$ mortal(X)
- man(socrates)
- $\neg$mortal(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

$$\neg q(X, X) \vee p(X)$$
$$\neg p(X) \vee r(X, Y)$$

- For one thing, the identity of the variables is independent in each.
- For another, the arguments are generally **terms**, not just simple variables:

$$\neg q(X, X) \vee p(f(X))$$
$$\neg p(X) \vee r(g(X), c)$$

What Resolution Must Do

- Suppose we have derived three formulas (where c is a constant symbol):
  - $p(c)$
  - $\forall X\ (p(X) \rightarrow q(f(X))$
  - $\forall X\ (q(X) \rightarrow r(X, g(X))$

- 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
  - $\{p(c),\quad \neg p(X) \vee q(f(X)),\quad \neg q(X) \vee r(X, g(X))\}$

- 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←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 $X \leftarrow g(Y)$ vs. $X \leftarrow Y$.
- 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 $X \leftarrow g(X)$, $Y \leftarrow W$.
- **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

Term 1	Term 2	Unifiable?
p(X, X)	p(f(Y), f(Z))	
p(X, X)	p(f(Y), g(Y))	
p(X, Y)	p(Z, f(Z))	
p(X, f(X))	p(g(Y), W)	
p(X, f(X))	p(f(Y), Y)	

Even More Unification Examples

(assume renaming was already done; how these can arise will be seen later)

Term 1	Term 2	Unifiable?
p(X, Y)	p(f(Y), g(Z))	
p(X, f(X))	p(f(Y), Y)	
p(f(X), Y)	p(X, Y)	
p(f(X), f(X))	p(c, c)	
p(f(X), g(X))	p(Y, g(Y))	

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 τ is a term, then $\tau\beta$ denotes the result of making the substitutions β in for variables in τ, e.g.

$$\tau = p(X, g(Y, \qquad W))$$
$$\tau\beta = p(Z, g(f(Z, c), c))$$

Composing Variable Substitutions

- If β and γ are substitutions and τ is a term, then $(\tau\beta)\gamma$ denotes the result of first applying β to τ, then γ 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 τ

  $\tau(\beta\gamma) = (\tau\beta)\gamma$.

Generality of Substitutions

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

- To say that β is a "most general unifier" really means that is at least as general as any other unifier.

Find the MGU or indicate non-unifiable

Term 1	Term 2	MGU?
p(X, Y)	p(Z, Z)	
p(X, c)	p(Y, Y)	
p(f(X), Y)	p(Y, f(Z))	
p(f(X), Y)	p(X, Y)	
p(f(Z), g(X))	p(Y, g(Y))	

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(…)))