Resolution Theorem Proving

Robert Keller
March 2010

What is this?

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

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 tableau 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 set of clauses is **not satisfiable**.

Example

- Clause set:
  - $p \vee \neg q$
  - $\neg q \vee \neg r$
  - q
- This clause set is satisfiable:
  - Valuation p = T, q = T, r = F will satisfy it.

Example

- Clause set:
  - $p \vee \neg q$
  - $q \vee r$
  - $\neg p$
  - $\neg r$
- This clause set is unsatisfiable:
  - There is **no valuation** that makes all formulas T at the same time.
  - Why not?
    We'd need p = r = F, but then there is no way to set q.

Purpose

- A clause set is another way of representing a propositional formula.
- It represents a formula in conjunctive normal form:

  $\underbrace{\text{a conjunction}}_{\text{the set}}$ of $\underbrace{\text{disjunctions of literals}}_{\text{clauses}}$

Equivalence of Propositional Formulas

- Two propositional formulas are **equivalent** ($\equiv$) iff they are satisfied by the same valuations.
- Examples:
  - $p \wedge \neg q \equiv \neg(q \vee \neg p)$
  - $p \rightarrow (q \rightarrow r) \equiv (p \wedge q) \rightarrow r$

Equivalence of Clause Sets

- Two clause sets are called **equivalent** if they are satisfied by the same set of valuations.
- In particular, if two clause sets are equivalent, they are either:
  - both satisfiable, or
  - both unsatisfiable

How General is the Clausal Form?

- Claim: Every propositional formula can be represented in clausal form.
- Examples:
  - $p \vee q$ in clausal form is $\{p \vee q\}$ (one clause).
  - $p \wedge q$ in clausal form is $\{p, q\}$ (two clauses)
  - $p \rightarrow q$ in clausal form is $\{\neg p \vee q\}$ (one clause)
- What about T?
  - T in clausal form is the **empty set** of clauses: {} (no clauses)

The Empty Clause

- The **empty clause** is $\perp$, sometimes denoted by an empty box ❒.
- Observation: Any clause set containing the empty clause is unsatisfiable, because no valuation can make $\perp$ true.

  Example: $\{\neg p \vee q, p, \perp\}$

How General is the Clausal Form?

In general, to get clausal form:

1. Replace $\varphi \rightarrow \psi$ with $(\neg\varphi \vee \psi)$.
2. Push $\neg$ inward, 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.
3. 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

- $\neg(p \boxed{\rightarrow} (\neg q \wedge (r \wedge \neg s)))$ replace $\rightarrow$
- $\boxed{\neg}(\neg p \vee (\neg q \wedge (r \wedge \neg s)))$ push $\neg$ inward
- $\boxed{\neg\neg}p \wedge \neg(\neg q \wedge (r \wedge \neg s))$ delete $\neg\neg$
- $p \wedge \boxed{\neg}(\neg q \wedge (r \wedge \neg s))$ push $\neg$ inward
- $p \wedge (\boxed{\neg\neg}q \vee \neg(r \wedge \neg s))$ delete $\neg\neg$
- $p \wedge (q \vee \boxed{\neg}(r \wedge \neg s))$ push $\neg$ inward
- $p \wedge (q \vee \neg r \vee \boxed{\neg\neg}s)$ delete $\neg\neg$
- $p \wedge (q \vee \neg r \vee s)$ conjuncts are clauses
- $\{p,\ \ q \vee \neg r \vee s\}$

Clausal Form from a Truth Table

- A truth table displays a **disjunctive** normal form DNF.
- Each row is a **min-term**:
  $p_1^* \wedge p_2^* \wedge \ldots \wedge p_n^*$ where p_i^* indicates a **literal**.
- The overall truth-function is a **disjunction** of min-terms:
  $\bigvee(p_1^* \wedge p_2^* \wedge \ldots \wedge p_n^*)$ over the rows for which the result is **T**.
- The **negation** of the function is the disjunction over the rows for which the result is **F**.
- **For those rows**, negating each literal and flipping $\wedge$'s and $\vee$'s gives the **negation of the negation** of the function, i.e. the original function.
- This is the original function in conjunctive normal form.

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

DNF: $p'q'r' \vee p'q'r \vee pq'r$: negate to get $\{p \vee q \vee r,\ p \vee q \vee \neg r,\ \neg p \vee q \vee \neg r\}$

Check CNF from Truth Table

p	q	r	value	$p \vee q \vee r$	$p \vee q \vee \neg r$	$\neg p \vee q \vee \neg r$	**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

$p'q'r' \vee p'q'r \vee pq'r$: negate to get $\{p \vee q \vee r,\ p \vee q \vee \neg r,\ \neg p \vee q \vee \neg r\}$

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** of the literal.
  - **Drop** any clauses containing **a literal and its negation**. (Such clauses are equivalent to T.)
  - Replace multiple occurrences of a clause (as a **set**) with a single occurrence.

reduce example

reduce({p ∨ ¬q ∨ p,
p ∨ q ∨ ¬p ∨ q}) =

{p ∨ ¬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** ⊥.
- There is no interpretation that satisfies ⊥, much less ⊥ together with other clauses.

How Resolution Works

- Do Repeatedly:
  - From the set of clauses, pick a pair from which a **new** clause, called the "resolvent", can be created.
  - 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
  - $p \vee \varphi$
  - $\neg p \vee \psi$
- where φ and ψ are formulas (either could be empty), then the resolvent is

$$\varphi \vee \psi.$$

- p and ¬p are said to be "clashing" literals.

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.

Example of Resolvents

- Consider the clauses
  - p ∨ ¬q
  - q ∨ r ∨ ¬s
- A resolvent (based on literal q) is:
  - p ∨ r ∨ ¬s

Example of Resolvents

- Consider the clauses
  - p ∨ r
  - ¬r
- Since r and ¬r occur in different clauses, a resolvent is:
  - p

Example of Resolvents

- Consider the clauses
  - p
  - ¬p
- Since p and ¬p occur in different clauses, the resolvent is:
  - ⊥

Example of Resolvents

- Consider the clauses
  - p ∨ ¬q ∨ r
  - q ∨ ¬r ∨ ¬s
- One resolvent (based on literal q) is:
  - p ∨ r ∨ ¬r ∨ ¬s
- Another (based on literal r) is:
  - p ∨ q ∨ ¬q ∨ ¬s
- Both of these will be dropped in reducing, however.

Resolution Algorithm (Crude form)

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

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

while(⊥ ∉ S ∧ T - S ≠ ∅)
    {
    S := S ∪ T;
    T := reduce(resolveall(S));
    }
If ⊥ ∈ S, then unsatisfiable, else satisfiable.
```

- where resolveall(S) =

$$\{\varphi \vee \psi \mid (p \vee \varphi) \in S \text{ and } (\neg p \vee \psi) \in S\}$$

Example 1

- S = {p ∨ ¬q, q ∨ r, ¬p, ¬r}
- T = reduce(resolveall(S)) = {p ∨ r, ¬q, q}
- ⊥ ∉ S ∧ T - S ≠ ∅
- S = {p ∨ ¬q, q ∨ r, ¬p, ¬r, p ∨ r, ¬q, q}
- T = reduce(resolveall(S)) = {p ∨ r, ¬q, q, …, ⊥}
- Stop ⊥ ∈ S.
- S is unsatisfiable, since ⊥ ∈ S.

Example 2

- S = {p ∨ ¬q, q ∨ r, ¬p}
- T = reduce(resolveall(S)) = {p ∨ r, ¬q}
- ⊥ ∉ S ∧ T - S ≠ ∅
- S = {p ∨ ¬q, q ∨ r, ¬p, p ∨ r, ¬q}
- T = reduce(resolveall(S)) = {p ∨ r, ¬q, r}
- ⊥ ∉ S ∧ T - S ≠ ∅
- S = {p ∨ ¬q, q ∨ r, ¬p, p ∨ r, ¬q, r}
- T = reduce(resolveall(S)) = {p ∨ r, ¬q, r}
- Stop T - S = ∅.
- S is satisfiable, since ⊥ ∉ S.

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 ν is an valuation that satisfies *both* $p \vee \varphi$ and $\neg p \vee \psi$.
- Suppose $\nu(p) = T$. Then $\nu(\neg p) = F$. But since ν satisfies $\neg p \vee \psi$, $\nu(\psi) = T$.
- The case for $\nu(p) = F$ is symmetric: $\nu(\varphi) = T$.
- So $\nu(\varphi \vee \psi) = T$.

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 ν satisfies $\varphi \vee \psi$ (which contains neither p nor $\neg$p), then it must satisfy one or the other of φ or ψ.
- If ν satisfies φ then it also satisfies $p \vee \varphi$. Extend ν to ν' such that $\nu'(p) = F$. Then $\nu'(\neg p \vee \psi) = T$.
- If ν satisfies ψ, so extend ν to ν' such that $\nu'(p) = T$. Then $\nu'(p \vee \varphi) = T$.

Resolution Algorithm Loop Invariant

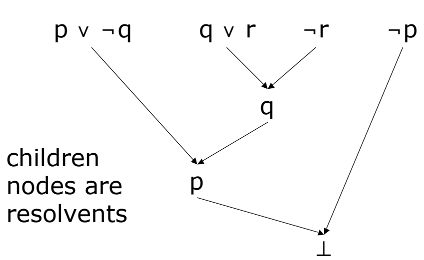
- ```
  S := reduce(S0);
  T := reduce(resolveall(S));

  while(⊥ ∉ S ∧ T - S ≠ ∅)
      {
      S := S ∪ T;
      T := reduce(resolveall(S));
      }
  If ⊥ ∈ S, then unsatisfiable, else satisfiable.
  ```
- **Invariant**:
  {S is unsatisfiable $\leftrightarrow$ 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 for n proposition symbols?

Resolution as a Tree



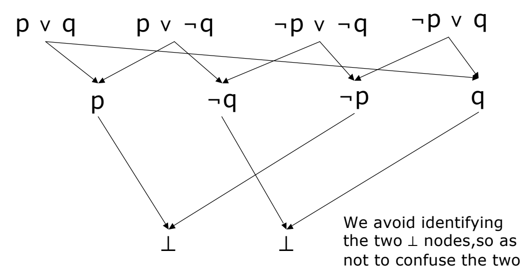
Resolution in tabular form

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 clause sets:

- $p \vee \neg q \vee r$,
  $q \vee \neg r$,
  $\neg p$
- $\neg p \vee \neg q \vee \neg r$,
  $\neg q \vee r$,
  $q \vee s$,
  $\neg s$,
  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 := 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)

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

Further Useful Optimizations

- Uncomplemented Literal Lemma (Lemma 4.9 of Ben-Ari):

  If a **literal** appears in one or more clauses, but its complement appears in no clause, then every clause containing that literal can be deleted from the set without changing satisfiability.
- **Rationale**: The literal in question can be assigned T in a satisfying interpretation, without being constrained by the other literals.

Example of Uncomplemented Literal Lemma

- $\neg p \vee q \vee r$,
  $\neg q \vee r$,
  $q \vee s$,
  $\neg s$,
  p
- r occurs only uncomplemented.
- The clause set is unsatisfiable iff the following set is:
- $q \vee s$,
  $\neg s$,
  p

Further Useful Optimizations

- **Unit Clause Lemma** (Lemma 4.11 of Ben-Ari):

  If a **unit** cause (clause with only one literal L) exists within the set, the following operation may be performed without affecting satisfiability:

  - Remove **all** clauses containing L.
  - Remove the complement of L from all remaining clauses.
- **Rationale**: The literal in question **must** be assigned T in a satisfying interpretation. Hence all clauses containing it will be T and contribute nothing to the set. Likewise, its complement must be assigned F, and contribute nothing to the individual clauses.

Example of Unit Clause Lemma

- $\neg p \vee q \vee r$,
  $\neg q \vee r$,
  $q \vee s$,
  $\neg s$,
  p
- $\neg s$ is a unit clause. The complement of $\neg s$ is s.
- The clause set is unsatisfiable iff the following set is:
- $\neg p \vee q \vee r$,
  $\neg q \vee r$,
  q,
  p

Further Useful Optimizations

Subsumption Lemma (Lemma 4.15 of Ben-Ari):

- One clause **subsumes** another if the former's literals are a **subset** of the latter's.
- If one clause of a set subsumes another, the subsumed clause can be dropped from the set.
- **Rationale**: If C subsumes D, then any interpretation satisfying C must also satisfy D (because the literals are disjoined). Thus the satisfiability of the set of clauses is unaffected if D is removed.

Example of Subsumption Lemma

- $\neg p \vee q \vee r$,
  $\neg p \vee r$,
  $p \vee \neg r$
- The second clause subsumes the first.
- The clause set is unsatisfiable iff the following set is:
- $\neg p \vee r$,
  $p \vee \neg r$

Common Special Case of Clause Set

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

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

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).

Strategic Optimizations

- **Unit-Preference**: If possible, chose to resolve with unit clauses. These reduce the size of resulting clauses.
- **Set-of-Support**: Divide the clauses into two sets:
  - A known-satisfiable subset.
  - Other
- Always resolve with an "other" or a clause derived from one.
- These clause are called the "set of support" (SOS).

Set-of-Support

- Showing that the following clause set is **unsatisfiable**:

$\{\neg\varphi_{11} \vee \neg\varphi_{12} \vee \ldots \vee \neg\varphi_{1m1} \vee \psi_1,$
$\neg\varphi_{21} \vee \neg\varphi_{22} \vee \ldots \vee \neg\varphi_{1m2} \vee \psi_2,$
$\ldots$
$\neg\varphi_{n1} \vee \neg\varphi_{n2} \vee \ldots \vee \neg\varphi_{nmn} \vee \psi_n,$ — Satisfiable "axioms"

$\neg\chi_1 \vee \neg\chi_2 \vee \ldots \vee \neg\chi_p\}$ — Set of support

Horn Clauses

- A Horn clause is one in which there is **at most one non-negated** literal:
  - $\neg\varphi_1 \vee \neg\varphi_2 \vee \ldots \vee \neg\varphi_m \vee \boxed{\psi}$ (**one** non-negated)

  or
  - $\neg\varphi_1 \vee \neg\varphi_2 \vee \ldots \vee \neg\varphi_m$ (**no** non-negated)
- Horn clause are the basis of the **Prolog** language, where:

  $\neg\varphi_1 \vee \neg\varphi_2 \vee \ldots \vee \neg\varphi_m \vee \boxed{\psi}$

  is written

  ψ :- $\varphi_1, \varphi_2, \ldots \varphi_m.$

  If m = 0, then we just write

  $\psi.$

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

- {p ∨ ¬r ∨ ¬s,
  r ∨ ¬q,
  s ∨ ¬q,
  q,
  ¬p,
  }

  Non-negated literals in red.

  becomes

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

Dialog with Prolog:

```
consult(user).
p :- r, s.
r :- q.
s :- q.
q.
 ^D

| ?- 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 thus **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

- Clause set {p(X), q(X, Y), ¬q(X, X) ∨ p(X)} stands for the conjunction
- $\forall X\ p(X)$
  $\wedge \forall X\ \forall Y\ q(X, Y)$
  $\wedge \forall X\ \forall Y\ (\neg q(X, X) \vee p(X))$

  which is the same as
- $\forall X_1\ p(X_1)$
  $\wedge \forall X_2\ \forall Y_2\ q(X_2, Y_2)$
  $\wedge \forall X_3\ \forall Y_3\ (\neg q(X_3, X_3) \vee p(X_3))$

  i.e. the clause set
- $\{p(X_1), q(X_2, Y_2), \neg q(X_3, X_3) \vee p(X_3)\}$

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

- ¬man(X) ∨ mortal(X)
- man(socrates)
- ¬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

  ¬q(X, X) ∨ p(X)
  ¬p(X) ∨ 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:
  ¬q(X, X) ∨ p(f(X))
  ¬p(X) ∨ r(g(X), c)

Example of 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)
  - ∀X (p(X) → q(f(X))
  - ∀X (q(X) → r(X, g(X))
- is
  - {p(c), ¬p(X) ∨ q(f(X)), ¬q(X) ∨ r(X, g(X))}

- Resolution has 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 [c/X] (c for X) makes them identical.
- Consider q(c, d), q(X, X) (c and d are constants).
- These terms are **not unifiable**, since distinct **constant symbols do not unify**. There is no substitution that will make them identical.

Literals from Different Clauses

- Remember that variables are not identified 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.
- This is called "renaming apart" the clauses.

Literals from Different Clauses

- Consider p(X, f(a)) vs. p(g(Y), f(X))
- These might appear not to unify, since we would have a conflict [g(Y)/X] vs. [a/X].
- However, if we **rename** the variables in the second clause we get:
  p(X, f(a)) vs. p(g(Z), f(W)).
- These unify, using [g(Z)/X], [a/W].
- **Note**: Renaming apart 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

Term 1	Term 2	Unifiable?
p(X, Y)	p(f(Z), g(Z))	
p(X, f(X))	p(f(Z), U)	
p(f(X), g(X))	p(f(U), U)	
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
  [f(c)/X, c/Y].
- However, this would **not** be the most general, since we could leave Y as a variable:
  [f(Z)/X]
  and each of the original literals would unify with this.

Notation for Variable Substitutions

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

  β = [Z/X, f(Z, c)/Y, c/W]

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

  if τ = p(X, g(Y, W))
  then $\tau\beta$ = 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.

  τ = p(X, g(Y, W)) literal
  β = {Z/X, f(Z, c)/Y, c/W} sub
  γ = {V/Z} sub
  $(\tau\beta)\gamma$ = p(V, g(f(V, c), c))

- The **composition** $\beta\gamma$ **of substitutions** β and γ is the substitution such that for **all** terms τ
  $\tau(\beta\gamma) = (\tau\beta)\gamma$
  e.g. {V/X, f(V, c)/Y, c/W} above

Generality of Substitutions

- Substitution β is **as general as** substitution ν if there is a γ such that $\nu = \beta\ \gamma$.

- To say that β is a "most general unifier" means that is as general as *any* 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(W, f(Z))	
p(f(X), Y)	p(Z, 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 bypasses 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(…)))

MGU Algorithm (Martelli & Montanari)

- **Input**: Two terms, or two atoms, τ_1, τ_2, already renamed apart.
- **Output**: Either the most general unifier for τ_1, τ_2, or "not unifiable".
-

```
S := {[τ1, τ2]};                                  // functions as a sort-of stack
μ := the empty substitution;
while( S ≠∅ )
    remove a pair [L, R] from S;                          // pop    case
    if( L = R )                                                     (1)
            do nothing;
    else if( L=f(s1, s2, …, sn) and R=f(t1, t2, …, tn) )  // same f, n (2)
            S := S ∪ {[s1, t1], [s2, t2],… [sn, tn]};          // pushes
    else if( L = x where x is a variable not occurring in R)        (3)
             μ := μ[R/x];                                     // composing
            S := applytoallpairs([R/x],  S);
    else if( R = x where x is a variable not occurring in L)        (4)
            μ := μ[L/x];
            S := applytoallpairs([L/x],  S);
    else return "not unifiable";                                    (5)
return μ as the MGU;
```

Intuitive Unification

- Remember when two things **don’t** unify:
  - Distinct constant symbols don’t unify.
  - Terms with outermost function symbols that are distinct don’t unify.
  - A term with an outermost function symbol doesn’t unify with a constant.
  - Two terms with the same outermost function symbol don’t unify if some of their arguments don’t pairwise unify.
- Remember that substitutions are **cumulative** during unification.

Example

- p(X, f(X)) vs. p(Y, f(Y)) initial
- S := {[p(X, f(X)), p(Y, f(Y))]}
- μ := []
- Remove [p(X, f(X)), p(Y, f(Y))] case 2
- S := {[X, Y], [f(X), f(Y)]}
- Remove [X, Y] case 3
- μ := [Y/X]; S := {[f(Y), f(Y)]}
- Remove [f(Y), f(Y)] case 1
- S := {}
- Result: unifiable with mgu [Y/X]

Diagrammatically

- p(X, f(X))
| ↑ substitution [Y/X]
p(Y, f(Y))
- p(Y, f(Y))
| | | |
p(Y, f(Y))

Example

- p(X, f(X)) vs. p(f(Y), Y) initial
- S := {[p(X, f(X)), p(f(Y), Y)]}
- μ := {}
- Remove [p(X, f(X)), p(f(Y), Y)] case 2
- S := {[X, f(Y)], [f(X), Y]}
- Remove [X, f(Y)] case 3
- μ := [f(Y)/X]; S := {[f(f(Y)), Y]}
- Remove [f(f(Y)), Y] case 5
- Result: not unifiable

Diagrammatically

- p(X, f(X))
| ↑↓ substitution [f(Y)/X]
p(f(Y), Y)
- p(f(Y), f(f(Y)))
| | ↓ occur check fails, not unifiable
p(f(Y), Y)

Example

- p(X, g(Z), X) vs. p(f(Y), Y, W) initial
- S := {[p(X, g(Z), X), p(f(Y), Y, W)]}
- μ := {}
- Remove [p(X, g(Z), X), p(f(Y), Y, W)] case 2
- S := {[X, f(Y)], [g(Z), Y], [X, W]}
- Remove [X, f(Y)] case 3
- μ := [f(Y)/X]; S := {[g(Z), Y], [f(Y), W]}
- Remove [g(Z), Y] case 4
- μ := [f(g(Z))/X, g(Z)/Y]; S := {[f(g(Z)), W]}
- Remove [f(g(Z)), W] case 4
- μ := [f(g(Z))/X, g(Z)/Y, f(g(Z))/W]; S := {}
- Result: unifiable with
mgu [f(g(Z))/X, g(Z)/Y, f(g(Z))/W]

Diagrammatically

- p(X, g(Z), X) vs.
  | ↑ substitution [f(Y)/X]
  p(f(Y), Y, W)
- p(f(Y), g(Z), f(Y)) vs.
  | || ↓ substitution [g(Z)/Y, f(g(Z))/X]
  p(f(Y), Y, W)
- p(f(g(Z)), g(Z), f(g(Z))) vs.
  | ||| | ↓ substitution [f(g(Z))/W, g(Z)/Y, f(g(Z))/X]
  p(f(g(Z)), g(Z), W)
- p(f(g(Z)), g(Z), f(g(Z))) vs.
  | ||| | | substitution [f(g(Z))/W, g(Z)/Y, f(g(Z))/X]
  p(f(g(Z)), g(Z), f(g(Z)))

Checking Unifiability with Prolog

- As long as there are no occur-check violations, can use = to test.

```
bash-3.2$ /opt/local/bin/swipl
Welcome to SWI-Prolog

?- p(X, g(Z), X) = p(f(Y), Y, W).

X = f(g(Z)),
Y = g(Z),
W = f(g(Z))
```

Try These

τ_1	τ_2	mgu (or not unifiable)
p(X, f(X), d)	p(c, f(c), Y)	
p(f(g(X)), g(Z))	p(f(Y), Y)	
p(f(g(X)), Z)	p(g(Y), Y)	
p(f(g(X)), X)	p(f(g(h(Z))), h(Z))	

Resolving Predicate Calculus Clauses

- Resolvable clauses must contain literals with the **same predicate** symbol but of **opposite sign** (one negated, the other not).
- Pick two such literals, one from each clause.
- Rename the clauses apart.
- Determine whether the literals are unifiable, with mgu μ. If they are, apply μ to ***all*** literals in both clauses. If not, the clauses don't resolve on these particular literals.
- In the modified clauses, remove ***all*** instances of the modified literals used in unification, and form the disjunction of the remaining literals.

Complete Predicate Resolution Process

- The process is the same as for the propositional case, except that we have to rename variables, then unify literals prior to resolution and apply the mgu to all literals in the two clauses, before obtaining the resolvent.

Example of Predicate Resolution

- Clauses:
  - ¬man(X) ∨ mortal(X)
  - man(socrates)
  - ¬mortal(socrates)

¬man(X) ∨ mortal(X)   ¬mortal(socrates)

[socrates/X]

¬man(socrates)   man(socrates)

⊥

Example Resolving Predicate Clauses

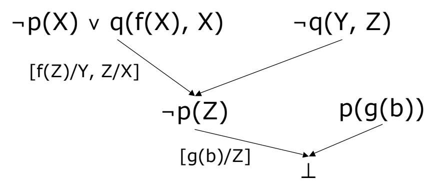
- clause 1: p(X, g(Z), X) ∨ q(X, h(Z))
- clause 2: ¬p(f(Y), Y, W) ∨ r(f(Y), g(W))
- These are already renamed apart.

- The first literals of each unify with mgu
  [f(g(Z))/X, g(Z)/Y, f(g(Z))/W]

- Apply the mgu to both clauses:
- clause 1′: p(f(g(Z)), g(Z), f(g(Z))) ∨ q(f(g(Z)), h(Z))
- clause 2′: ¬p(f(g(Z)), g(Z), f(g(Z))) ∨ r(f(g(Z)), g(f(g(Z))))
- Remove the instances of the unified atoms and form the disjunction.
- Resolvent: q(f(g(Z)), h(Z)) ∨ r(f(g(Z)), g(f(g(Z))))

Example of Predicate Resolution

- Clauses:
  - ¬p(X) ∨ q(f(X), X))
  - p(g(b))
  - ¬q(Y, Z)



Try This Set

1. ¬e(X) ∨ q(X) ∨ s(X, f(X))
2. ¬e(X) ∨ q(X) ∨ r(f(X))
3. p(a)
4. e(a)
5. ¬s(a, Y) ∨ p(Y)
6. ¬p(X) ∨ ¬q(X)
7. ¬p(X) ∨ ¬r(X)

Example where there is more than one instance of a literal to remove

- q(b, X) ∨ p(X) ∨ q(b, a)
- ¬q(Y, a) ∨ p(Y)
- These are already renamed apart.

- unify q(b, X) with ¬q(Y, a)
- mgu is [a/X, b/Y]

- Modified clauses:
- q(b, a) ∨ p(a) ∨ q(b, a)
- ¬q(b, a) ∨ p(b)

- There are **two** instances of q(b, a) in the first clause; both are removed in resolving.
- Resolvent: p(a) ∨ p(b)

Binary Resolution and Factoring

- What we have seen so far is "binary" resolution — unifying two literals to achieve a resolvent.
- In general, binary resolution alone might not be enough.
- We might need to "factor" two or more literals in the same clause to make progress.

Factoring

- Within a single clause, two or more literals of the same sign can be unified so that the resulting literals can be collapsed into one.
- The resulting clause is called a **factor** of the original.
- The factor (with all variables quantified) is logically implied by the more-general original (with all variables quantified).

Factoring Example

- Consider the clause:

  P(x) v P(f(y)) v ¬Q(x)

- The first two literals can be unified using the substitution [f(y)/x].
- The resulting factor is:

  P(f(y)) v ¬Q(f(y))

- (∀x ∀y (P(x) v P(f(y)) v ¬Q(x))) → ∀y (P(f(y)) v ¬Q(f(y))) is valid

Use of Factoring

- Suppose our clause set includes:
  P(x) v P(f(y)) v ¬Q(x)
  ¬P(f(a))
- With binary resolution, we'd get the resolvent: P(x) v ¬Q(x).
- If we **first factor**, to get P(f(y)) v ¬Q(f(y)) as on the previous slide, we can get a resolvent ¬Q(f(a)), which is better.

Full Resolution of Two Clauses

- Binary resolution of the clauses.
- Binary resolution of one clause with a factor of the other.
- Binary resolution of factors of both clauses.

Case Where Factoring is Necessary

- P(x) v P(y)
- ¬P(a) v ¬P(b)
- Without factoring, generate:
- P(y) v ¬P(b)
- P(x) v ¬P(a)
- and more similar clauses,
  but never the empty clause.

Case Where Factoring is Necessary

- P(x) v P(y)
- ¬P(a) v ¬P(b)
- With factoring, generate:
- ¬P(b)
- ⊥

Clausal Form for Predicate Logic

- Often, we'll want to prove a sequent of the form
  - ∀x ∀y (…)
  - ∀x ∀y (…)

    |- …
  - For premises of the form ∀x ∀y (…) where … has no quantifiers, we can just drop the quantifiers.
  - We need to **negate** the conclusion.

Mushroom Example

1. Every fungus is a mushroom or a toadstool.
2. Every boletus is a fungus.
3. All toadstools are poisonous.
4. No boletus is a mushroom.
5. Specimen b is a boletus.
6. Therefore: Specimen b is poisonous.

Mushroom Example

1. $\forall X$ fungus(x) → (mushroom(X) ∨ toadstool(X))
2. $\forall X$ boletus(X) → fungus(X)
3. $\forall X$ toadstool(X) → poisonous(X)
4. $\forall X$ boletus(X) → ¬mushroom(X)
5. boletus(b)
6. Therefore: poisonous(b)

Mushroom Clauses

1. ¬fungus(X) ∨ mushroom(X) ∨ toadstool(X)
2. ¬boletus(X) ∨ fungus(X)
3. ¬toadstool(X) ∨ poisonous(X)
4. ¬boletus(X) ∨ ¬mushroom(X)
5. boletus(b)
6. ¬poisonous(b) (negated conclusion)

Using Otter or Prover9

- Otter was a sophisticated resolution theorem prover developed at Argonne National Laboratory.
- It will accept input in either clausal form, or non-clausal form.
- It has been succeeded by prover9:

  http://www.cs.unm.edu/~mccune/prover9/

- There is an on-line version Otter-λ which has enhancements of Otter for the λ calculus:

  http://www.michaelbeeson.com/research/otter-lambda/

Mushroom Clauses in Otter

```
set(auto).
list(usable).

-fungus(x) | mushroom(x) | toadstool(x).

-boletus(x) | fungus(x).

-toadstool(x) | poisonous(x).

-boletus(x) | -mushroom(x).

 boletus(b).

-poisonous(b).

end_of_list.
```

Otter's Output for Mushrooms

```
---------------- PROOF ----------------

1 [] -fungus(x)|mushroom(x)|toadstool(x).
2 [] -boletus(x)|fungus(x).
3 [] -toadstool(x)|poisonous(x).
4 [] -boletus(x)| -mushroom(x).
5 [] -poisonous(b).
6 [] boletus(b).
7 [hyper,6,2] fungus(b).
8 [hyper,7,1] mushroom(b)|toadstool(b).
9 [hyper,8,3,unit_del,5] mushroom(b).
10 [hyper,9,4,6] $F.

------------ end of proof -------------
```

[...] indicates rule name and formulas used

$F indicates the null clause.

Checking Unifiability with Otter

- In contrast to Prolog,
  Otter *does* use an occur-check.

unification succeeds

```
set(auto).
set(prolog_style_variables).
list(usable).

p(X, g(Z)) | $ans(X).

-p(f(Y), Y).

end_of_list.
```

unification fails due to occur-check

```
set(auto).
set(prolog_style_variables).
list(usable).

p(X, g(X)) | $ans(X).

-p(f(Y), Y).

end_of_list.
```

(more on $ans later)

```
-------------- PROOF --------------

1 [] -p(f(A),A).
2 [] p(A,g(B))|$ans(A).
3 [binary,2.1,1.1] $ans(f(g(B))).

----------- end of proof ------------
```

binding for X

```
=========== start of search ===========

given clause #1: (wt=4) 2 [] p(A,g(A))|$ans(A).

Search stopped because sos empty.

============ end of search ============
```

Clausal Form for Predicate Logic

- Often, we'll want to prove a sequent of the form
  - $\forall x\ \forall y\ (\ldots)$,
  - $\forall x\ \forall y\ (\ldots)$
    $\vdash \forall x\ \forall y\ (\ldots)$
  - For premises of the form $\forall x\ \forall y\ (\ldots)$ where … has no quantifiers, we can just drop the quantifiers.
  - We need to **negate** the conclusion, so that will become
    $\neg \forall x\ \forall y\ (\ldots)$ which is equivalent to

    $$\exists x\ \exists y\ \neg(\ldots).$$

We cannot simply drop the quantifiers in this case!!

Clausal Form for Predicate Logic

- Consider the sequent

  $\forall y\ p(y) \vdash \forall y\ p(x)$

- The premise translates to a clause

  $p(y)$

- The conclusion is negated to become $\exists x\ \neg p(x)$.
- How do we handle this?

Skolem Constants/Functions to the Rescue!

- To get rid of the quantifier in

  $\exists x\ \neg p(x)$

  we use a trick:

  Create a **new constant**, say b (called a Skolem constant) and replace x with that:

  $\neg p(b)$

- Some thought will show that:

  There is an interpretation that satisfies $\neg p(b)$ **iff** there is one that satisfies the original formula $\exists x\ \neg p(x)$.

- Colloquially, we get to pick the value for b in finding a satisfying interpretation, just as we get to pick the value for x in $\exists x$.

Clausal Form for Predicate Logic

- Consider the sequent

  $\forall y\ p(y) \vdash \forall x\ p(x)$

- The premise translates to a clause

  $p(y)$

- The conclusion translates to a clause, where b is a Skolem function.

  $\neg p(b)$

- We are good to go!
- Resolution produces $\perp$ in 1 step.

Another Example

- Consider the sequent

  $\exists x\ \forall y\ p(x, y) \vdash \forall y\ \exists x\ p(x, y)$

- Premise clause:

  $p(b, y)$

- Conclusion clause:

  $\neg p(x, c)$

Resolution produces $\perp$ in 1 step.

Skolem Functions for the General Case

- $\forall x\ \forall y \ldots \exists v \ldots$
- v is replaced with $f(x, y, \ldots)$
- f is a **new function symbol**, the arguments of which are the $\forall$ quantified variables on the left.
- The rationale here is that "the v" that exists **depends on** $x, y, \ldots$.
- Again, there is an interpretation satisfying the original formula iff there is an interpretation satisfying the revised formula.
- We love Skolem!

Skolem with Arguments Example

- Prove: "The composition of two onto [surjective] functions is onto."
- **Represent the two functions as binary predicates**. $F(x, y)$ means y is the image of x.
- "F is onto": $\forall y\ \exists x\ F(x, y)$
- "G is onto": $\forall z\ \exists y\ G(y, z)$
- "H is the composition of F and G":

  $\forall x\ \forall y\ \forall z\ ((F(x,y) \wedge G(y,z)) \rightarrow H(x, z))$
  $\wedge\ \forall x\ \forall z\ (H(x, z) \rightarrow \exists y\ (F(x,y) \wedge G(y,z)))$
- "H is onto": $\forall z\ \exists x\ H(x, z)$

Translation to Clausal Form

- $\forall y\ \exists x\ F(x, y)$ becomes $F(f(x), y)$ [f is a Skolem function]
- $\forall z\ \exists y\ G(y, z)$ becomes $G(g(z), z)$ [g is a Skolem function]
- $\forall x\ \forall y\ \forall z\ ((F(x,y) \wedge G(y,z)) \rightarrow H(x, z))$ becomes
  $\neg F(x, y) \vee \neg G(y, z) \vee H(x, z)$
- $\forall x\ \forall z\ (H(x, z) \rightarrow \exists y\ (F(x,y) \wedge G(y,z)))$ becomes
  - $\neg H(x, z) \vee F(x, h(x, z))$ [h is a Skolem function]
  - $\neg H(x, z) \vee G(h(x, z), z)$
- $\neg \forall z\ \exists x\ H(x, z)$ becomes $\exists z \forall x\ \neg H(x, z)$, which as a clause is:
  - $\neg H(x, a)$ [a is a Skolem constant]

Resolution Proof

1. $F(f(x), y)$ Premises
2. $G(g(z), z)$
3. $\neg F(x, y) \vee \neg G(y, z) \vee H(x, z)$
4. $\neg H(x, z) \vee F(x, h(x, z))$
5. $\neg H(x, z) \vee G(h(x, z), z)$
6. $\neg H(x, a)$

---

7. $\neg F(x, y) \vee \neg G(y, a)$ from 3, 6
8. $\neg G(y, a)$ from 1, 7
9. $\perp$ from 2, 8

(3 and 4 were not needed in the proof.)

How to get a clause form in general?

- First convert the formula into "**prenex form**" (all quantifiers are outside on the left). [The parts of this form are the "prefix" and the "matrix".]
- Skolemize $\exists$ quantified variables.
- Drop $\forall$ quantifiers.
- Convert the resulting matrix to CNF.

Conversion to Prenex Form

- Replace all connectives other than $\wedge \vee \neg$ with their $\wedge \vee \neg$ counterparts.
- Push negations inward
- Pull quantifiers to the outside using the rules on the next page.

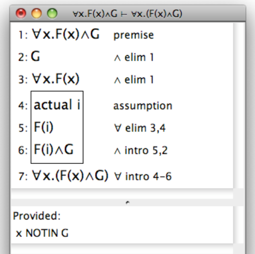
Basic Prenex Quantifier Rules

(for pulling quantifiers to the outside)

- Here $\Rightarrow$ means "replace with"
  1. $(\forall x\ F) \wedge G \Rightarrow \forall x\ (F \wedge G)$, provided x is not free in G
  2. $(\forall x\ F) \vee G \Rightarrow \forall x\ (F \vee G)$, provided x is not free in G
  3. $(\exists x\ F) \wedge G \Rightarrow \exists x\ (F \wedge G)$, provided x is not free in G
  4. $(\exists x\ F) \vee G \Rightarrow \exists x\ (F \vee G)$, provided x is not free in G
  - plus the symmetric counterparts of these rules with G part quantified instead of the F part.
  - Renaming some variables may be need to enable the rule to be applied
- Example:

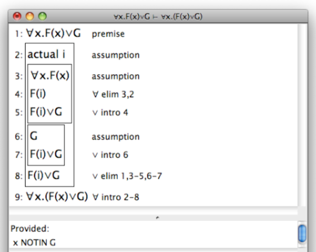
$(\exists x\ F[x]) \wedge (\forall x\ G[x]) \Rightarrow$	(by renaming second x)
$(\exists x\ F[x]) \wedge (\forall y\ G[y]) \Rightarrow$	(by rule 3, as x is not free)
$(\exists x\ (F[x] \wedge (\forall y\ G[y])) \Rightarrow$	(by rule 1 symmetric counterpart)
$\exists x\ \forall y\ (F[x] \wedge G[y])$	

Justification of Rules Using Natural Deduction



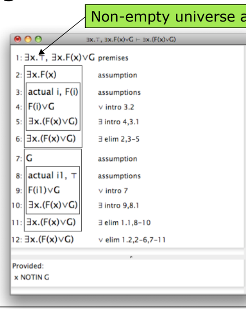
Proviso is introduced by unifying G with G[x\i] (sub i for x) in JAPE.

Justification of Rules Using Natural Deduction

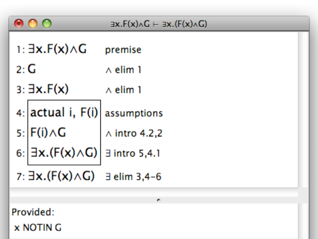


Justification of Rules Using Natural Deduction

Non-empty universe assumption, needed in 7-11



Justification of Rules Using Natural Deduction



Example of Prenex Conversion

- $\forall x\ \forall y((\exists z(p(x, z) \wedge p(y, z)) \boxed{\rightarrow} \exists u\ q(x, y, u))$
- $\forall x\ \forall y(\boxed{\neg}(\exists z(p(x, z) \wedge p(y, z)) \vee \exists u\ q(x, y, u))$
- $\forall x\ \forall y((\forall z\boxed{\neg}(p(x, z) \wedge p(y, z)) \vee \exists u\ q(x, y, u))$
- $\forall x\ \forall y((\boxed{\forall z}\ (\neg p(x, z) \vee \neg p(y, z)) \vee \exists u\ q(x, y, u))$
- $\forall x\ \forall y\ \forall z(\neg p(x, z) \vee \neg p(y, z) \vee \boxed{\exists u}\ q(x, y, u))$
- $\underbrace{\forall x\ \forall y\ \forall z\ \exists u}_{\text{prefix}}\underbrace{(\neg p(x, z) \vee \neg p(y, z) \vee q(x, y, u))}_{\text{matrix}}$

Completion of Conversion

- ∀x ∀y ∀z ∃u(¬p(x, z) ∨ ¬p(y, z) ∨ q(x, y, u))
- Skolemize as f(x, y, z) and drop ∀x ∀y ∀z
- ¬p(x, z) ∨ ¬p(y, z) ∨ q(x, y, f(x, y, z))

Example: Group Theory Clauses

- f is the group operation, e is the equality predicate
- ∀x ∀y ∀x e(f(x, f(y, z)), f(f(x, y), z))
  becomes
  e(f(x, f(y, z)), f(f(x, y), z))
- ∀x e(f(x, u), x)
  becomes
  e(f(x, u), x)
- ∀x e(f(x, i(x)), u)
  becomes
  e(f(x, i(x)), u)

Example: Equality Theory Clauses

- We need to axiomatize equality predicate e, e.g.
- ∀x e(x, x)
  becomes
  e(x, x)
- ∀x ∀y ∀u ∀v (e(x, y)∧ e(v, w)) → e(f(x, v), f(y, w))
  becomes
  ¬e(x, y) ∨ ¬e(u, v) ∨ e(f(x, v), f(y, w))
- ∀x ∀y e(x, y) → e(y, x)
  becomes
  ¬e(x, y) ∨ e(y, x)

  etc.

Example of Group Theory Clauses with Negated Conclusion

1. e(f(x, f(y, z)), f(f(x, y), z))
2. e(f(x, u), x)
3. e(f(x, i(x)), u)
4. e(x, x)
5. ¬e(x, y) ∨ ¬e(v, w) ∨ e(f(x, v), f(y, w))
6. ¬e(x, y) ∨ e(y, x)
7. ¬e(x, y) ∨ ¬e(y, z) ∨ e(x, z)

8. ¬e(i(i(b)), b)

   This is to show that ∀x e(i(i(x)), x):

   "In a group, the inverse of the inverse of an element is the element itself."

Equality in Otter, Paramodulation

- Otter has a built-in equality, so the approach illustrated in the previous example is not generally necessary.
- Equality can be handled, for example, by the "paramodulation" rule, which essentially captures one of the two ND rules for equality:

$$\frac{\alpha \vee (s = t) \qquad \beta \vee \gamma[r]}{(\alpha \vee \beta \vee \gamma[t])\theta}$$

$\gamma[r]$ is a literal containing term r

$\theta = \text{unify}(s, r)$