Resolution Theorem Proving

Robert Keller
March 2013

What is this?

- Resolution is a special kind of theorem proving used in:
  - Automated theorem proving and reasoning, where the goal is complete automation
  - Databases and Answer Extraction (AI)
  - Prolog language (a subset of general resolution)
- Resolution forms a complete proof system for refutation.

History

Alan Robinson

- Resolution was introduced by Prof. J. Alan Robinson in 1965 at Syracuse U.

- There were previous hints at it by Dag Prawitz (1960, for the function-free case) and Herbrand (1930's).

Dag Prawitz

Recent

- Resolution is used for fully automated theorem-proving and reasoning systems.

- http://en.wikipedia.org/wiki/Automated_theorem_proving

- http://www.cs.miami.edu/~tptp/cgi-bin/SystemOnTPTP

How resolution works

- A logic formula is first negated, then converted into “clausal form”.
  (Significant logic is wired into this step.)
- In clausal form, quantifiers have been eliminated.
- The clausal form is proved by **refutation**, i.e. showing that its negation is unsatisfiable.

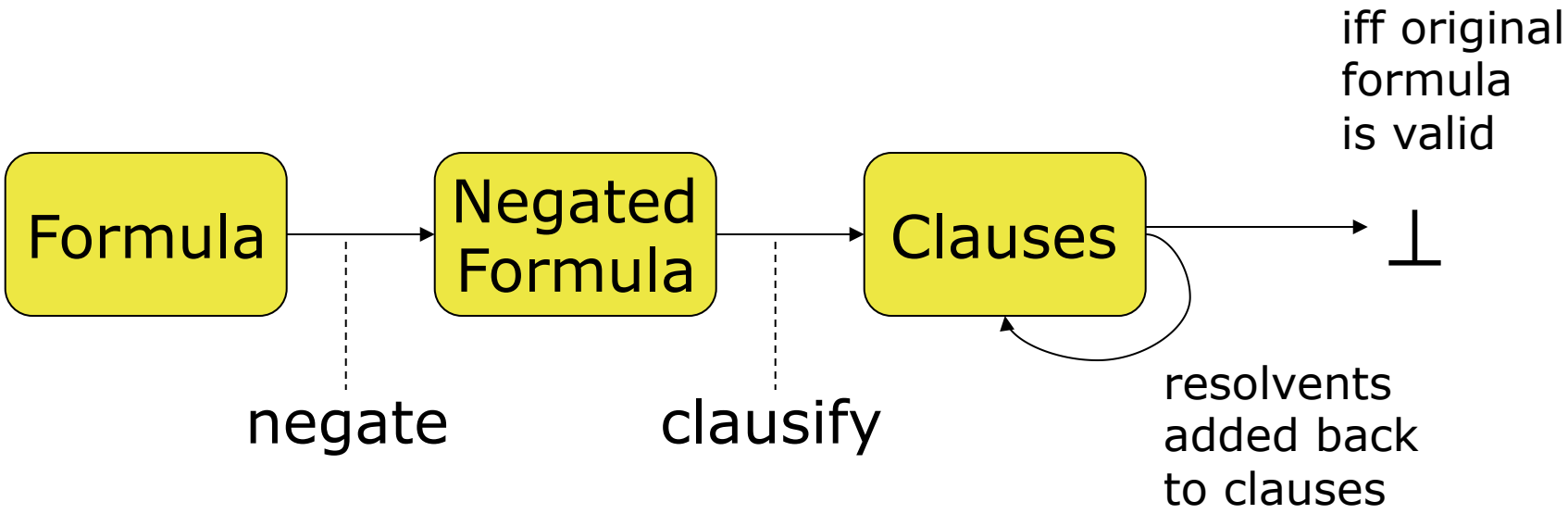
Two Types of Resolution

- Predicate calculus resolution:
  - Our main objective
- Propositional resolution:
  - Needed to understand predicate resolution
  - Big role in algorithms and complexity theory (NP completeness, for example)

Propositional Form 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**.

Resolution Schematic



Example Clause Set

- 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

- 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 in order to satisfy all clauses,
    but then there is no way to set q so that all clauses are satisfied.

Conjunctive Normal Form (CNF)

- A clause set is another way of representing a propositional formula.

- A formula is in conjunctive normal form iff it is a conjunction of disjunctions of literals

  (an “and” of “ors” of propositions and their negations”

CNF Example

$(p \vee \neg q \vee r) \wedge (q \vee r \vee s) \wedge (\neg s)$

Three clauses, containing 3, 3, 1 literals, respectively.

Sets

- Throughout this discussion, we want each clause to be a mathematical **set**, meaning that duplicates literals are eliminated.

- Likewise, a set of clauses (i.e. CNF) has no clause duplicated.

Example

- Clause set:
  - $p \vee \neg q$
  - $q \vee r$
  - $\neg p$
  - $\neg r$
- is representable as

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

where the $\vee$ connective is implicit within clauses, and the $\wedge$ connective is implicit among them.

Equivalence of Propositional Formulas

- Two propositional formulas are **equivalent** (≡) 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)
  - $p \leftrightarrow q$ in clausal form is $\{\neg p \vee q, p \vee \neg q\}$ (two clauses)

Extreme Cases

- ∅ or {}, the empty **set** of clauses, is equivalent to **True**

  (as True is the identity for the ∧ operation, and a set of clauses is a conjunction)

- Note: Every valuation satisfies every clause in {}, because there are no clauses to satisfy.

The Empty Clause

- The **empty clause** is $\bot$,
  sometimes denoted by an empty box $\square$,
  and is equivalent to False.

- Do not confuse the empty clause with an empty *set* of clauses {}.

- Observation: Any clause set containing the empty clause is unsatisfiable, because no valuation can make $\bot$ true.

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

Conversion to Clausal Form

- We rely on rules that can be proved in propositional logic:
  - Commutative
    - $A \wedge B \equiv B \wedge A$
    - $A \vee B \equiv B \vee A$
  - Distributive
    - $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$
    - $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$ <<< important
  - DeMorgan
    - $\neg(A \wedge B) \equiv \neg A \vee \neg B$
    - $\neg(A \vee B) \equiv \neg A \wedge \neg B$

Conversion to Clausal Form
(see also Huth & Ryan, section 1.5.1-1.5.2)

1. Replace each $\varphi \rightarrow \psi$ with $(\neg\varphi \vee \psi)$.

2. Replace each $\varphi \leftrightarrow \psi$ with $(\neg\varphi \vee \psi) \wedge (\varphi \vee \neg\psi)$.

3. 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.

4. Push $\vee$ inward toward literals:
   - 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

• ¬(p → (¬q ∧ (r ∧ ¬ s)))	replace →
• ¬(¬p ∨ (¬q ∧ (r ∧ ¬ s)))	push ¬ inward
• ¬¬p ∧ ¬(¬q ∧ (r ∧ ¬ s))	delete ¬¬
• p ∧ ¬(¬q ∧ (r ∧ ¬ s))	push ¬ inward
• p ∧ (¬¬q ∨ ¬(r ∧ ¬s))	delete ¬¬
• p ∧ (q ∨ ¬(r ∧ ¬s))	push ¬ inward
• p ∧ (q ∨ ¬r ∨ ¬¬s)	delete ¬¬
• p ∧ (q ∨ ¬r ∨ s)	conjuncts are clauses
• {p, q ∨ ¬r ∨ s}	

Example of Conversion to Clauses

• ¬(p ∧ (¬q ∨ (r ∧ ¬ s)))	push ¬ inward
• ¬p ∨ ¬(¬q ∨ (r ∧ ¬ s))	push ¬ inward
• ¬p ∨ (¬¬q ∧ ¬(r ∧ ¬ s))	delete ¬¬
• ¬p ∨ (q ∧ ¬(r ∧ ¬ s))	push ¬ inward
• ¬p ∨ (q ∧ (¬r ∨ ¬¬ s))	delete ¬¬
• ¬p ∨ (q ∧ (¬r ∨ s))	distribute ∨ over ∧
• (¬p ∨ q) ∧ (¬p ∨ (¬r ∨ s))	flatten ∨
• (¬p ∨ q) ∧ (¬p ∨ ¬r ∨ s)	
• {(¬p ∨ q), (¬p ∨ ¬r ∨ s)}	conjuncts are clauses

Try this one

- $(\neg p \wedge (\neg q \rightarrow (r \leftrightarrow s)))$

Code for CNF conversion

- Python
  http://aima-python.googlecode.com/svn/trunk/logic.py

- Prolog

http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/logic_topics/normal_forms/normal_form.html

Clausal Form from a Truth Table

- A truth table is typically thought of as displaying one of possibly many **disjunctive normal forms** (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:

  $\mathbf{V}(p_1^* \wedge p_2^* \wedge \ldots \wedge p_n^*)$ over the rows for which the result is **T**.

- If there are no rows with result T, the DNF represents False.
- Minterms are additive.

Example: Truth Table

	p	q	r	value
	F	F	F	F
	F	F	T	F
(¬p∧q∧¬r)	F	T	F	T
(¬p∧q∧r)	F	T	T	T
(p∧¬q∧¬r)	T	F	F	T
	T	F	T	F
(p∧q∧¬r)	T	T	F	T
(p∧q∧r)	T	T	T	T

A DNF: (¬p∧q∧¬r) ∨ (¬p∧q∧r) ∨ (p∧¬q∧¬r) ∨ (p∧q∧¬r) ∨ (p∧q∧r)

Clausal Form from Truth Table

- A DNF has the value **True** exactly when at least one of the disjuncts does, e.g.

  $(\neg p \wedge q \wedge \neg r) \vee (\neg p \wedge q \wedge r) \vee \ldots$

- Similarly a CNF has the value **False** exactly when at least one of the **maxterms** does, e.g.

  $(p \vee q \vee r) \wedge (p \vee q \vee \neg r) \wedge (\neg p \vee q \vee \neg r)$

- Hence we can "read off" a CNF from the *False* rows of the table by ***inverting*** the sense of the variables.

- The case where no rows are False is True.
- Maxterms are subtractive.

Example: Clausal Form from Truth Table

p	q	r	value	
F	F	F	F	(p ∨ q ∨ r)
F	F	T	F	(p ∨ q ∨ ¬r)
F	T	F	T	
F	T	T	T	
T	F	F	T	
T	F	T	F	(¬p ∨ q ∨ ¬r)
T	T	F	T	
T	T	T	T	

A CNF: (p ∨ q ∨ r) ∧ (p ∨ q ∨ ¬r) ∧ (¬p ∨ q ∨ ¬r)

Summary

- In a DNF (such as minterm form) a term (product) is trying to make the function more True. (The empty case is False).

- In CNF (such as maxterm form) a term (sum) is trying to make the function more False. (The empty case is True).

Housekeeping: 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, and do thus do not affect satisfiability of the set of clauses.)
  - 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 **empty clause** $\perp$.
- There is no valuation that satisfies $\perp$, much less $\perp$ together with other clauses.

How Resolution Works

- Do Repeatedly, until no further steps can be taken:
  - From the set of clauses, pick a pair from which a **new** clause, called the "resolvent", can be created. (Must resolve the pair to find this out.)
  - Add the resolvent to the set.
  - If $\perp$ is ever added to the set, stop. 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 clauses of both forms
  - $p \vee \varphi$
  - $\neg p \vee \psi$
- where φ and ψ are formulas (either could be empty), then the resolvent is the reduced version of

  $$\varphi \vee \psi.$$

- p and ¬p are said to be “clashing” literals.

Resolution as a Deduction Rule

$$\frac{p \vee \varphi \qquad\qquad \neg p \vee \psi}{\varphi \vee \psi}$$ (in reduced form)

where p is any proposition symbol and φ and ψ are clauses (possibly empty).

Relation to Sequent Calculus

The **optional** "Cut" Rule in sequent calculus is similar to resolution. (Gentzen proved that it was unnecessary.)

(1) $\Gamma \vdash A, \Delta$

and

(2) $\Pi, A \vdash \Lambda$

allows one to infer

(3) $\Gamma, \Pi \vdash \Delta, \Lambda.$

http://en.wikipedia.org/wiki/Cut-elimination_theorem
http://mathworld.wolfram.com/CutEliminationTheorem.html

Example of Resolvents

- Consider the clauses
  - p v **¬q** v ¬s
  - **q** v r v ¬s

- A resolvent (based on literals q, ¬q) is:
  - p v r v ¬s

Example of Resolvents

- Consider the clauses
  - p ∨ r
  - ¬r
- The 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 literals q, ¬q) is:
  - p ∨ r ∨ ¬r ∨ ¬s
- Another (based on literals r, ¬r) is:
  - p ∨ q ∨ ¬q ∨ ¬s

- Both of these would be **dropped** in reducing, however, since each contains a literal and its negation.

Resolution Algorithm

- Start with a set S of reduced clauses.
- while S does not contain $\perp$ and the following step adds something new to S:
  - Add to S the resolvent R of any two clauses such that R is not already in S.
- The original S is unsatisfiable iff $\perp$ is in S.

Unit Clauses

- A clause with exactly one literal is called a unit clause.
- The penultimate step prior to resolving to $\perp$ will be to resolve two unit clauses.
- Resolving a unit clause with a clause having $n > 0$ literals results in a clause with fewer than n literals.

Note on choice of clauses

- Preferring unit clauses is a good heuristic.

Example 1 (Highlighting unit clauses)

- S = {p ∨ ¬q, q ∨ r, **¬p**, **¬r**}
  resolve p ∨ ¬q with **¬p**, adding **¬q** to S.
- S = {p ∨ ¬q, q ∨ r, **¬p**, **¬r**, **¬q**}
  resolve q ∨ r with **¬q**, adding **r** to S.
- S = {p ∨ ¬q, q ∨ r, **¬p**, **¬r**, **¬q**, **r**}
  resolve ¬r with r adding ⊥ to S.
- Stop ⊥ ∈ S.
- The original S is unsatisfiable, as ⊥ ∈ S.

Example 2

- $S = \{p \vee \neg q,\ q \vee r,\ \neg \mathbf{p}\}$
  Resolve $p \vee \neg q$ with $\neg \mathbf{p}$, adding $\neg \mathbf{q}$ to S.

- $S = \{p \vee \neg q,\ q \vee r,\ \neg \mathbf{p}, \neg \mathbf{q}\}$
  Resolve $q \vee r$ with $\neg \mathbf{q}$, adding $\mathbf{r}$ to S.

- $S = \{p \vee \neg q,\ q \vee r,\ \neg \mathbf{p}, \neg \mathbf{q}, \mathbf{r}\}$
  Resolve $p \vee \neg q$ with $q \vee r$, adding $p \vee r$ to S.

- $S = \{p \vee \neg q,\ q \vee r,\ \neg \mathbf{p}, \neg \mathbf{q}, \mathbf{r}, p \vee r\}$
  Stop. No new resolvents are possible. The original set is satisfiable, as $\bot \notin S$.

Soundness: Any valuation satisfying both $p \vee \varphi$ and $\neg p \vee \psi$ satisfies $\varphi \vee \psi$.

- Suppose ν satisfies **both** $p \vee \varphi$ and $\neg p \vee \psi$.
- Either $\nu(p) = T$ or $\nu(p) = F$.
- If $\nu(p) = T$, then $\nu(\neg p) = F$. In order to satisfy $\neg p \vee \psi$ then, $\nu(\psi) = T$. Thus $\nu(\varphi \vee \psi) = T$.
- If $\nu(p) = F$, in order to satisfy $p \vee \varphi$, $\nu(\varphi) = T$. Thus $\nu(\varphi \vee \psi) = T$.
- Thus adding the resolvent preserves the valuations that satisfy the set of clauses.

Completeness

- Completeness is more complicated and we will not prove it here.
- We'd have to show that if a set is unsatisfiable, there is a set of resolution steps that result in the empty clause.
- For a constructive proof, see: J. Gallier, 2006: http://www.cis.upenn.edu/~cis510/tcl/resol.pdf

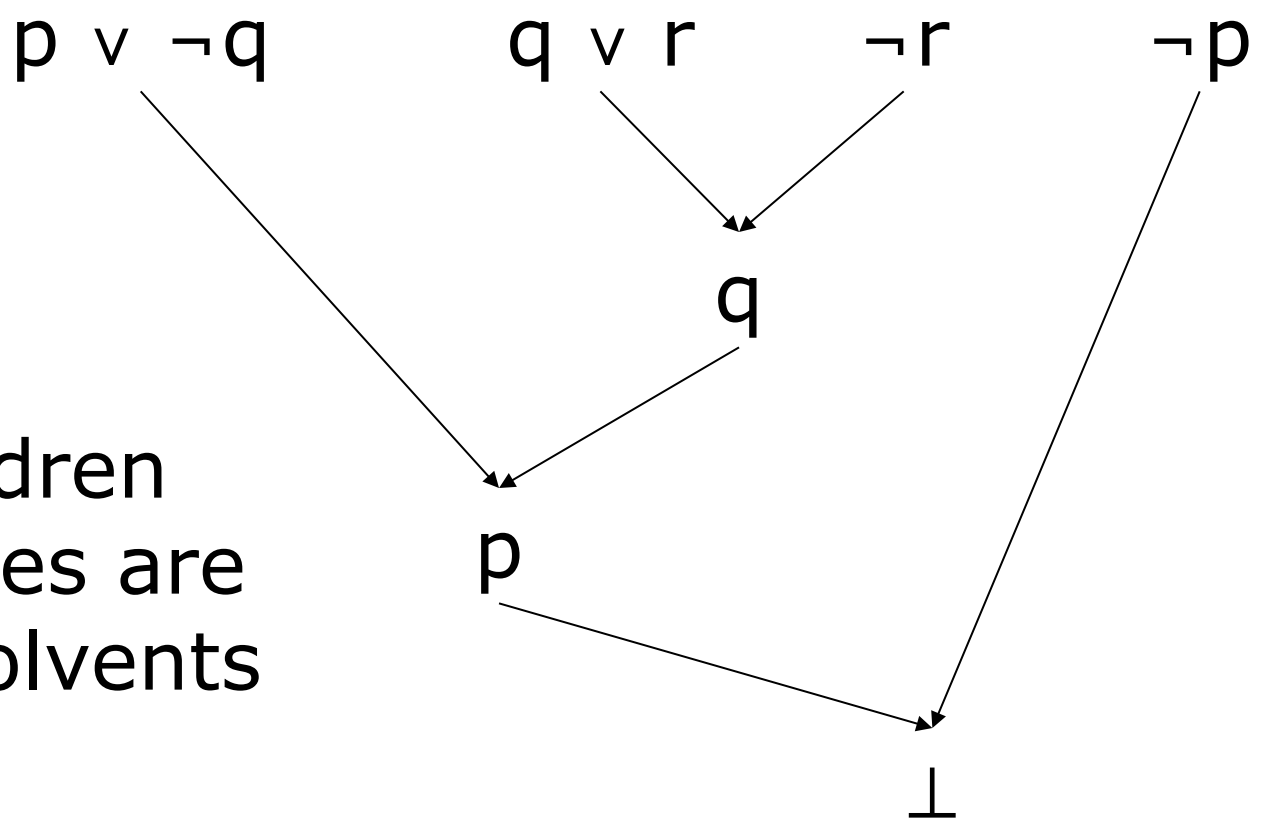
Resolution Algorithm Termination

- 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 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.	$\bot$	Resolution 6, 4

Resolution as a Tree



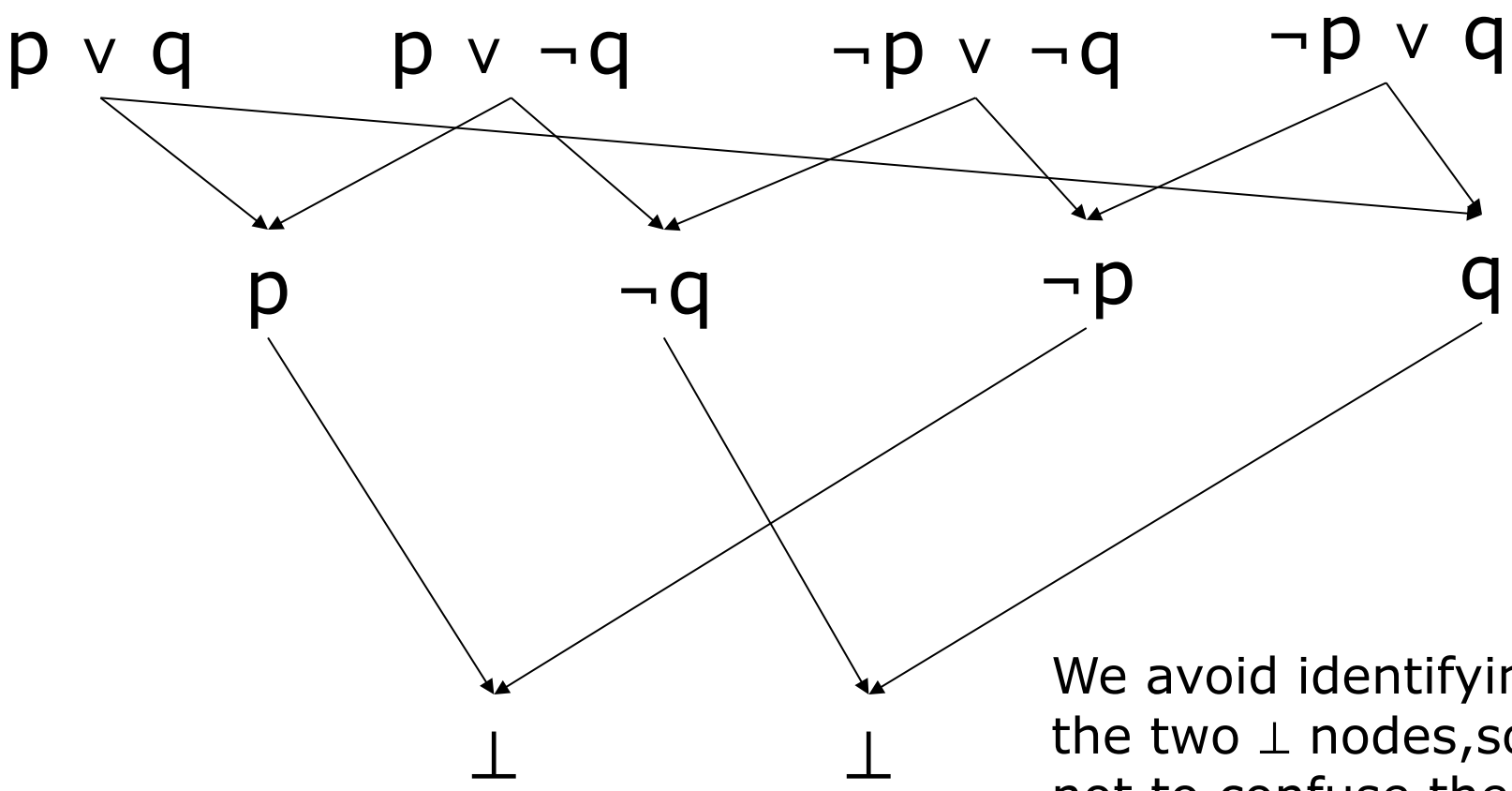
children nodes are resolvents

Try resolving these clause sets:

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

- p ∨ ¬q ∨ r,
  q ∨ r,
  ¬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.

Useful Editing Short-cuts

- Uncomplemented Literal Lemma
  (also called the "Purity Rule")

  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 without loss of generality, thus clauses containing it cannot affect satisfiability.

Example of Uncomplemented Literal Lemma

- ¬p ∨ q ∨ r,
  ¬q ∨ r,
  q ∨ s,
  ¬s,
  p
- r occurs only uncomplemented.
- The clause set is unsatisfiable iff the following set is:
- q ∨ s,
  ¬s,
  p
- and this set is unsatisfiable iff ¬s is unsatisfiable (which it isn't).

Further Editing Short-Cuts

- **Unit Clause Lemma**:

  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 thus contribute nothing to the individual clauses.

Example of Unit Clause Lemma

- ¬p ∨ q ∨ r
  q ∨ s
  ¬s
  p ∨ ¬s

- ¬s is a unit clause. The complement of ¬s is s.

- The clause set is unsatisfiable iff the following set is:
  ¬p ∨ q ∨ r,
  q, (formerly q ∨ s)

  (¬s and p ∨ ¬s were removed.)

DPLL

- The previous two edit rules are the basis of another algorithm for satisfiability: DPLL for Davis-Putnam-Logemann-Loveland
- http://en.wikipedia.org/wiki/DPLL_algorithm

Further Useful Optimizations

Subsumption Lemma:

- 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 (the larger one) 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

- $\neg p \vee q \vee \neg r$,
  $\neg p \vee \neg r$,
  $p \vee r \vee q$
- The second clause subsumes the first.

- The clause set is unsatisfiable iff the following set is:

  $\neg p \vee \neg r$,
  $p \vee r \vee q$

Common Special Case of Clause Set

- Often we want to prove a **sequent** such as:
  - $\varphi_{11} \wedge \varphi_{12} \wedge \ldots \wedge \varphi_{1m1} \rightarrow \psi_1$,
    $\varphi_{21} \wedge \varphi_{22} \wedge \ldots \wedge \varphi_{2m2} \rightarrow \psi_2$,
    . . .
    $\varphi_{n1} \wedge \varphi_{n2} \wedge \ldots \wedge \varphi_{nmn} \rightarrow \psi_n$
    “axioms”

    $\vdash \chi_1 \wedge \chi_2 \wedge \ldots \wedge \chi_p$ “theorem”

    where each symbol represents a literal.

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

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

Strategic Optimizations

- **Unit-Preference**: Prefer resolving 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.
- The latter 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"

$\neg\chi_1 \vee \neg\chi_2 \vee \ldots \vee \neg\chi_p\}$

Initial 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

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

  interpreted as

  $$\varphi_1 \wedge \varphi_2 \wedge \ldots \wedge \varphi_m \rightarrow \psi$$

  If m = 0, then we just write

  $$\psi.$$

Prolog uses a special form of resolution to do its work ("SLD" = Selective Linear Definite resolution)

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

  Non-negated literals in red.

  becomes in Prolog syntax:

- 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
```

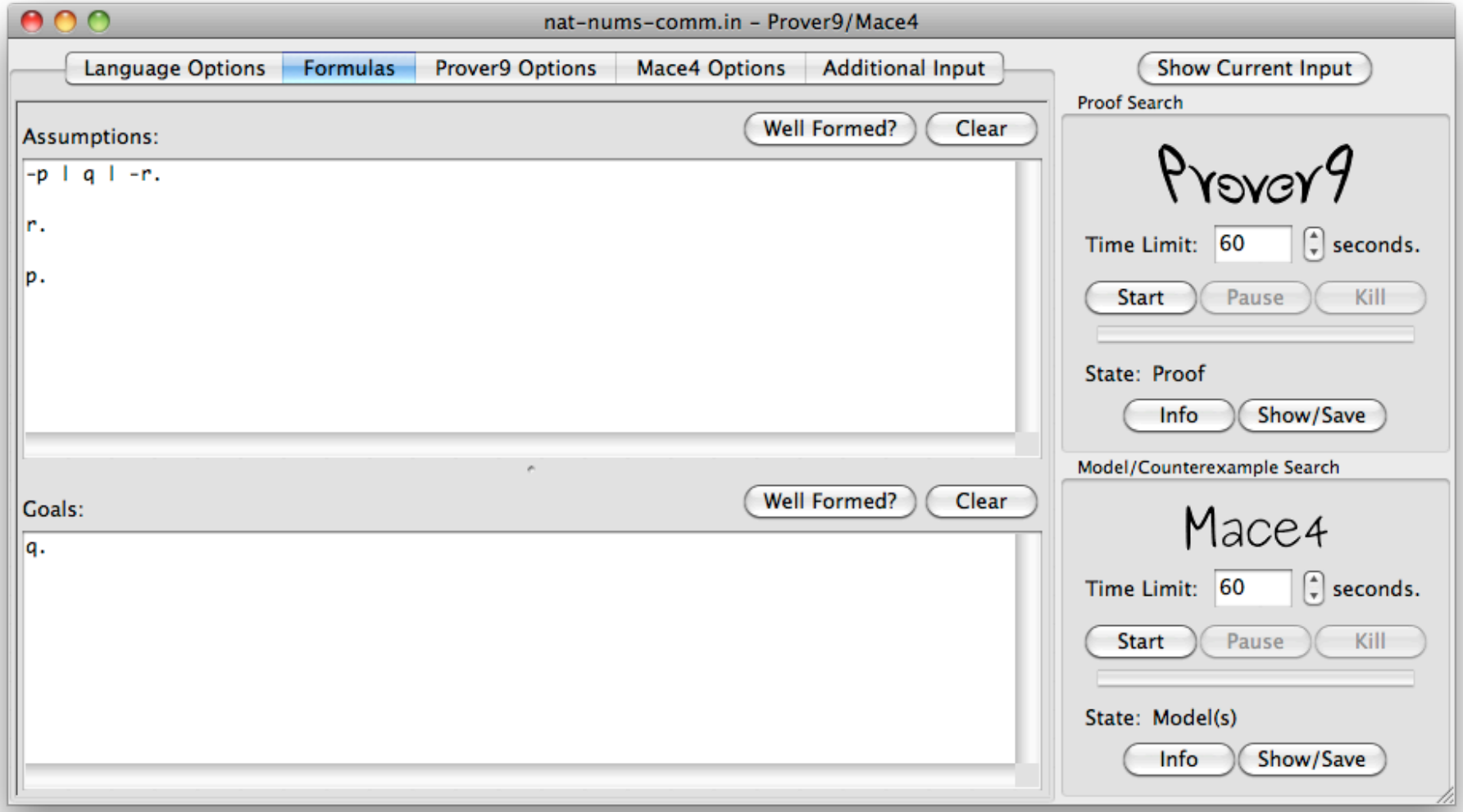
Resolution Theorem Provers

- Prolog cannot handle general negation
- Resolution theorem provers can
- Examples: Prover9, Vampire, …

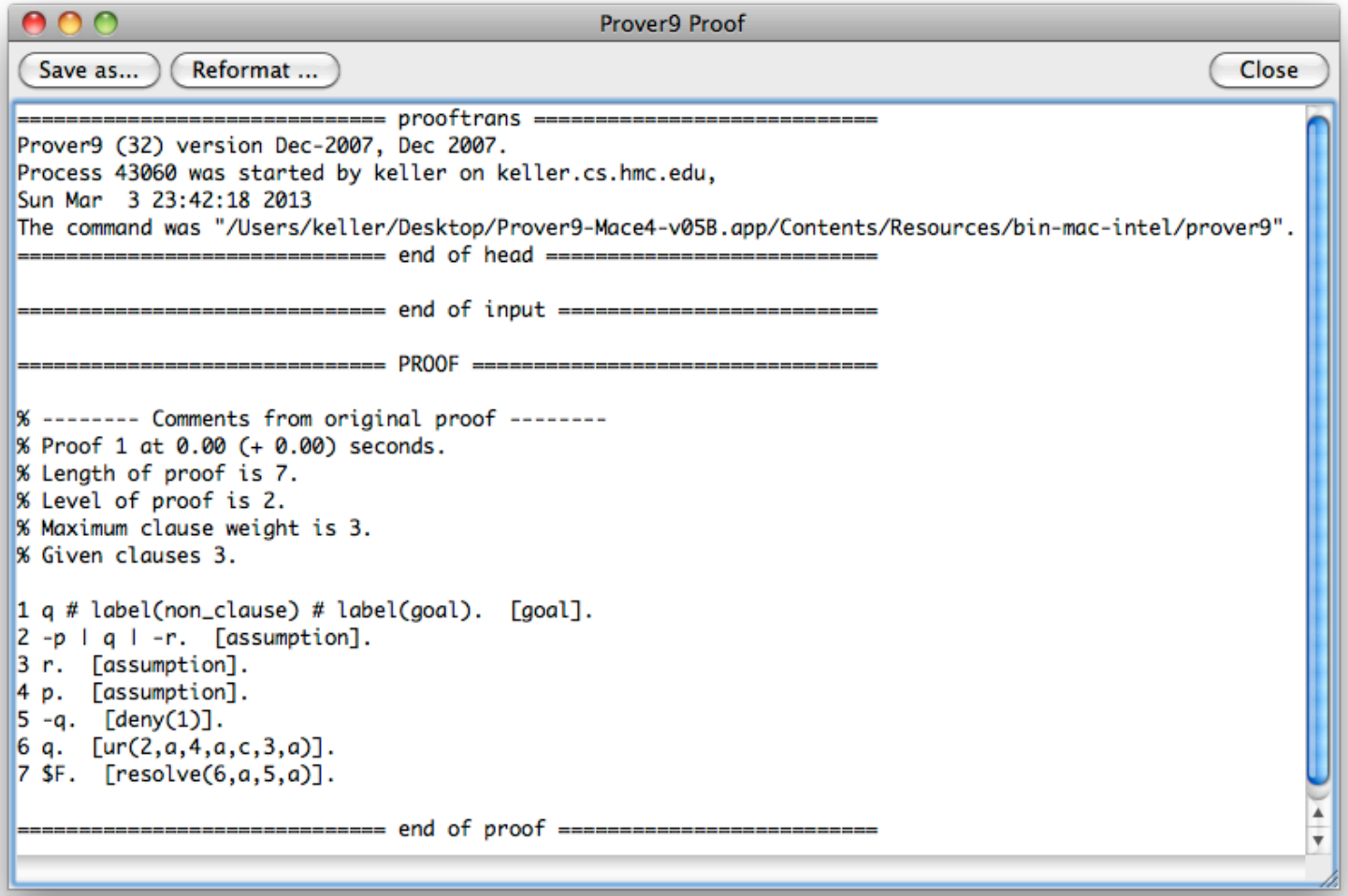
Prover9

- Extends the former program “Otter”
- Developed at Argonne National Laboratory
- Free download for all platforms
- http://www.cs.unm.edu/~mccune/prover9/
- Also includes Mace for constructing counterexamples

Prover9 GUI (- is “not”, | is “or”)



Prover9 Proof ($F is empty clause)



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 each 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 $\{p(X), q(X, Y), \neg q(X, X) \vee p(X)\}$ 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 completely general, as far as showing unsatisfiability is concerned.

Examples of Predicate Clausal Form

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

- These clauses can be used to prove the syllogism:
  - All humans are mortal.
  - Socrates is a human.
  - 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. X in

  $\neg q(X, X) \vee p(X)$
  $\neg p(Z) \vee r(Z, Y)$
- For one thing, the identity of the **variables** is **independent** in each.
- For another, the arguments are generally **terms**, not just simple variables:

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

Example of 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 “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 **set of substitutions** of terms for their variables that makes them **identical**.
- If such a substitution set 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, X a variable).

- These terms are **unifiable**, since the substitution [c/X] (substitute c for X) makes them identical.

Unification Examples

- Consider q(c, d), q(X, X)
  (c and d are constants, X a variable).
- These terms are **not unifiable**.
- Distinct **constant symbols do not unify**. There is no substitution that will make them identical.
- (Note: This is not the same as saying constant symbols cannot be equated. They can, with a separate equation such as c = d. Equality is handled separately.)

Renaming Apart

- 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**.

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

Unifiers

- A set of substitutions that unifies two literals is called a **unifier**.

More Unification Examples

Term 1	Term 2	Unifier, if any?
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	Unifier, if any?
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.

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))	

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 := $\{[\tau_1, \tau_2]\}$; // functions as a sort-of stack
  μ := the empty substitution;
  while(S $\neq \emptyset$)
  remove a pair [L, R] from S; // pop case
  if(L = R) (1)
  do nothing;
  else if(L=**f**(s_1, s_2, …, s_n) and R=**f**(t_1, t_2, …, t_n)) // same f, n (2)
  S := S $\cup$ {[s_1, t_1], [s_2, t_2],… [s_n, t_n]}; // 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;

see also http://en.wikipedia.org/wiki/Unification_(computer_science)

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.
  

- p(f(Y), g(Z), f(Y)) vs.
  

- p(f(g(Z)), g(Z), f(g(Z))) vs.
  

- p(f(g(Z)), g(Z), f(g(Z))) vs.
  

Note on Unification in Prolog

- In Prolog, unification is used in goal matching and in the = (unify) 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(…)))

Checking Unifiability with Prolog

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

```
$ 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

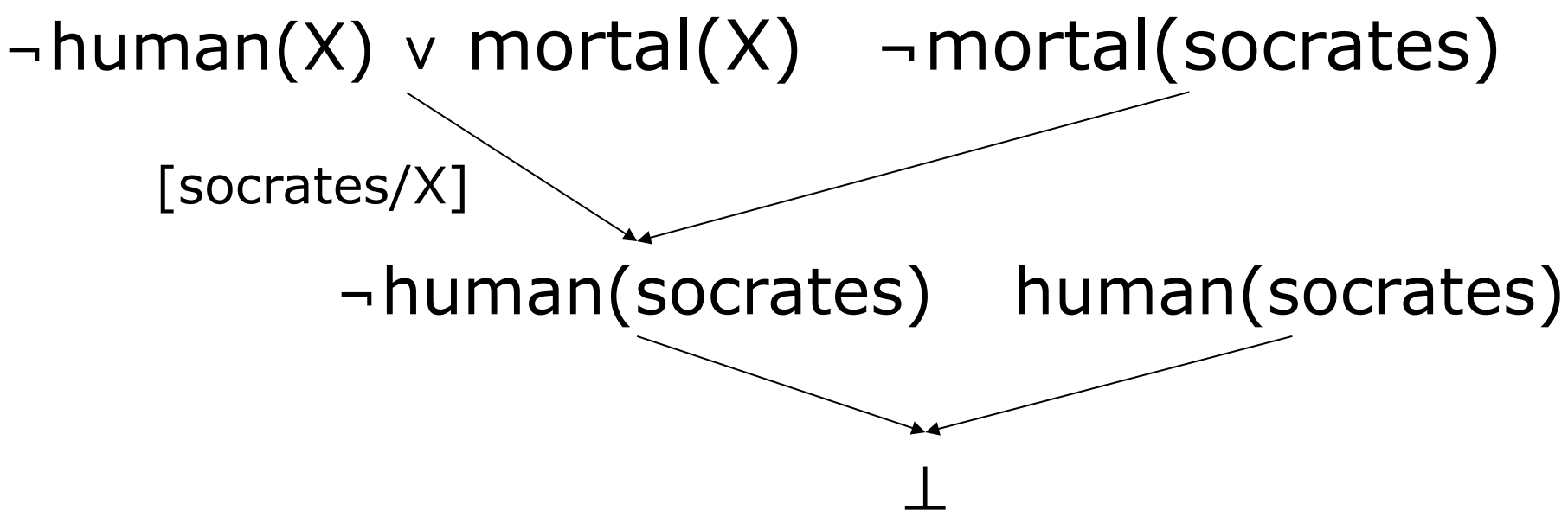
- Resolvable clauses must contain literals with the **same predicate** symbol but of **opposite sign** (one negated, the other not).
- Pick two such literals, one from each clause.
- Rename the clauses apart.
- Determine whether the literals are unifiable, with mgu μ. If they are, apply μ to ***all*** 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 similar 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.
- There are also a special issue: “factoring”, that needs to be considered.

Example of Predicate Resolution

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



Example Resolving Predicate Clauses

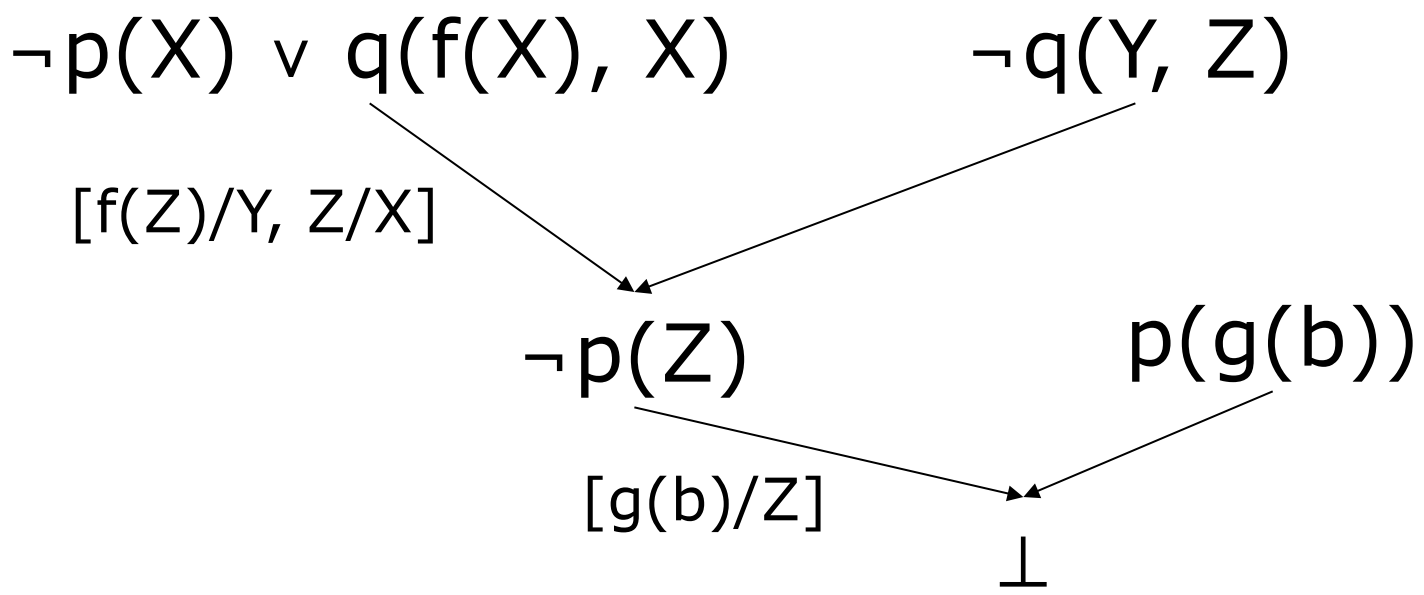
- clause 1: p(X, g(Z), X) ∨ q(X, h(Z))
- clause 2: ¬p(f(Y), Y, W) ∨ r(f(Y), g(W))
- These are already renamed apart.

- 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)



Unit Preference

- As with propositional resolution, resolving with unit clauses first is a good heuristic.

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)

Herbrand's Theorem: Why Resolution Works

Jacques Herbrand *(au centre)*
au cours de l'excursion où il trouva la mort

- Jacques Herbrand showed the following:
  - A set of clauses is unsatisfiable iff there is a refutation using a certain kind of symbolic interpretation (known as a **Herbrand Interpretation**):
    - For each constant symbol, the interpretation is literally that symbol. (If no constant symbols, add one.)
    - Functions are defined as follows:

$$I(f(t_1, \ldots, t_n)) = \text{the } \textit{string } f(I(t_1), \ldots, I(t_n))$$

http://mathworld.wolfram.com/HerbrandUniverse.html
http://en.wikipedia.org/wiki/Herbrand_interpretation

Ground Terms and Clauses

- A ground term is a term in which there are no variables.
- A ground clause is a clause in which there are no variables.
- If there is a refutation at all, there is one using only ground clauses from a Herbrand interpretation.

Remember to treat clauses as sets.

- 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 is not enough.
- We might need to “factor” two or more literals in the same clause to make progress.

Factoring

- Two or more literals of the same sign in one clause can be unified (without renaming apart) 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:

  $$P(x) \vee P(f(y)) \vee \neg Q(x)$$

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

- The resulting factor is:

  $$P(f(y)) \vee \neg Q(f(y))$$

- $(\forall x\ \forall y\ (P(x) \vee P(f(y)) \vee \neg Q(x))) \rightarrow \forall y\ (P(f(y)) \vee \neg Q(f(y)))$ is valid

Use of Factoring

- Suppose our clause set includes:
  $P(x) \vee P(f(y)) \vee \neg Q(x)$
  $\neg P(f(a))$

- With binary resolution, we'd get the resolvent:
  $P(x) \vee \neg Q(x)$.

- If we **first factor**, to get $P(f(y)) \vee \neg Q(f(y))$ as on the previous slide, we can get a resolvent $\neg Q(f(a))$, 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

- $P(x) \vee P(y)$
- $\neg P(a) \vee \neg P(b)$
- Without factoring, generate:
- $P(y) \vee \neg P(b)$
- $P(x) \vee \neg P(a)$
- and more similar clauses,
  but never the empty clause.

Case Where Factoring is Necessary

- $P(x) \vee P(y)$
- $\neg P(a) \vee \neg P(b)$
- With factoring, get factor p(x) from first clause, then generate:
- $\neg P(b)$
- $\perp$

Subsumption

- A clause C **subsumes** a clause D if there is a substitution θ such that $C\theta \subseteq D$, where we interpret the clauses as **sets** of their literals.

- If a clause D in a set of clauses is subsumed by another clause C ***within the set***, then we can delete D from the set without affecting the case of whether the empty clause $\perp$ is derivable.

Subsumption Examples

- $P(X)$ subsumes $P(X) \vee Q(Y)$
  (by the empty substitution $\{\}$).

- $\neg P(X) \vee Q(f(X))$ subsumes
  $\neg P(Z) \vee \neg P(h(Y)) \vee Q(f(h(Y)))$
  (by the substitution $\{X \leftarrow h(Y), Z \leftarrow h(Y)\}$).

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. ∀X fungus(x) → (mushroom(X) ∨ toadstool(X))
2. ∀X boletus(X) → fungus(X)
3. ∀X toadstool(X) → poisonous(X)
4. ∀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)

Mushroom Clauses in Prover9

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

-boletus(x) | fungus(x).

-toadstool(x) | poisonous(x).

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

 boletus(b).


Goal:

poisonous(b).
```

Prover9 Output for Mushrooms

```
============================== PROOF =================================

% -------- Comments from original proof --------
% Proof 1 at 0.00 (+ 0.00) seconds.
% Length of proof is 13.
% Level of proof is 5.
% Maximum clause weight is 0.
% Given clauses 0.

1 poisonous(b) # label(non_clause) # label(goal).  [goal].
2 -boletus(x) | fungus(x).  [assumption].
3 -fungus(x) | mushroom(x) | toadstool(x).  [assumption].
4 -boletus(x) | mushroom(x) | toadstool(x).  [resolve(2,b,3,a)].
5 -toadstool(x) | poisonous(x).  [assumption].
6 boletus(b).  [assumption].
7 -boletus(x) | -mushroom(x).  [assumption].
8 -boletus(x) | mushroom(x) | poisonous(x).  [resolve(4,c,5,a)].
9 mushroom(b) | poisonous(b).  [resolve(8,a,6,a)].
10 -poisonous(b).  [deny(1)].
11 mushroom(b).  [resolve(9,b,10,a)].
12 -mushroom(b).  [resolve(6,a,7,a)].
13 $F.  [resolve(11,a,12,a)].

============================== end of proof ==========================
```

Checking Unifiability with Prover9

- In contrast to Prolog,
  Prover9 *does* use an occur-check.

unification succeeds

```
-p(f(y), y).

p(x, g(z)).
```

Proof:

```
1 p(x,g(y)).  [assumption].
2 -p(f(x),x).  [assumption].
3 $F.  [resolve(1,a,2,a)].
```

unification fails due to occur-check

```
-p(f(y), y).

p(z, g(z)).
```

Prover9 Exit: Exhausted

Clausal Form for Predicate Logic

- Often, we'll want to prove a sequent of the form
  - ∀x ∀y (…),
  - ∀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, so that will become
    ¬∀x ∀y (…) which is equivalent to

    ∃x ∃y ¬(…).

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 negated 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.

Yet Another Example

- Consider the sequent

  $\forall x\ a(x) \rightarrow \exists x\ b(x) \vdash \exists x\ (a(x) \rightarrow b(x))$

- Premise clause:

  $\neg a(c) \vee b(d)$

- Conclusion clauses:

  $a(x)$
  $\neg b(x)$

Resolution produces $\perp$ in 2 steps.

Skolem Functions for the General Case

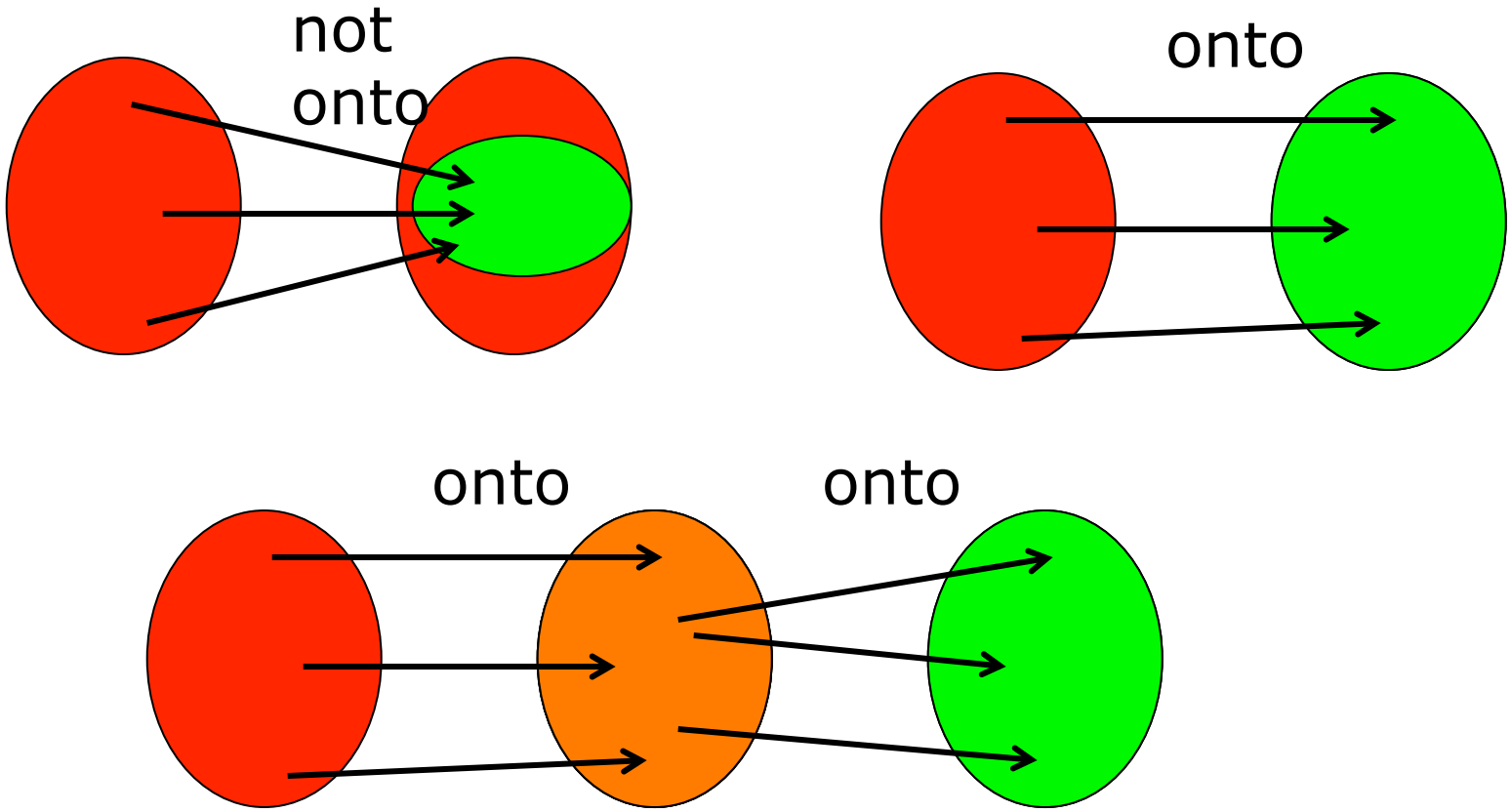
- ∀x ∀y … ∃v …
- v is replaced with f(x, y, …)
- f is a **new function symbol**, the arguments of which are the ∀ quantified variables on the left.

- The rationale here is that “the v” that exists **depends on** x, y, … .

- Again, there is an interpretation satisfying the original formula iff there is an interpretation satisfying the revised formula.

Example: Skolem with Arguments

Prove: “The composition of two onto [surjective] functions is onto.”



Example: Skolem with Arguments

- 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 $F(f(y), y)$ [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 called the "prefix" and the "matrix".]

- Skolemize ∃ quantified variables.

- Drop ∀ quantifiers.

- Convert the resulting matrix to CNF.

Conversion to Prenex Form

- Replace all connectives other than ∧∨¬ with their ∧∨¬ counterparts.
- Push negations inward
- Pull quantifiers to the outside using the rules on the next page.

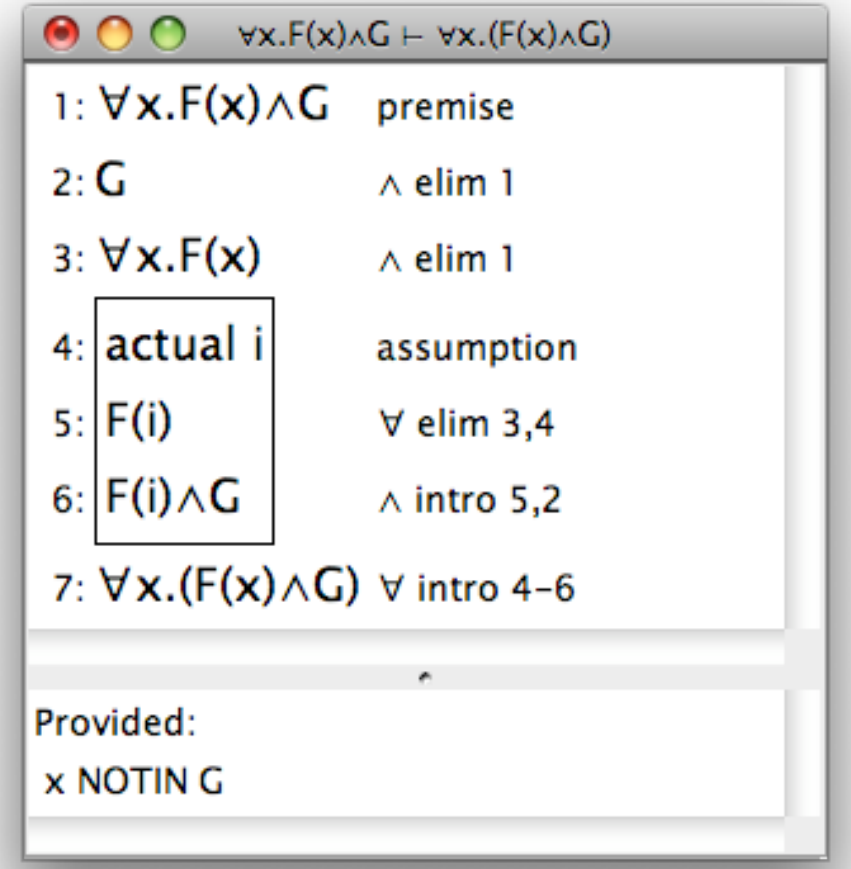
Basic Prenex Quantifier Rules

(for pulling quantifiers to the outside)

- We can show that the following replacements are equivalent.
- Here ⇒ 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:

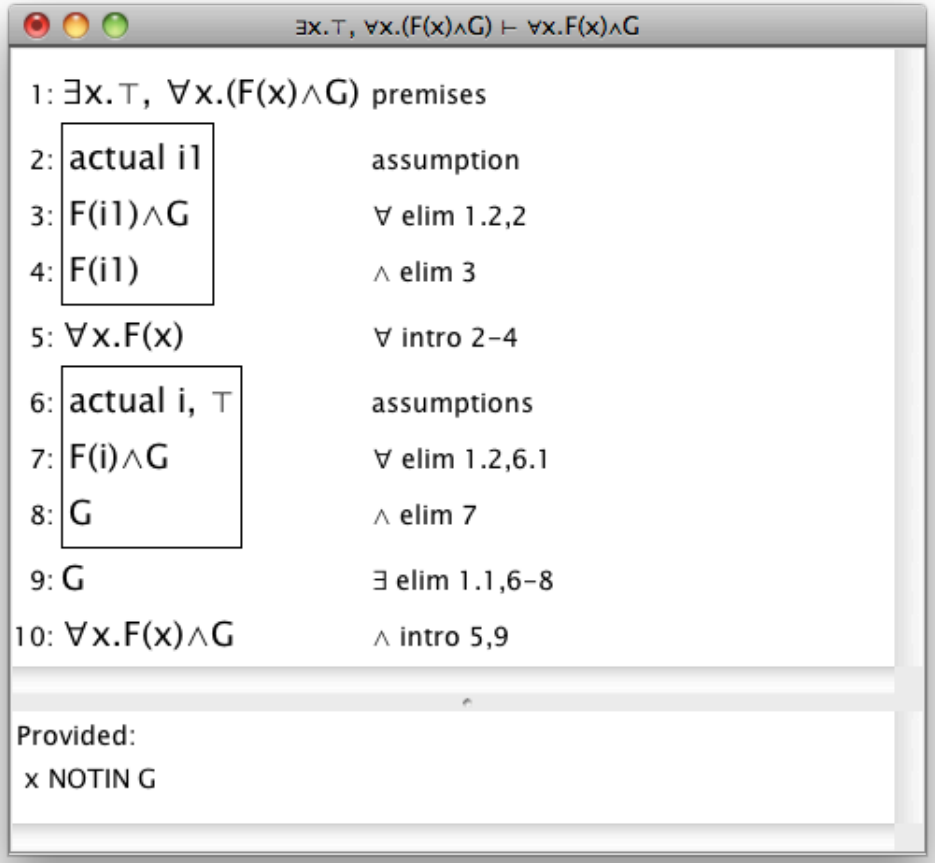
$(\exists x\ F[x]) \wedge (\forall x\ G[x]) \Rightarrow$ (by renaming second x)
$(\exists x\ F[x]) \wedge (\forall y\ G[y]) \Rightarrow$ (by rule 3, as x is not free)
$(\exists x\ (F[x] \wedge (\forall y\ G[y]))) \Rightarrow$ (by rule 1 symmetric counterpart)
$\exists x\ \forall y\ (F[x] \wedge G[y])$

Justifying Quantifier Motion Rules Using Natural Deduction: ∀∧ Rule a.



Proviso is introduced by prefixing 'WHERE x NOTIN G IS' in Jape.

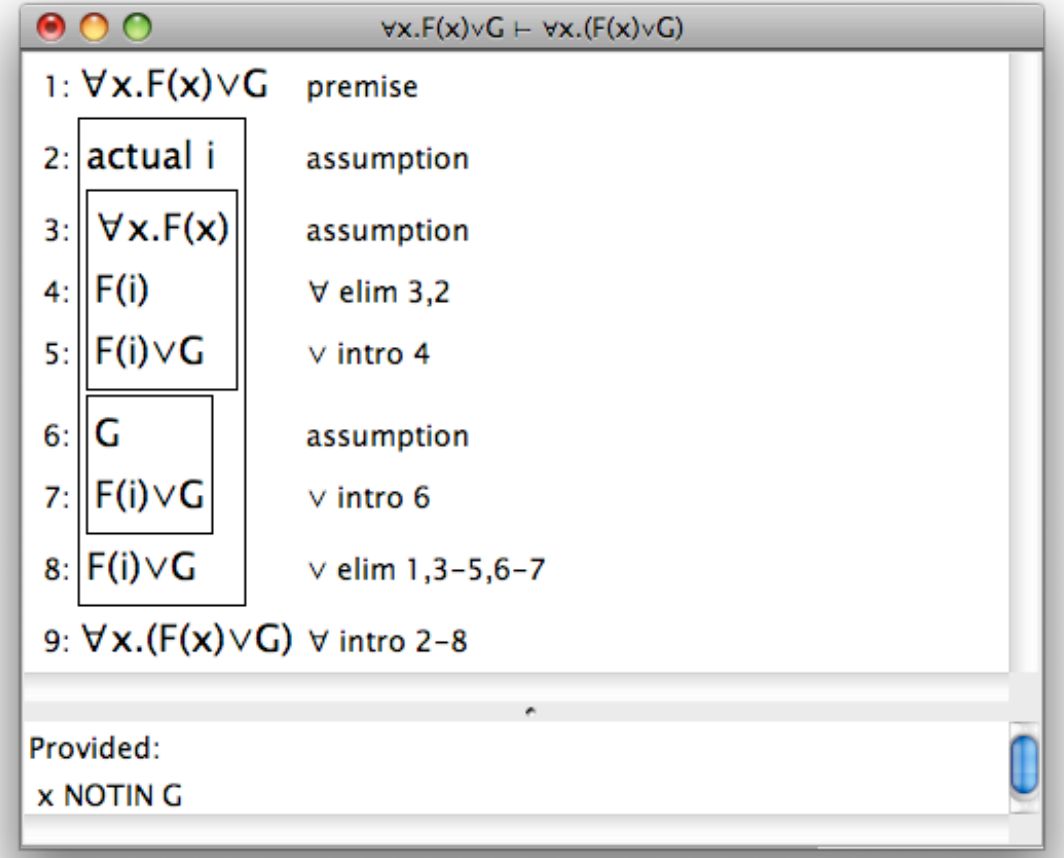
For equivalence, we need the converse of each rule: ∀∧ Rule b.



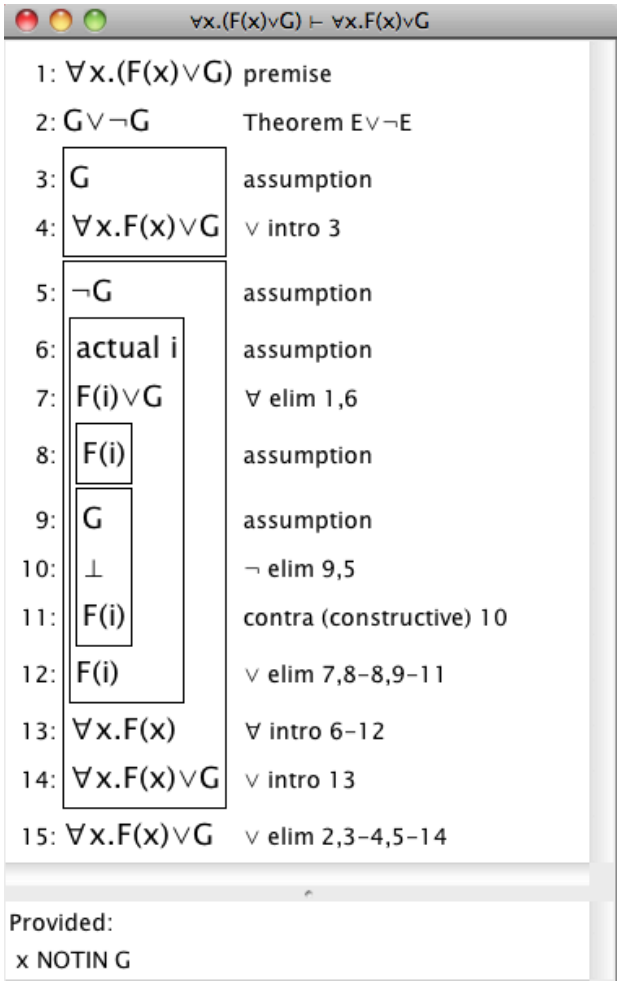
Need the non-empty universe assumption in this direction.

Otherwise, there is no way to get G by itself.

Justification of Rules
Using Natural Deduction: ∀∨ Rule a.

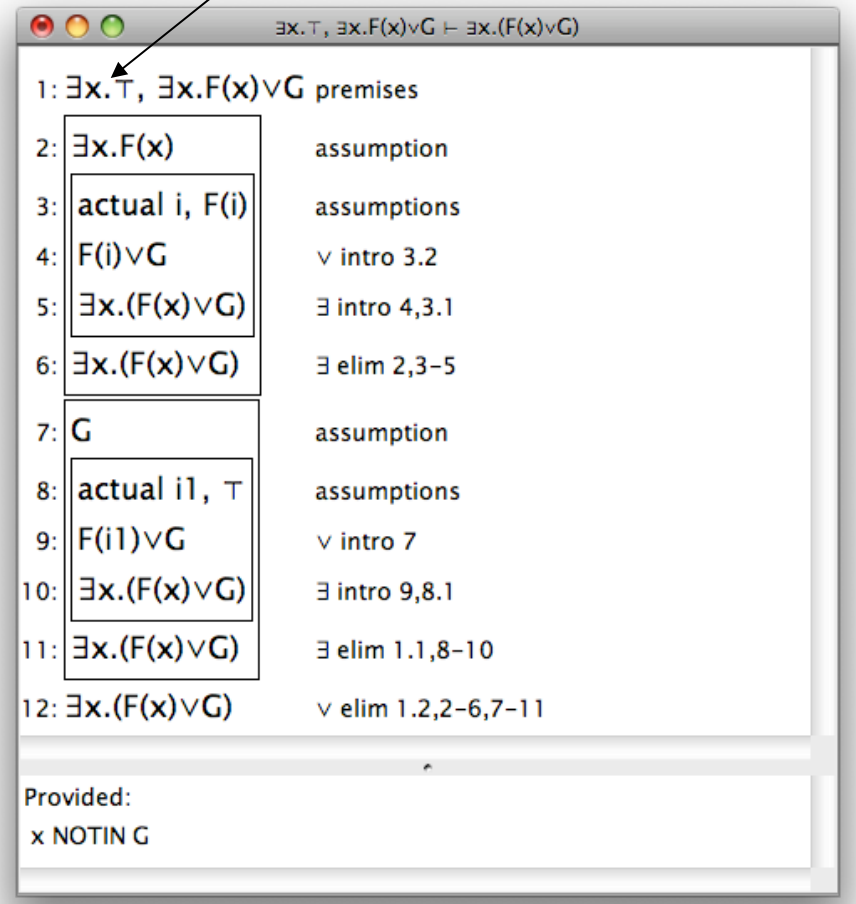


Justification of Rules
Using Natural Deduction: ∀v Rule b.

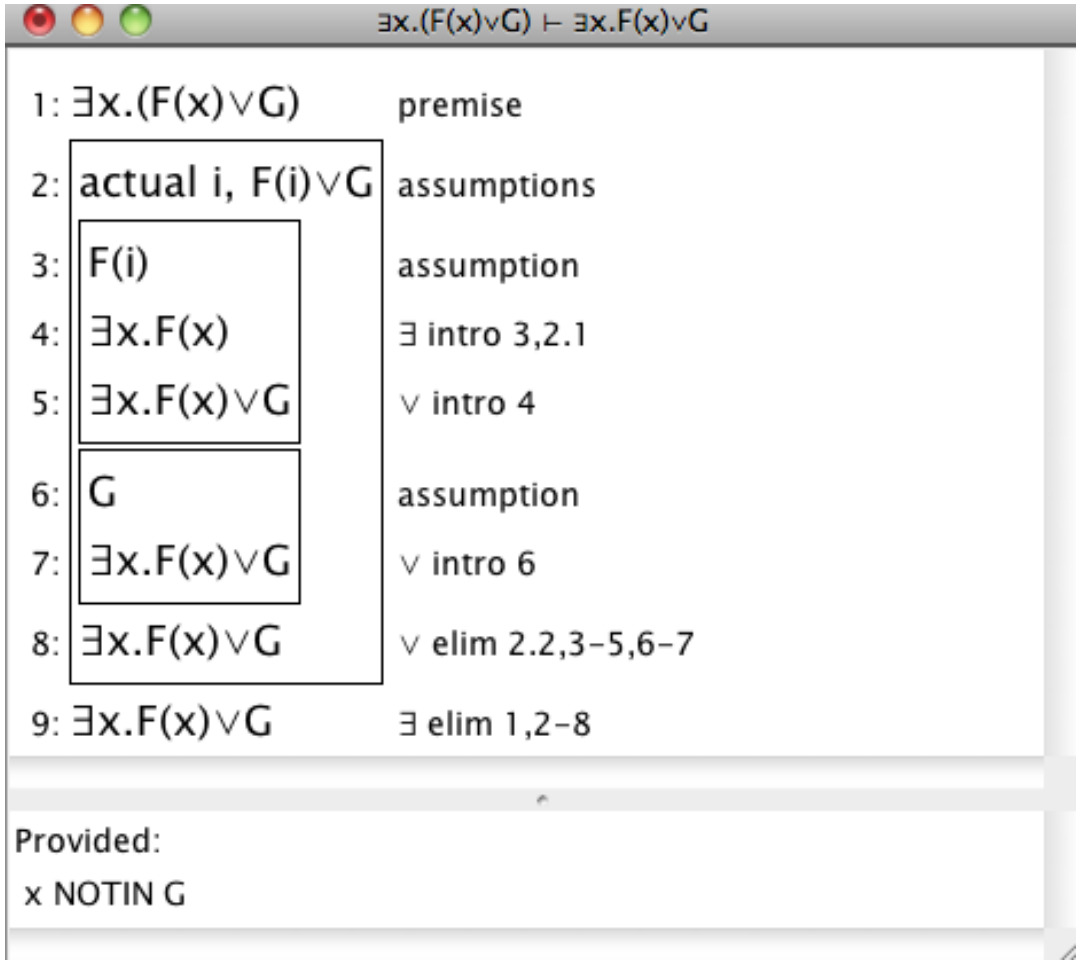


Justification of Rules Using Natural Deduction: ∃v a.

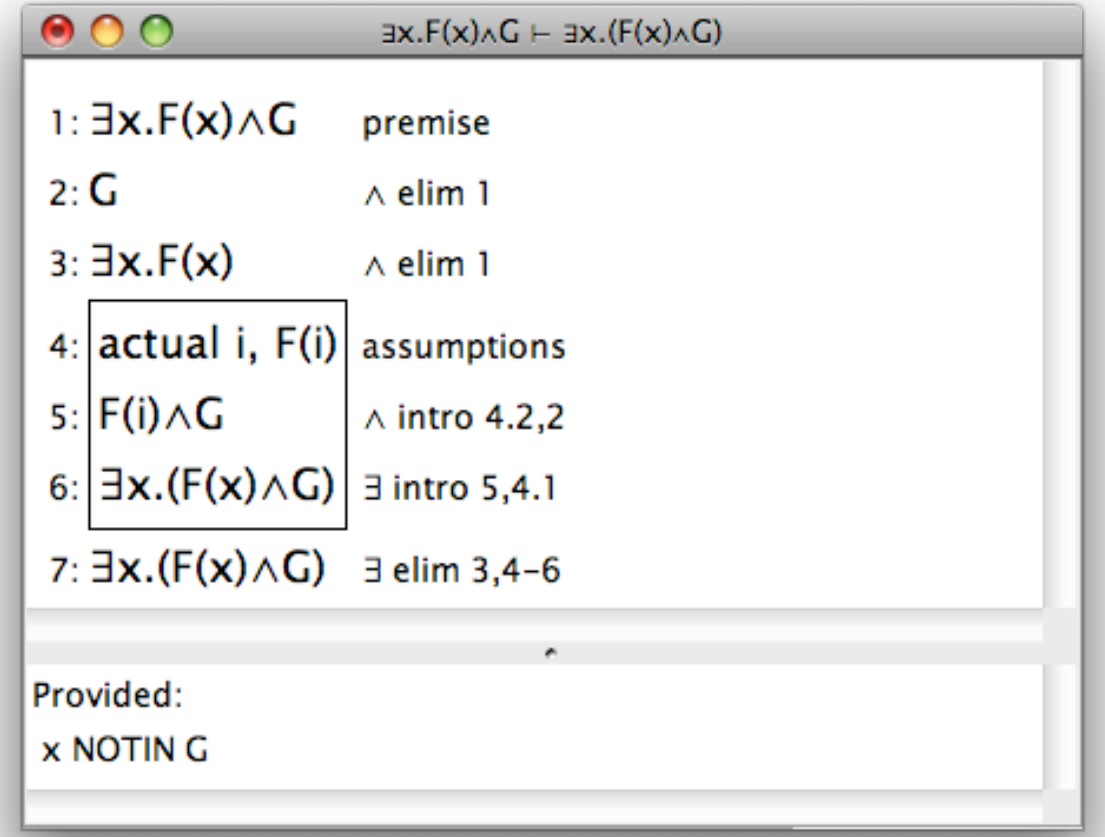
Non-empty universe assumption, needed in 7-11



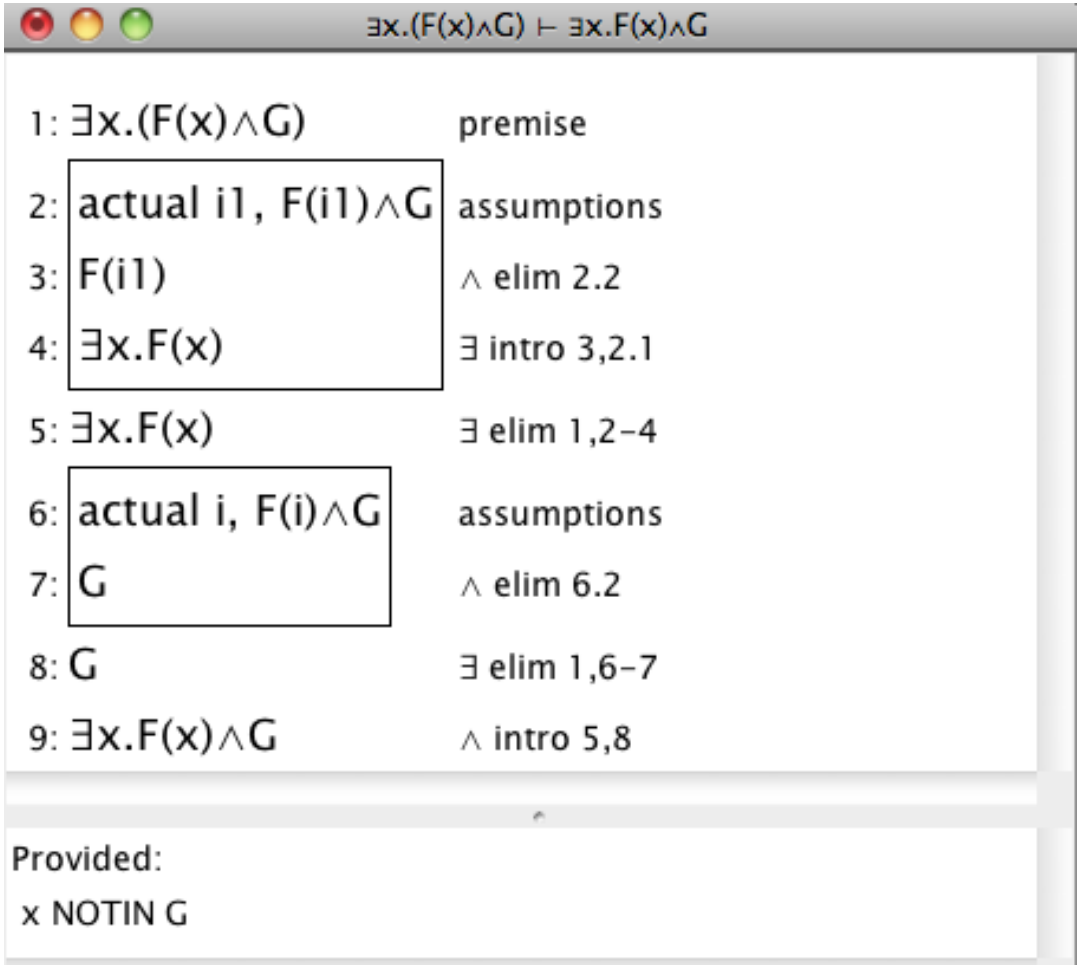
Justification of Rules Using Natural Deduction: ∃v b.



Justification of Rules Using Natural Deduction: ∃∧ a.



Justification of Rules Using Natural Deduction: ∃∧ b.



Example of Prenex Conversion

- $\forall x\ \forall y((\exists z(p(x, z) \wedge p(y, z)) \rightarrow \exists u\ q(x, y, u))$ replace →
- $\forall x\ \forall y(\neg(\exists z(p(x, z) \wedge p(y, z)) \vee \exists u\ q(x, y, u))$ push ¬ in
- $\forall x\ \forall y((\forall z\ \neg(p(x, z) \wedge p(y, z)) \vee \exists u\ q(x, y, u))$ push ¬ in
- $\forall x\ \forall y\ (\forall z(\neg p(x, z) \vee \neg p(y, z)) \vee \exists u\ q(x, y, u))$ pull ∃u out
- $\forall x\ \forall y\ \exists u\ (\forall z\ (\neg p(x, z) \vee \neg p(y, z)) \vee q(x, y, u))$ pull ∀z out
- $\underbrace{\forall x\ \forall y\ \exists u\ \forall z}_{\text{prefix}}\ \underbrace{(\neg p(x, z) \vee \neg p(y, z) \vee q(x, y, u))}_{\text{matrix}}$

Completion of Conversion to CNF

- Prenex Form:

  $\forall x\ \forall y\ \exists u\ \forall z\ (\neg p(x, z) \vee \neg p(y, z) \vee q(x, y, u))$

- Skolemize u as f(x, y) and drop $\forall x\ \forall y\ \forall z$

  $\neg p(x, z) \vee \neg p(y, z) \vee q(x, y, f(x, y))$

Example: Group Theory Clauses

- f is the group operation, i is the inverse 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.
- $\forall x\ e(x, x)$
  becomes
  $e(x, x)$
- $\forall x\ \forall y\ \forall u\ \forall v\ ((e(x, y) \wedge e(v, w)) \rightarrow e(f(x, v), f(y, w)))$
  becomes
  $\neg e(x, y) \vee \neg e(v, w) \vee e(f(x, v), f(y, w))$
- $\forall x\ \forall u\ (e(x, u) \rightarrow e(i(x), i(u))$
  becomes
  $\neg e(x, u) \vee e(i(x), i(u))$
- $\forall x\ \forall y\ (e(x, y) \rightarrow e(y, x))$
  becomes
  $\neg e(x, y) \vee e(y, x)$

  Exercise: Convert the transitive property of e.

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. $\neg e(x, y) \vee \neg e(v, w) \vee e(f(x, v), f(y, w))$
6. $\neg e(x, y) \vee e(y, x)$
7. $\neg e(x, y) \vee \neg e(y, z) \vee e(x, z)$

8. $\neg e(i(i(b)), b)$

   This is to show that $\forall x\ e(i(i(x)), x)$:

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

Prover9 input using builtin equality

```
f(x, f(y, z)) = f(f(x, y), z).
f(x, c) = x.
f(x, i(x)) = c.
i(i(b)) != b.

(Show unsatisfiable)
```

Prover9 proof using builtin equality

```
1 f(x,f(y,z)) = f(f(x,y),z).  [assumption].
2 f(f(x,y),z) = f(x,f(y,z)).  [copy(1),flip(a)].
3 f(x,c) = x.                     [assumption].
4 f(x,i(x)) = c.                  [assumption].
5 i(i(b)) != b.                   [assumption].
6 f(x,f(c,y)) = f(x,y).           [para(3(a,1),2(a,1,1)),flip(a)].
7 f(x,f(i(x),y)) = f(c,y).        [para(4(a,1),2(a,1,1)),flip(a)].
12 f(c,i(i(x))) = x.      [para(4(a,1),7(a,1,2)),rewrite([3(2)]),flip(a)].
14 f(x,i(i(y))) = f(x,y). [para(12(a,1),2(a,2,2)),rewrite([3(2)])].
15 f(c,x) = x.            [para(12(a,1),6(a,2)),rewrite([14(5),6(4)])].
18 f(x,f(i(x),y)) = y.    [back_rewrite(7),rewrite([15(5)])].
21 i(i(x)) = x.           [para(4(a,1),18(a,1,2)),rewrite([3(2)]),flip(a)].
22 $F.                    [resolve(21,a,5,a)].
```

Equality: Paramodulation

- Prover9 has a built-in equality, so axiomatizing equality as a predicate is not generally necessary.

- The "paramodulation" rule, which essentially captures the = rules of Natural Deduction.

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

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

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

Example: Non-Clausal Input in Prover9: Automatic Translation to Clausal Form

```
all x exists y r(x, y).

all x all y all z ((r(x,y)&r(y,z))->r(x,z)).

all x all y (r(x,y) -> r(y,x)).

-(all x r(x,x)).
```

Example: Non-Clausal Input in Prover9: Automatic Translation to Clausal Form

becomes (c1, f1 are Skolem constant and function):

```
1 (all x exists y r(x,y)) # label(non_clause).                          [assumption].
2 (all x all y all z (r(x,y) & r(y,z) -> r(x,z))) # label(non_clause).   [assumption].
3 (all x all y (r(x,y) -> r(y,x))) # label(non_clause).                 [assumption].
4 -(all x r(x,x)) # label(non_clause).                                  [assumption].

5 r(x,f1(x)).                                                           [clausify(1)].
6 -r(x,y) | -r(y,z) | r(x,z).                                           [clausify(2)].
7 -r(x,y) | r(y,x).                                                     [clausify(3)].
8 -r(c1,c1).                                                            [clausify(4)].

10 r(f1(x),x).                                                          [ur(7,a,5,a)].
12 -r(f1(c1),c1).                                                       [ur(6,a,5,a,c,8,a)].
13 $F.                                                                  [resolve(12,a,10,a)].
```

Answer Extraction

- Resolution is not just for proving theorems anymore.
- Resolution can be used for extracting answers from a database, knowledge base, or reasoning system.

From Yes-No Answer to Terms

- Consider the clause set:
  - ¬human(x) ∨ mortal(x)
  - human(socrates)
  - ¬mortal(socrates)
- Obviously this set is unsatisfiable, and a proof can be obtained by resolution.
- What if we drop the third clause. The first two clauses are satisfiable, and can be considered a "knowledge base".
- We can ask a question of this knowledge base, such as:

Name someone who is mortal.

Answer Literals

- An answer literal is a special literal that captures the answer to a question.

- We convert the negation of a specific conclusion into a clause involving an answer literal:

$$\neg \text{mortal}(x) \vee \text{answer}(x).$$

- Resolution stops when a clause with only the answer is present.

Resolution with Answer Literals

- Clauses:
  1. ¬human(x) ∨ mortal(x)
  2. human(socrates)
  3. ¬mortal(x) ∨ answer(x)
- Resolution steps:
  4. mortal(socrates) from 1, 2
  5. answer(socrates) from 3, 4

Answering Questions in Prover9

- To find terms such that p(x), incant:

```
-p(x) # answer(x).
```

Who is Mortal, in Prover9

Clausal Form

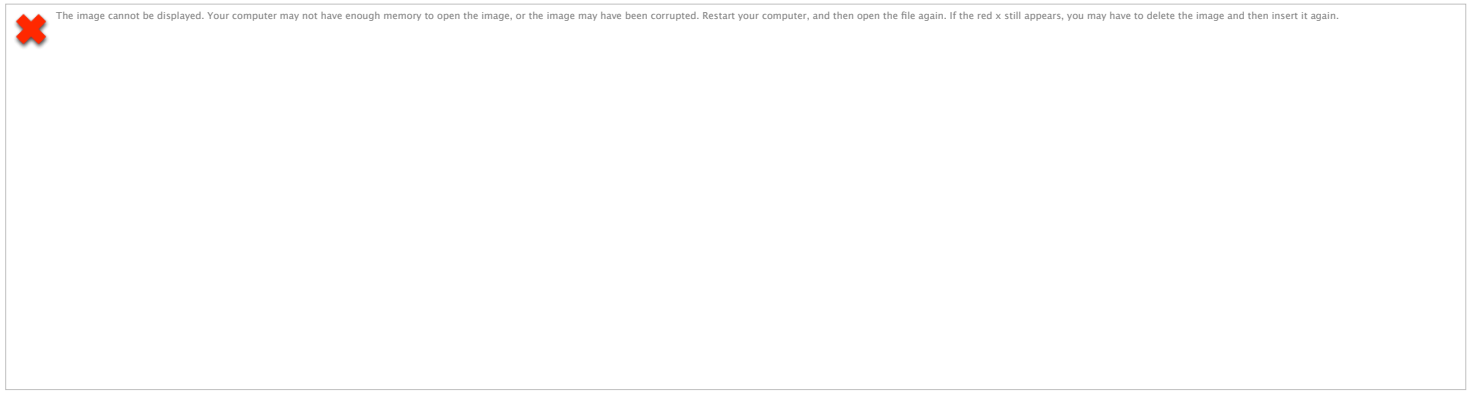
- -human(x) | mortal(x).
- human(socrates).
- -mortal(x) # answer(x).

Prover9 Solution with Answer

Clauses

- `-human(x) | mortal(x).`
- `human(socrates).`
- `-mortal(x) # answer(x).`

Proof



Who is Caroline's Grandfather?

Clauses

-father(x, y) | parent(x, y).
-father(x, y) | -parent(y, z) | grandfather(x, z).
father(joe, john).
father(john, caroline).

-grandfather(x, caroline) # answer(x).

Proof

```
1 father(joe,john).  [assumption].
2 -father(x,y) | parent(x,y).  [assumption].
3 -father(x,y) | -parent(y,z) | grandfather(x,z).  [assumption].
4 father(john,caroline).  [assumption].
5 -parent(john,x) | grandfather(joe,x).  [resolve(1,a,3,a)].
6 -grandfather(x,caroline) # answer(x).  [assumption].
8 -parent(john,caroline) # answer(joe).  [resolve(5,b,6,a)].
10 parent(john,caroline).  [resolve(4,a,2,a)].
12 $F # answer(joe).  [resolve(8,a,10,a)].
```

Logic Puzzles solvable by Resolution

% Professors Dodds, Lewis, and Stone each frequent different establishments (one of Alice's, Harry's, or Joe's) for liquid refreshment.

% Each prof prefers a different beer (one of Anchor, Bud, and Miller)
% Each establishment serves a unique beer.

% Professor Stone prefers Bud.
% Professor Lewis doesn't prefer Miller.
% The prof who prefers Miller frequents Alice's bar.
% The prof who prefers Anchor does not frequent Joe's.

% Which bar does each prof frequent and what beer does each prefer?

Encoding Information & Query

```
% Clauses corresponding to the clues:

prefer(Stone, Bud).                          % Clue 1
-prefer(Lewis, Miller).                      % Clue 2
-prefer(x, Miller) | frequent(x, Alice).     % Clue 3
-prefer(x, Anchor) | -frequent(x, Joe).      % Clue 4

% Individuals

prof(Dodds).    prof(Stone).    prof(Lewis).
beer(Anchor).   beer(Bud).      beer(Miller).
bar(Alice).     bar(Harry).     bar(Joe).

% Although constants do not unify, they could conceivably be equal.
Dodds != Stone.  Dodds != Lewis.  Stone != Lewis.
Anchor != Bud.  Anchor != Miller. Bud != Miller.
Alice != Harry.   Alice != Joe.      Harry != Joe.

% Any bar that a professor frequents serves the beers that he or she prefers.

-frequent(x, y) | serves(y, z) | prefer(x, z).
```

```
% Every bar is frequented by some prof.

-bar(y) | frequent(Dodds, y) | frequent(Stone, y) | frequent(Lewis, y).

% Every beer is preferred by some prof.

-beer(y) | prefer(Dodds, y) | prefer(Stone, y) | prefer(Lewis, y).

% Each bar serves a unique beer.

-serves(x, y) | -serves(x, z) | y = z.

% Each prof prefers a unique beer.

-prefer(x, y) | -prefer(x, z) | y = z.

% Each prof frequents a unique bar.

-frequent(x, y) | -frequent(x, z) | y = z.
```

```
% Which bars are frequented, and which beers preferred, by which professors?

 -frequent(Dodds, x)  | -frequent(Stone, y) | -frequent(Lewis, z)
|  -prefer(Dodds, u)  |  - prefer(Stone, v) |  -prefer(Lewis, w)
#answer([Dodds, x, u, Stone, y, v, Lewis, z, w]).
```

Prover9 Solution

Dodds-Alice-Miller
Stone-Joe-Bud
Lewis-Harry-Anchor

```
% Proof 1 at 0.01 (+ 0.01) seconds: [Dodds,Alice,Miller,Stone,Joe,Bud,Lewis,Harry,Anchor].
% Length of proof is 46.
% Level of proof is 11.
% Maximum clause weight is 18.
% Given clauses 85.

1 -beer(x) | prefer(Dodds,x) | prefer(Stone,x) | prefer(Lewis,x).  [assumption].
2 beer(Anchor).  [assumption].
4 beer(Miller).  [assumption].
5 -bar(x) | frequent(Dodds,x) | frequent(Stone,x) | frequent(Lewis,x).  [assumption].
7 bar(Harry).  [assumption].
8 bar(Joe).  [assumption].
9 prefer(Stone,Bud).  [assumption].
10 -prefer(Lewis,Miller).  [assumption].
11 -prefer(x,Miller) | frequent(x,Alice).  [assumption].
12 -prefer(x,Anchor) | -frequent(x,Joe).  [assumption].
54 Anchor != Bud.  [assumption].
55 Anchor != Miller.  [assumption].
61 Bud != Miller.  [assumption].
62 Miller != Bud.  [copy(61),flip(a)].
73 Alice != Harry.  [assumption].
74 Harry != Alice.  [copy(73),flip(a)].
75 Alice != Joe.  [assumption].
76 Joe != Alice.  [copy(75),flip(a)].
77 Harry != Joe.  [assumption].
78 Joe != Harry.  [copy(77),flip(a)].
81 -prefer(x,y) | -prefer(x,z) | y = z.  [assumption].
82 -frequent(x,y) | -frequent(x,z) | y = z.  [assumption].
83 -frequent(Dodds,x) | -frequent(Stone,y) | -frequent(Lewis,z) | -prefer(Dodds,u) | -prefer(Stone,w) | -prefer(Lewis,v5) # answer([Dodds,x,u,Stone,y,w,Lewis,z,v5]).  [assumption].
84 prefer(Dodds,Anchor) | prefer(Stone,Anchor) | prefer(Lewis,Anchor).  [resolve(1,a,2,a)].
85 prefer(Dodds,Miller) | prefer(Stone,Miller) | prefer(Lewis,Miller).  [resolve(1,a,4,a)].
86 prefer(Dodds,Miller) | prefer(Stone,Miller).  [copy(85),unit_del(c,10)].
88 frequent(Dodds,Harry) | frequent(Stone,Harry) | frequent(Lewis,Harry).  [resolve(5,a,7,a)].
89 frequent(Dodds,Joe) | frequent(Stone,Joe) | frequent(Lewis,Joe).  [resolve(5,a,8,a)].
93 -prefer(Stone,Miller).  [ur(81,b,9,a,c,62,a)].
94 -prefer(Stone,Anchor).  [ur(81,b,9,a,c,54,a)].
99 prefer(Dodds,Miller).  [back_unit_del(86),unit_del(b,93)].
100 prefer(Dodds,Anchor) | prefer(Lewis,Anchor).  [back_unit_del(84),unit_del(b,94)].
110 frequent(Dodds,Joe) | frequent(Stone,Joe) | -prefer(Lewis,Anchor).  [resolve(89,c,12,b)].
112 frequent(Dodds,Alice).  [resolve(99,a,11,a)].
115 -prefer(Dodds,Anchor).  [ur(81,b,99,a,c,55,a)].
121 prefer(Lewis,Anchor).  [back_unit_del(100),unit_del(a,115)].
122 frequent(Dodds,Joe) | frequent(Stone,Joe).  [back_unit_del(110),unit_del(c,121)].
123 -frequent(Stone,x) | -frequent(Lewis,y) | -prefer(Dodds,z) | -prefer(Stone,u) | -prefer(Lewis,w) # answer([Dodds,Alice,z,Stone,x,u,Lewis,y,w]).  [resolve(112,a,83,a)].
126 -frequent(Dodds,Joe).  [ur(82,b,112,a,c,76,a)].
127 -frequent(Dodds,Harry).  [ur(82,b,112,a,c,74,a)].
134 frequent(Stone,Joe).  [back_unit_del(122),unit_del(a,126)].
138 frequent(Stone,Harry) | frequent(Lewis,Harry).  [back_unit_del(88),unit_del(a,127)].
150 -frequent(Stone,Harry).  [ur(82,b,134,a,c,78,a(flip))].
158 frequent(Lewis,Harry).  [back_unit_del(138),unit_del(a,150)].
197 -frequent(Stone,x) # answer([Dodds,Alice,Miller,Stone,x,Bud,Lewis,Harry,Anchor]).  [ur(123,b,158,a,c,99,a,d,9,a,e,121,a)].
198 $F # answer([Dodds,Alice,Miller,Stone,Joe,Bud,Lewis,Harry,Anchor]).  [resolve(197,a,134,a)].
```

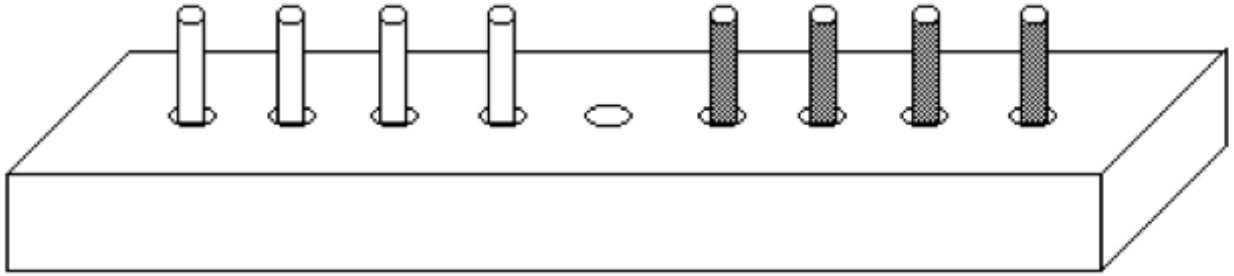
What if Solution not Unique?

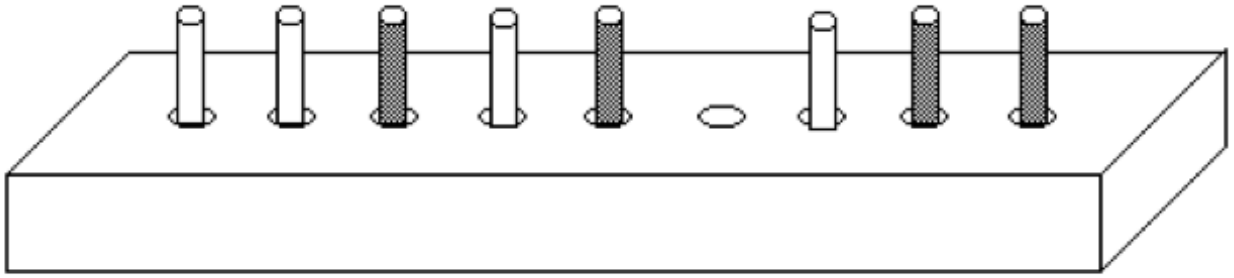
- I’m not quite sure how Prover9 handles this, if it indeed can.
- It’s predecessor, Otter, could handle it by showing the alternatives as a disjunction of answer literals.

State and Motion Puzzles and Games

- Moves in a motion puzzle or game can often be encoded as logic.
- Resolution can be used to find a solving or winning sequence of moves.

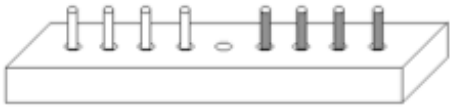
Example: Linear Peg Solitaire



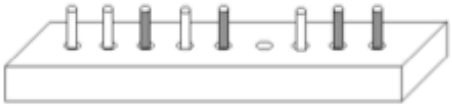


Linear Peg Solitaire Objective

- Pegs of two colors are shown in their home positions at the top.
- The objective is to completely reverse the pegs, so that each peg's original home is occupied by a peg of the opposite color.
- Allowable actions:
  - Move: A peg can be moved toward the opposite side by moving into an adjacent empty hole.
  - Jump: A peg can jump toward the opposite side over a peg of the opposite color, provided that there is a hole to receive the jumping peg.



  - Versions of the puzzle exists for 2n pegs (n of each color) and 2n+1 holes.
  - Ideally, each version can be solved.



Formulation (This will be important when we talk about **computability** later on.)

- Represent the **state** of the game with two terms.
- Say the pegs are **w** for white, **r** for red.
- Represents the pegs **away from the hole** in either direction as a composition of function symbols.
- The initial state shown is:
  s(w(w(w(w(c)))), r(r(r(r(c)))

- The second state shown is:
  s(r(w(r(w(w(c))))), w(r(r(c))

- c is a dummy constant symbol

Formulating Moves

- Simple moves (non-jump):
  - move(s(w(X), Y), s(X, w(Y))) (wm)
  - move(s(X, r(Y)), s(r(X), Y)) (rm)
- Jump moves:
  - move(s(r(w(X)), Y), s(X, r(w(Y))) (wj)
  - move(s(X), w(r(Y))), s(w(r(X)), Y)) (rj)

Formulating Reachability

- Initial state:
  reachable(s(w(w(w(w(c))))), r(r(r(r(c))))))
- State change:
  ¬reachable(X) ∨ ¬move(X, Y) ∨ reachable(Y)
- Final state:
  ¬reachable(s(r(r(r(r(c))))), w(w(w(w(c))))).

Prover9 Version

```
move(s(w(x), y), s(x, w(y))).
move(s(x, r(y)), s(r(x), y)).

move(s(r(w(x)), y), s(x, r(w(y)))).
move(s(x, w(r(y))), s(w(r(x)), y)).

reachable(s(w(w(w(w(c)))), r(r(r(r(c)))))).

-reachable(x) | -move(x, y) | reachable(y).

-reachable(s(r(r(r(r(c))))), w(w(w(w(c)))))).
```

Proof for 2 pegs of each color

1 -reachable(x) | -move(x,y) | reachable(y). [assumption].
2 move(s(r(x),y),s(x,r(y))). [assumption].
3 move(s(x,b(y)),s(b(x),y)). [assumption].
5 move(s(b(r(x)),y),s(x,b(r(y)))). [assumption].
6 move(s(x,r(b(y))),s(r(b(x)),y)). [assumption].
8 reachable(s(r(r(c)),b(b(c)))). [assumption].
9 -reachable(s(b(b(c)),r(r(c)))). [assumption].
10 -reachable(s(r(x),y)) | reachable(s(x,r(y))). [resolve(1,b,2,a)].
11 -reachable(s(x,b(y))) | reachable(s(b(x),y)). [resolve(1,b,3,a)].
13 -reachable(s(b(r(x)),y)) | reachable(s(x,b(r(y)))). [resolve(1,b,5,a)].
14 -reachable(s(x,r(b(y)))) | reachable(s(r(b(x)),y)). [resolve(1,b,6,a)].
16 -reachable(s(r(b(b(c))),r(c))). [resolve(10,b,9,a)].
17 reachable(s(r(c),r(b(b(c))))). [ur(10,a,8,a)].
22 -reachable(s(b(c),r(b(r(c))))). [resolve(16,a,14,b)].
25 -reachable(s(c,b(r(b(r(c)))))). [resolve(22,a,11,b)].
28 reachable(s(r(b(r(c))),b(c))). [ur(14,a,17,a)].
31 reachable(s(b(r(b(r(c)))),c)). [ur(11,a,28,a)].
33 -reachable(s(b(r(c)),b(r(c)))). [resolve(25,a,13,b)].
37 -reachable(s(b(r(b(r(c)))),c)). [resolve(33,a,13,b)].
38 $F. [resolve(37,a,31,a)].

Proof for 3 Pegs of Each Color

(output of Otter rather than Prover9, more traceable)

```
7  []              reachable(s(w(w(w(c))),r(r(r(c))))).
8  [hyper,7,1,4]   reachable(s(r(w(w(w(c)))),r(r(c)))).
12 [hyper,5,1,8]   reachable(s(w(w(c)),r(w(r(r(c)))))).
16 [hyper,12,1,3]  reachable(s(w(c),w(r(w(r(r(c))))))).
20 [hyper,16,1,6]  reachable(s(w(r(w(c))),w(r(r(c))))).
27 [hyper,20,1,6]  reachable(s(w(r(w(r(w(c))))),r(c))).
34 [hyper,27,1,4]  reachable(s(r(w(r(w(r(w(c)))))),c)).
39 [hyper,34,1,5]  reachable(s(r(w(r(w(c)))),r(w(c)))).
44 [hyper,39,1,5]  reachable(s(r(w(c)),r(w(r(w(c)))))).
51 [hyper,44,1,5]  reachable(s(c,r(w(r(w(r(w(c))))))).
57 [hyper,51,1,4]  reachable(s(r(c),w(r(w(r(w(c))))))).
63 [hyper,57,1,6]  reachable(s(w(r(r(c))),w(r(w(c))))).
69 [hyper,63,1,6]  reachable(s(w(r(w(r(r(c))))),w(c))).
72 [hyper,69,1,3]  reachable(s(r(w(r(r(c)))),w(w(c)))).
75 [hyper,72,1,5]  reachable(s(r(r(c)),r(w(w(w(c)))))).
77 [hyper,75,1,4]  reachable(s(r(r(r(c))),w(w(w(c))))).
78 [binary,77.1,2.1] $F.
```

Pegs vs. # of Moves in Solution

Pegs of Each Color	# of Moves
1	3
2	8
3	15
4	24
5	35
n	$n^2 + 2n$

Determining the Move Sequence

- The previous proofs only showed that the puzzle could be solved for those variations.
- The actual move sequence would have to be dug out from the proof steps.
- We can modify the rules so that the move sequence is obtained as a **byproduct**.

Determining the Move Sequence

- Use function composition to represent accumulated move sequence.
- Revised rules (4-pegs, where specific):
  - move(s(w(x), y), s(x, w(y)), z, wm(z)).
  - move(s(x, r(y)), s(r(x), y), z, rm(z)).
  - move(s(r(w(x)), y), s(x, r(w(y))), z, wj(z)).
  - move(s(x, w(r(y))), s(w(r(x)), y), z, rj(z)).
  - reachable(s(w(w(w(w(c)))), r(r(r(r(c))))), d).
  - -reachable(x, z) | -move(x, y, z, zz) | reachable(y, zz).
  - -reachable(s(r(r(r(r(c)))), w(w(w(w(c))))), z) #answer(z).

Move Sequence Read Inside-Out

- For 4 pegs of each color:
  $answer(rm(wj(wm(rj(rj(rm(wj(wj(wj(wm(rj(rj(rj(rj(wm(wj
  (wj(wj(rm(rj(rj(wm(wj(rm(d)))))))))))))))))))))).

- The sequence is:
  rm wj wm rj rj rm wj wj wj wm rj rj
  rj rj wm wj wj wj rm rj rj wm wj rm

- (For this puzzle, the move sequence is coincidentally a palindrome.)