EECS 3401: Intro to AI & LP
Inference in First-Order Logic

- Required Readings: R & N 9.1, 9.2, and 9.5
- Resolution Proofs.
  - Part I: Convert to clausal form
  - Part II: Dealing with variables (unification).
  - Part III: Constructing Resolution Proofs.

Computing logical consequences

- We want procedures for computing logical consequences that can be implemented in our programs.
- This would allow us to reason with our knowledge
  - Represent the knowledge as logical formulas
  - Apply procedures for generating logical consequences
- These procedures are called proof procedures.

Proof Procedures

- Interesting, proof procedures work by simply manipulating formulas. They do not know or care anything about interpretations.
- Nevertheless they respect the semantics of interpretations!
- We will develop a proof procedure for first-order logic called resolution.
  - Resolution is the mechanism used by PROLOG

Properties of Proof Procedures

- Before presenting the details of resolution, we want to look at properties we would like to have in a (any) proof procedure.
- We write KB $\vdash$ f to indicate that f can be proved from KB (the proof procedure used is implicit).

Properties of Proof Procedures

- Soundness
  - KB ⊢ f → KB ⊨ f

    i.e all conclusions arrived at via the proof procedure are correct: they are logical consequences.
- Completeness
  - KB ⊨ f → KB ⊢ f

    i.e. every logical consequence can be generated by the proof procedure.
- Note proof procedures are computable, but they might have very high complexity in the worst case. So completeness is not necessarily achievable in practice.

Resolution

- Clausal form.
  - Resolution works with formulas expressed in clausal form.
  - A literal is an atomic formula or the negation of an atomic formula. dog(fido), ¬cat(fido)
  - A clause is a disjunction of literals:
    - ¬owns(fido,fred) ∨ ¬dog(fido) ∨ person(fred)
    - We write
      (¬owns(fido,fred), ¬dog(fido), person(fred))
  - A clausal theory is a conjunction of clauses.

Resolution

- Prolog Programs
  - Prolog programs are clausal theories.
  - However, each clause in a Prolog program is Horn.
  - A horn clause contains at most one positive literal.
    - The horn clause

      ¬q1 ∨ ¬q2 ∨… ∨ ¬qn ∨ p

      is equivalent to

      q1 ∧ q2 ∧… ∧ qn ⇒ p

      and is written as the following rule in Prolog:

      p :- q1 , q2 ,… ,qn

Resolution Rule for Ground Clauses

- The resolution proof procedure consists of only one simple rule:
  - From the two clauses
    - (P, Q1, Q2, …, Qk)
    - (¬P, R1, R2, …, Rn)
  - We infer the new clause
    - (Q1, Q2, …, Qk, R1, R2, …, Rn)
  - Example:
    - (¬largerThan(clyde,cup), ¬fitsIn(clyde,cup)
    - (fitsIn(clyde,cup))

    ⇒ ¬largerThan(clyde,cup)

Resolution Proof: Forward chaining

- Logical consequences can be generated from the resolution rule in two ways:

1. Forward Chaining inference.
   - If we have a sequence of clauses C1, C2, …, Ck
   - Such that each Ci is either in KB or is the result of a resolution step involving two prior clauses in the sequence.
   - We then have that KB ⊢ Ck.

   Forward chaining is sound so we also have KB ⊨ Ck

Resolution Proof: Refutation proofs

2. Refutation proofs.
   - We determine if KB ⊢ f by showing that a contradiction can be generated from KB ∧ ¬f.
   - In this case a contradiction is an empty clause ().
   - We employ resolution to construct a sequence of clauses C1, C2, …, Cm such that
     - Ci is in KB ∧ ¬f, or is the result of resolving two previous clauses in the sequence.
     - Cm = () i.e. its the empty clause.

Resolution Proof: Refutation proofs

- If we can find such a sequence C1, C2, …, Cm=(), we have that
  - KB ⊢ f.
  - Furthermore, this procedure is sound so
    - KB ⊨ f
- And the procedure is also complete so it is capable of finding a proof of any f that is a logical consequence of KB. I.e.
  - If KB ⊨ f then we can generate a refutation from KB ∧ ¬f

Resolution Proofs Example

Want to prove likes(clyde,peanuts) from:

1. (elephant(clyde), giraffe(clyde))
2. (¬elephant(clyde), likes(clyde,peanuts))
3. (¬giraffe(clyde), likes(clyde,leaves))
4. ¬likes(clyde,leaves)

Forward Chaining Proof:

- 3&4 → ¬giraffe(clyde) [5.]
- 5&1 → elephant(clyde) [6.]
- 6&2 → likes(clyde,peanuts) [7.] ✓

Resolution Proofs Example

1. (elephant(clyde), giraffe(clyde))
2. (¬elephant(clyde), likes(clyde,peanuts))
3. (¬giraffe(clyde), likes(clyde,leaves))
4. ¬likes(clyde,leaves)

Refutation Proof:
First add negation of query to KB:
5. ¬likes(clyde,peanuts)

- 5&2 → ¬elephant(clyde) [6.]
- 6&1 → giraffe(clyde) [7.]
- 7&3 → likes(clyde,leaves) [8.]
- 8&4 → () ✓

Resolution Proofs

- Proofs by refutation have the advantage that they are easier to find.
  - They are more focused to the particular conclusion we are trying to reach.
- To develop a complete resolution proof procedure for First-Order Logic we need :
  1. A way of converting KB and f (the query) into clausal form. [we focus on this in the rest of this lecture]
  2. A way of doing resolution even when we have variables (unification). [this will be covered in the next lecture]

Conversion to Clausal Form

To convert the KB into Clausal form we perform the following 8-step procedure:

1. **Eliminate Implications.**
2. **Move Negations inwards (and simplify ¬¬).**
3. **Standardize Variables.**
4. **Skolemize.**
5. **Convert to Prenex Form.**
6. **Distribute conjunctions over disjunctions.**
7. **Flatten nested conjunctions and disjunctions.**
8. **Convert to Clauses.**

C-T-C-F: Eliminate implications

We use this example to show each step:

$$\forall X.p(X) \rightarrow \Big(\forall Y.p(Y) \rightarrow p(f(X,Y)) \wedge \neg(\forall Y.\ \neg q(X,Y) \wedge p(Y))\Big)$$

1. Eliminate implications: $A \rightarrow B$ ➔ $\neg A \vee B$

$$\forall X.\ \neg p(X) \vee \Big(\forall Y.\neg p(Y) \vee p(f(X,Y)) \wedge \neg(\forall Y.\ \neg q(X,Y) \wedge p(Y)) \Big)$$

C-T-C-F: Move ¬ Inwards

$\forall X. \neg p(X)$
$\quad \vee (\forall Y. \neg p(Y) \vee p(f(X,Y))$
$\quad\quad \wedge \neg(\forall Y. \neg q(X,Y) \wedge p(Y)))$

2. Move Negations Inwards (and simplify ¬¬)

$\forall X. \neg p(X)$
$\quad \vee (\forall Y. \neg p(Y) \vee p(f(X,Y))$
$\quad\quad \wedge \exists Y. q(X,Y) \vee \neg p(Y))$

C-T-C-F: : ¬ continue…

Rules for moving negations inwards

- $\neg(A \wedge B) \rightarrow \neg A \vee \neg B$
- $\neg(A \vee B) \rightarrow \neg A \wedge \neg B$
- $\neg\forall X. f \rightarrow \exists X. \neg f$
- $\neg\exists X. f \rightarrow \forall X. \neg f$
- $\neg\neg A \rightarrow A$

C-T-C-F: Standardize Variables

$\forall X. \neg p(X)$
$\quad \vee (\forall Y. \neg p(Y) \vee p(f(X,Y))$
$\quad\quad \wedge \exists Y. q(X,Y) \vee \neg p(Y))$

3. Standardize Variables (Rename variables so that each quantified variable is unique)

$\forall X. \neg p(X)$
$\quad \vee (\forall Y.(\neg p(Y) \vee p(f(X,Y))$
$\quad\quad \wedge \exists Z. q(X,Z) \vee \neg p(Z))$

C-T-C-F: Skolemize

$\forall X. \neg p(X)$
$\quad \vee (\forall Y. \neg p(Y) \vee p(f(X,Y))$
$\quad\quad \wedge \exists Z. q(X,Z) \vee \neg p(Z))$

4. Skolemize (Remove existential quantifiers by introducing new function symbols).

$\forall X. \neg p(X)$
$\quad \vee (\forall Y. \neg p(Y) \vee p(f(X,Y))$
$\quad\quad \wedge q(X,g(X)) \vee \neg p(g(X)))$

C-T-C-F: Skolemization continue…

Consider ∃Y.elephant(Y) ∧ friendly(Y)

- This asserts that there is some individual (binding for Y) that is both an elephant and friendly.
- To remove the existential, we **invent** a name for this individual, say ***a.*** This is a new constant symbol not equal to any previous constant symbols to obtain:

  elephant(**a**) ∧ friendly(**a**)
- This is saying the same thing, since we do not know anything about the new constant **a**.

C-T-C-F: Skolemization continue

- It is essential that the introduced symbol "a" is **new.** Else we might know something else about "a" in KB.
- If we did know something else about "a" we would be asserting more than the existential.
- In original quantified formula we know nothing about the variable "Y". Just what was being asserted by the existential formula.

C-T-C-F: Skolemization continue

Now consider ∀X∃Y. loves(X,Y).

- This formula claims that for every *X* there is some *Y* that X loves (perhaps a different Y for each X).
- Replacing the existential by a new constant won't work

  ∀X.loves(X,**a**).

  Because this asserts that there is a **particular** individual "a" loved by every X.
- To properly convert existential quantifiers scoped by universal quantifiers we must use **functions** not just constants.

C-T-C-F: Skolemization continue

- We must use a function that mentions every universally quantified variable <u>that scopes the existential</u>.
- In this case *X* scopes *Y* so we must replace the existential *Y* by a function of *X*

  ∀X. loves(X,g(X)).

  where g is a **new** function symbol.
- This formula asserts that for every X there is some individual (given by g(X)) that X loves. g(X) can be different for each different binding of X.

C-T-C-F: Skolemization Examples

- ∀XYZ ∃W.r(X,Y,Z,W) ➔ ∀XYZ.r(X,Y,Z,h1(X,Y,Z))
- ∀XY∃W.r(X,Y,g(W)) ➔ ∀XY.r(X,Y, g(h2(X,Y)))
- ∀XY∃W∀Z.r(X,Y,W) ∧ q(Z,W)

  ➔ ∀XYZ.r(X,Y,h3(X,Y)) ∧ q(Z,h3(X,Y))

C-T-C-F: Convert to prenex

∀X. ¬p(X)
∨ (∀Y.¬p(Y) ∨ p(f(X,Y))
∧ q(X,g(X)) ∨ ¬p(g(X)))

5. Convert to prenex form. (Bring all quantifiers to the front—only universals, each with different name).

∀X∀Y. ¬p(X)
∨ (¬p(Y) ∨ p(f(X,Y))
∧ q(X,g(X)) ∨ ¬p(g(X)))

C-T-C-F: Conjunctions over disjunctions

∀X∀Y. ¬p(X)
∨ (¬p(Y) ∨ p(f(X,Y))
∧ q(X,g(X)) ∨ ¬p(g(X)))

6. Conjunctions over disjunctions
A ∨ (B ∧ C) ➔ (A ∨ B) ∧ (A ∨ C)

∀XY. ¬p(X) ∨ ¬p(Y) ∨ p(f(X,Y))
∧ ¬p(X) ∨ q(X,g(X)) ∨ ¬p(g(X))

C-T-C-F: flatten & convert to clauses

7. **Flatten nested conjunctions and disjunctions.**
(A ∨ (B ∨ C)) ➔ (A ∨ B ∨ C)

8. **Convert to Clauses** (remove quantifiers and break apart conjunctions).
∀XY. ¬p(X) ∨ ¬p(Y) ∨ p(f(X,Y))
∧ ¬p(X) ∨ q(X,g(X)) ∨ ¬p(g(X))

a) ¬p(X) ∨ ¬p(Y) ∨ p(f(X,Y))
b) ¬p(X) ∨ q(X,g(X)) ∨ ¬p(g(X))

Unification

- Ground clauses are clauses with no variables in them. For ground clauses we can use syntactic identity to detect when we have a P and ¬P pair.

- What about variables can the clauses
  - (P(john), Q(fred), R(X))
  - (¬P(Y), R(susan), R(Y))

  Be resolved?

Unification.

- Intuitively, once reduced to clausal form, all remaining variables are universally quantified. So, implicitly (¬P(Y), R(susan), R(Y)) represents clauses like
  - (¬P(fred), R(susan), R(fred))
  - (¬P(john), R(susan), R(john))
  - …

- So there is a "specialization" of this clause that can be resolved with (P(john), Q(fred), R(X)

Unification.

- We want to be able to match conflicting literals, even when they have variables. This matching process automatically determines whether or not there is a "specialization" that matches.

- We don't want to over specialize!

Unification.

- (¬p(X), s(X), q(fred))
- (p(Y), r(Y))
- Possible resolvants
  - (s(john), q(fred), r(john)) {Y=X, X=john}
  - (s(sally), q(fred), r(sally)) {Y=X, X=sally}
  - (s(X), q(fred), r(X)) {Y=X}

- The last resolvant is "most-general", the other two are specializations of it.
- We want to keep the most general clause so that we can use it future resolution steps.

Unification.

- unification is a mechanism for finding a "most general" matching.
- First we consider substitutions.
  - A substitution is a finite set of equations of the form

    (V = t)

    where V is a variable and t is a term not containing V. (t might contain other variables).

Substitutions.

- We can apply a substitution σ to a formula f to obtain a new formula $f\sigma$ by simultaneously replacing every variable mentioned in the left hand side of the substitution by the right hand side.

  p(X,g(Y,Z))[X=Y, Y=f(a)] ➔ p(Y,g(f(a),Z))

- Note that the substitutions are not applied sequentially, i.e., the first Y is not subsequently replaced by f(a).

Substitutions.

- We can compose two substitutions. θ and σ to obtain a new substition $\theta\sigma$.

Let $\theta = \{X_1=s_1, X_2=s_2, \ldots, X_m=s_m\}$
$\sigma = \{Y_1=t_1, Y_2=t_2, \ldots, Y_k=s_k\}$

To compute $\theta\sigma$

1. $S = \{X_1=s_1\sigma, X_2=s_2\sigma, \ldots, X_m=s_m\sigma, Y_1=t_1, Y_2=t_2, \ldots, Y_k=s_k\}$

   we apply σ to each RHS of θ and then add all of the equations of σ.

Substitutions.

1. $S = \{X_1=s_1\sigma, X_2=s_2\sigma, \ldots, X_m=s_m\sigma, Y_1=t_1, Y_2=t_2, \ldots, Y_k=s_k\}$
2. Delete any identities, i.e., equations of the form V=V.
3. Delete any equation $Y_i=s_i$ where Y_i is equal to one of the X_j in θ.

The final set S is the composition $\theta\sigma$.

Composition Example.

θ = {X=f(Y), Y=Z}, σ = {X=a, Y=b, Z=Y}

$\theta\sigma$

Substitutions.

- The empty substitution ε = {} is also a substitution, and it acts as an identity under composition.
- More importantly substitutions when applied to formulas are associative:

$$(f\theta)\sigma = f(\theta\sigma)$$

- Composition is simply a way of converting the sequential application of a series of substitutions to a single simultaneous substitution.

Unifiers.

- A unifier of two formulas f and g is a substitution σ that makes f and g syntactically identical.
- Not all formulas can be unified—substitutions only affect variables.

p(f(X),a)   p(Y,f(w))

- This pair cannot be unified as there is no way of making a = f(w) with a substitution.

MGU.

- A substitution σ of two formulas f and g is a Most General Unifier (MGU) if

1. σ is a unifier.
2. For every other unifier θ of f and g there must exist a third substitution λ such that

$$\theta = \sigma\lambda$$

- This says that every other unifier is "more specialized than σ. The MGU of a pair of formulas f and g is unique up to renaming.

MGU.

p(f(X),Z)   p(Y,a)

1. σ = {Y = f(a), X=a, Z=a} is a unifier.

   p(f(X),Z)σ =
   p(Y,a)σ =

   But it is not an MGU.

2. θ = {Y=f(X), Z=a} is an MGU.
   p(f(X),Z) θ =
   p(Y,a) θ =

MGU.

p(f(X),Z)   p(Y,a)

3. $\sigma = \theta\lambda$, where λ={X=a}

   σ = {Y = f(a), X=a, Z=a}
   λ ={X=a}
   $\theta\lambda$ =

MGU.

- The MGU is the “least specialized” way of making clauses with universal variables match.
- We can compute MGUs.
- Intuitively we line up the two formulas and find the first sub-expression where they disagree. The pair of subexpressions where they first disagree is called the disagreement set.
- The algorithm works by successively fixing disagreement sets until the two formulas become syntactically identical.

MGU.

To find the MGU of two formulas f and g.

1. k = 0; σ_0 = {}; S_0 = {f,g}
2. If S_k contains an identical pair of formulas stop, and return σ_k as the MGU of f and g.
3. Else find the disagreement set D_k={e_1,e_2} of S_k
4. If e_1 = V a variable, and e_2 = t a term not containing V (or vice-versa) then let
   $\sigma_{k+1} = \sigma_k$ {V=t} (Compose the additional substitution)
   $S_{k+1} = S_k${V=t} (Apply the additional substitution)
   k = k+1
   GOTO 2
5. Else stop, f and g cannot be unified.

MGU Example 1.

S_0 = {p(f(a), g(X)) ; p(Y,Y)}

MGU Example 2.

S_0 = {p(a,X,h(g(Z))) ; p(Z,h(Y),h(Y))}

MGU Example 3.

S_0 = {p(X,X) ; p(Y,f(Y))}

Non-Ground Resolution

- Resolution of non-ground clauses. From the two clauses

  (L, Q1, Q2, …, Qk)
  (¬M, R1, R2, …, Rn)

  Where there exists σ a MGU for L and M.

  We infer the new clause

  (Q1σ, …, Qkσ, R1σ, …, Rnσ)

Non-Ground Resolution E.G.

1. (p(X), q(g(X)))
2. (r(a), q(Z), ¬p(a))

   L=p(X); M=p(a)
   σ = {X=a}

3. R[1a,2c]{X=a} (q(g(a)), r(a), q(Z))

The notation is important.

- "R" means resolution step.
- "1a" means the first (a-th) literal in the first clause i.e. p(X).
- "2c" means the third (c-th) literal in the second clause, ¬p(a).
  - 1a and 2c are the "clashing" literals.
- {X=a} is the substitution applied to make the clashing literals identical.

Resolution Proof Example

"Some patients like all doctors. No patient likes any quack. Therefore no doctor is a quack."

Resolution Proof Step 1.
Pick symbols to represent these assertions.

p(X): X is a patient
d(x): X is a doctor
q(X): X is a quack
l(X,Y): X likes Y

Resolution Proof Example

Resolution Proof Step 2.
Convert each assertion to a first-order formula.

1. Some patients like all doctors.

F1.

Resolution Proof Example

2. No patient likes any quack

F2.

3. Therefore no doctor is a quack.

Query.

Resolution Proof Example

Resolution Proof Step 3.
Convert to Clausal form.

F1.

F2.

Negation of Query.

Resolution Proof Example

Resolution Proof Step 4.
Resolution Proof from the Clauses.

1. p(a)
2. (¬d(Y), l(a,Y))
3. (¬p(Z), ¬q(R), ¬l(Z,R))
4. d(b)
5. q(b)

Resolution Proof Example

Answer Extraction.

- The previous example shows how we can answer true-false questions. With a bit more effort we can also answer “fill-in-the-blanks” questions (e.g., what is wrong with the car?).
- As in Prolog we use free variables in the query where we want the fill in the blanks. We simply need to keep track of the binding that these variables received in proving the query.
  - parent(art, jon) -is art one of jon’s parents?
  - parent(X, jon) -who is one of jon’s parents?

Answer Extraction.

- A simple bookkeeping device is to use an predicate symbol answer(X,Y,…) to keep track of the bindings automatically.
- To answer the query parent(X,jon), we construct the clause
  (¬ parent(X,jon), answer(X))
- Now we perform resolution until we obtain a clause consisting of only answer literals (previously we stopped at empty clauses).

Answer Extraction: Example 1

1. father(art, jon)
2. father(bob,kim)
3. (¬father(Y,Z), parent(Y,Z))
   i.e. *all fathers are parents*
4. (¬ parent(X,jon), answer(X))
   i.e. the query is: who is parent of jon?

Here is a resolution proof:

5. R[4,3b]{Y=X,Z=jon}
   (¬father(X,jon), answer(X))
6. R[5,1]{X=art} answer(art)

so art is parent of jon

Answer Extraction: Example 2

1. (father(art, jon), father(bob,jon)) //either bob or art is parent of jon
2. father(bob,kim)
3. (¬father(Y,Z), parent(Y,Z)) //i.e. *all fathers are parents*
4. (¬ parent(X,jon), answer(X)) //i.e. query is parent(X,jon)

Here is a resolution proof:

5. R[4,3b]{Y=X,Z=jon} (¬father(X,jon), answer(X))
6. R[5,1a]{X=art} (father(bob,jon), answer(art))
7. R[6,3b] {Y=bob,Z=jon}
   (parent(bob,jon), answer(art))
8. R[7,4] {X=bob} (answer(bob), answer(art))

A disjunctive answer: either bob or art is parent of jon.

Factoring (optional)

1. (p(X), p(Y)) // ∀ X.∀ Y. ¬p(X) → p(Y)
2. (¬p(V), ¬p(W)) // ∀ V.∀W. p(V) → ¬p(W)

- These clauses are intuitively contradictory, but following the strict rules of resolution only we obtain:

3. R[1a,2a](X=V) (p(Y), ¬p(W))
   Renaming variables: (p(Q), ¬p(Z))
4. R[3b,1a](X=Z) (p(Y), p(Q))

No way of generating empty clause!

Factoring is needed to make resolution complete, without it resolution is incomplete!

Factoring (optional).

- If two or more literals of a clause C have an mgu θ, then Cθ with all duplicate literals removed is called a **factor** of C.
- C = (p(X), p(f(Y)), ¬q(X))
  θ = {X=f(Y)}
  Cθ = (p(f(Y)), p(f(Y)), ¬q(f(Y))) ➔ (p(f(Y)), ¬q(f(Y)) is a factor

Adding a factor of a clause can be a step of proof:

1. (p(X), p(Y))
2. (¬p(V), ¬p(W))
3. f[1ab]{X=Y} p(Y)
4. f[2ab]{V=W} ¬p(W)
5. R[3,4]{Y=W} ().

Prolog

- Prolog search mechanism *(without not and cut)* is simply an instance of resolution, except
  1. Clauses are Horn (only one positive literal)
  2. Prolog uses a specific depth first strategy when searching for a proof. (Rules are used first mentioned first used, literals are resolved away left to right).

Prolog

- Append:

1. append([], Z, Z)
2. append([E1 | R1], Y, [E1 | Rest]) :-
   append(R1, Y, Rest)

Note:

- The second is actually the clause
  (append([E1|R1], Y, [E1|Rest]) , ¬append(R1,Y,Rest))
- [ ] is a constant (the empty list)
- [X | Y] is cons(X,Y).
- So [a,b,c] is short hand for cons(a,cons(b,cons(c,[])))

Prolog: Example of proof

- Try to prove : append([a,b], [c,d], [a,b,c,d]):

1. append([], Z, Z)
2. (append([E1|R1], Y, [E1|Rest]),
   ¬append(R1,Y,Rest))
3. ¬append([a,b], [c,d], [a,b,c,d])
4. R[3,2a]{E1=a, R1=[b], Y=[c,d], Rest=[b,c,d]}
   ¬append([b], [c,d], [b,c,d])
5. R[4,2a]{E1=b, R1=[], Y=[c,d], Rest=[c,d]}
   ¬append([], [c,d], [c,d])
6. R[5,1]{Z=[c,d]} ()

Review: One Last Example!

Consider the following English description

- Whoever can read is literate.
- Dolphins are not literate.
- Flipper is an intelligent dolphin.

- Who is intelligent but cannot read.

Example: pick symbols & convert to first-order formula

- Whoever can read is literate.
  ∀ X. read(X) → lit(X)
- Dolphins are not literate.
  ∀ X. dolp(X) → ¬ lit(X)
- Flipper is an intelligent dolphin
  dolp(flipper) ∧ intell(flipper)

- Who is intelligent but cannot read?
  ∃ X. intell(X) ∧ ¬ read(X).

Example: convert to clausal form

- ∀X. read(X) → lit(X)
  (¬read(X), lit(X))
- Dolphins are not literate.
  ∀X. dolp(X) → ¬ lit(X)
  (¬dolp(X), ¬lit(X))
- Flipper is an intelligent dolphin.
  dolp(flipper)
  intell(flipper)
- who are intelligent but cannot read?
  ∃ X. intell(X) ∧ ¬read(X).
  ➔ ∀ X. ¬ intell(X) ∨ read(X)
  ➔ (¬intell(X), read(X), answer(X))

Example: do the resolution proof

1. (¬read(X), lit(X))
2. (¬dolp(X), ¬lit(X))
3. dolp(flip)
4. intell(flip)
5. (¬intell(X), read(X),answer(X))

6. R[5a,4] X=flip. (read(flip), answer(flip))
7. R[6,1a] X=flip. (lit(flip), answer(flip))
8. R[7,2b] X=flip. (¬dolp(flip), answer(flip))
9. R[8,3] answer(flip)

so flip is intelligent but cannot read!