Prolog and the Resolution Method

The Logical Basis of Prolog

Background

- **Prolog is based on the resolution proof method developed by Robinson in 1966.**

Background – 2

- Prolog is based on the resolution proof method developed by Robinson in 1966.
- **Complete proof system with only one rule.**
  - **If something can be proven from a set of logical formulae, the method finds it.**

Background – 3

◊ Prolog is based on the **resolution proof** method developed by Robinson in 1966.

◊ Complete proof system with only one rule.

  » If something can be proven from a set of logical formulae, the method finds it.

◊ Correct

  » **Only theorems will be proven, nothing else.**

Background – 4

- Prolog is based on the resolution proof method developed by Robinson in 1966.
- Complete proof system with only one rule.
  - If something can be proven from a set of logical formulae, the method finds it.
- Correct
  - Only theorems will be proven, nothing else.
- **Proof by contradiction**
  - **Add negation of a purported theorem to a body of axioms and previous proven theorems**
  - **Show resulting system is contradictory**

Propositional Logic

◊ **Infinite list of propositional variables**

» **$a, b, \ldots, z, p_1 \ldots p_n, q_1 \ldots q_r, \ldots$**

Propositional Logic – 2

- Infinite list of propositional variables
  - $a, b, \ldots, z, p_1 \ldots p_n, q_1 \ldots q_r, \ldots$
- **Every variable represents 0 or 1 (True or False)**

Propositional Logic – 3

- ◊ Infinite list of propositional variables
  - » a, b, … , z, p_1 … p_n , q_n … q_r, …
- ◊ Every variable represents 0 or 1 (True or False)
- ◊ **Logical connectives**
  - » **~ (not) $\wedge$ (and) $\vee$ (or) $\rightarrow$ (implies) $\leftrightarrow$ (iff)**

Propositional Logic – 4

- Infinite list of propositional variables
  - a, b, … , z, p_1 … p_n , q_n … q_r, …
- Every variable represents 0 or 1 (True or False)
- Logical connectives
  - ~ (not) $\wedge$ (and) $\vee$ (or) $\rightarrow$ (implies) $\leftrightarrow$ (iff)
- **The set of formula's of propositional logic is the smallest set, FOR, such that**
  - **Every propositional variable is in FOR**
  - **If A and B are elements of FOR then**
    ~ A $A \wedge B$ $A \vee B$ $A \rightarrow B$ $A \leftrightarrow B$
    are elements of FOR

Propositional clauses – informal

- Have a collection of clauses in **conjunctive normal form**
  - » **Each clause is a set of propositions connected with or**
  - » **Propositions can be negated (use not ~)**
  - » **set of clauses implicitly and'ed together**

- Example

  A or B

  C or D or ~E

  F

  ==>

  (A or B) and (C or D or ~E) and F

Clausal Form

◊ A clause is an expression of the following form, called **clausal form**

$$\underbrace{l_0, l_1, l_2, \ldots l_k}_{\text{commas are disjunctions}} \leftarrow \underbrace{d_0, d_1, d_2, \ldots d_m}_{\text{commas are conjunctions}}$$

commas are disjunctions

commas are conjunctions

Clausal Form – 2

◊ We have the following clausal form

commas are disjunctions $l_0, l_1, l_2, \ldots l_k \leftarrow d_0, d_1, d_2, \ldots d_m$ **commas are conjunctions**

The following equivalence holds

$$a \leftarrow b \quad \equiv \quad a \vee \sim b$$

Clausal Form – 3

◊ We have the following clausal form

commas are disjunctions

$$l_0, l_1, l_2, \ldots l_k \leftarrow d_0, d_1, d_2, \ldots d_m$$

commas are conjunctions

The following equivalence holds

$$a \leftarrow b \quad \equiv \quad a \vee \sim b$$

As a consequence the clausal form can be written as

$$l_0 \vee l_1 \vee l_2 \vee \ldots \vee l_k \vee \sim(d_0 \wedge d_1 \wedge d_2 \wedge \ldots \wedge d_m)$$

Clausal Form – 4

◊ We have the following clausal form

commas are disjunctions $l_0, l_1, l_2, \ldots l_k \leftarrow d_0, d_1, d_2, \ldots d_m$ **commas are conjunctions**

The following equivalence holds

$$a \leftarrow b \quad \equiv \quad a \vee \sim b$$

As a consequence the clausal form can be written as

$$l_0 \vee l_1 \vee l_2 \vee \ldots \vee l_k \vee \sim(d_0 \wedge d_1 \wedge d_2 \wedge \ldots \wedge d_m)$$

Using de' Morgans law

$$l_0 \vee l_1 \vee l_2 \vee \ldots \vee l_k \vee \sim d_0 \vee \sim d_1 \vee \sim d_2 \vee \ldots \vee \sim d_m$$

Conjunctive Normal Form

◊ If S = { c_0 , c_1 , c_2 , … c_k } are a set of clauses then the representation of S is the formula

$$\alpha = (\alpha_{c0} \wedge \alpha_{c1} \wedge \alpha_{c2} \wedge \ldots \wedge \alpha_{ck})$$

Conjunctive Normal Form – 2

- If S = { $c_0 , c_1 , c_2 , \ldots c_k$ } are a set of clauses then the representation of S is the formula

$$\alpha = (\alpha_{c0} \wedge \alpha_{c1} \wedge \alpha_{c2} \wedge \ldots \wedge \alpha_{ck})$$

- α_{ci} is a disjunction of variables and their negations

$$l_0 \vee l_1 \vee l_2 \vee \ldots \vee l_k \vee {\sim}d_0 \vee {\sim}d_1 \vee {\sim}d_2 \vee \ldots \vee {\sim}d_m$$

Conjunctive Normal Form – 3

◊ If S = { c_0 , c_1 , c_2 , … c_k } are a set of clauses then the representation of S is the formula

$$\alpha = (\alpha_{c0} \wedge \alpha_{c1} \wedge \alpha_{c2} \wedge \ldots \wedge \alpha_{ck})$$

◊ α_{ci} is a disjunction of variables and their negations

$$l_0 \vee l_1 \vee l_2 \vee \ldots \vee l_k \vee \sim d_0 \vee \sim d_1 \vee \sim d_2 \vee \ldots \vee \sim d_m$$

◊ α is a conjunction of these disjunctions

Conjunctive Normal Form – 4

◊ If S = { c_0 , c_1 , c_2 , … c_k } are a set of clauses then the representation of S is the formula

$$\alpha = (\alpha_{c0} \wedge \alpha_{c1} \wedge \alpha_{c2} \wedge \ldots \wedge \alpha_{ck})$$

◊ α_{ci} is a disjunction of variables and their negations

$$l_0 \vee l_1 \vee l_2 \vee \ldots \vee l_k \vee {\sim}d_0 \vee {\sim}d_1 \vee {\sim}d_2 \vee \ldots \vee {\sim}d_m$$

◊ α is a conjunction of these disjunctions

◊ α is in CNF (conjunctive normal form)

Conjunctive Normal Form – 5

◊ If S = { c_0 , c_1 , c_2 , … c_k } are a set of clauses then the representation of S is the formula

$$\alpha = (\alpha_{c0} \wedge \alpha_{c1} \wedge \alpha_{c2} \wedge \ldots \wedge \alpha_{ck})$$

◊ α_{ci} is a disjunction of variables and their negations

$$l_0 \vee l_1 \vee l_2 \vee \ldots \vee l_k \vee \sim d_0 \vee \sim d_1 \vee \sim d_2 \vee \ldots \vee \sim d_m$$

◊ α is a conjunction of these disjunctions

◊ α is in CNF (conjunctive normal form)

Every formula can be converted to CNF

Contradiction in a set of clauses

- The set $\{ p \wedge \sim p \}$ is a contradiction of clauses

Contradiction in a set of clauses – 2

- ◊ The set $\{ p \wedge \sim p \}$ is a contradiction of clauses
- ◊ In clausal form this is

$$p \leftarrow$$
$$\leftarrow p$$

Contradiction in a set of clauses – 3

◊ The set $\{ p \wedge \sim p \}$ is a contradiction of clauses

◊ In clausal form this is

$$p \leftarrow$$
$$\leftarrow p$$

◊ We say that resolving upon p gives [ ] the empty clause, which is false.

Propositional case – Resolution

◊ What if there is a contradiction in the set of clauses

Propositional case – Resolution – 2

◊ What if there is a contradiction in the set of clauses

◊ Example – only one clause

P

Propositional case – Resolution – 3

◊ What if there is a contradiction in the set of clauses

◊ Example – only one clause

P

◊ Add ~P to the set of clauses

P
~P

==>

P and ~P

==>

[ ]   -- null the empty clause is false

Propositional case – Resolution – 4

- What if there is a contradiction in the set of clauses
- Example – only one clause

  P

- Add ~P to the set of clauses

  P
  ~P
  ==>
  P and ~P
  ==>
  [ ] -- null the empty clause is false

- Think of **P** and **~P** canceling each other out of existence

Resolution rule

◊ Given the clause

Q or ~R

◊ and the clause

R or P

◊ then resolving the two clauses is the following

(Q or ~R) and (R or P)

==>

P or Q **-- new clause that can be added to the set**

◊ Combining two clauses with a positive proposition and its negation (called **literals**) leads to adding a new clause to the set of clauses consisting of all the literals in both parent clauses except for the literals resolved on

Resolution rule – 2

- Given the clause

  L_1 or L_2 or … or L_p or ~R

- and the clause

  R or K_1 or K_2 or … or K_q

- then resolving the two clauses is the following

  (L_1 or L_2 or … or L_p or ~R) and (R or K_1 or K_2 or … or K_q)

  ==>

  (L_1 or L_2 or … or L_p or K_1 or K_2 or … or K_q)

  A new clause that can be added to the set

Resolution method

◊ Combine clauses using resolution to find the empty clause

» **Implies one or more of the clauses is false.**

◊ Given the clauses

1	**P**
2	**~P or Q**
3	**~ Q or R**
4	**~R**

◊ Can resolve as follows

5	**P and (~P or Q) ==> Q**	**resolve 1 and 2**
6	**Q and (~Q or R) ==> R**	**resolve 5 and 3**
7	**R and ~R ==> []**	**resolve 6 and 4**

Proving a theorem

1 Given a set of non contradictory clauses
   – assume the set of clauses is true

P
~P or Q
~ Q or R

Proving a theorem – 2

1 Given a set of non contradictory clauses
– assume the set of clauses is true

P
~P or Q
~ Q or R

2 **Add the negation of the theorem, R , to be proven true**

~R

Proving a theorem – 3

1 Given a set of non contradictory clauses

– assume the set of clauses is true

P
~P or Q
~ Q or R

2 **Add the negation of the theorem, R , to be proven true**

~R

– **If R is true, then the clause set now contains a contradiction**

Proving a theorem – 4

1 Given a set of non contradictory clauses
– assume the set of clauses is true

P
~P or Q
~ Q or ~R

2 Add the negation of the theorem, ~R , to be proven true

R

– Clause set now contains a contradiction

3 Find [ ] – showing that a contradiction exists,
(see the slide *Resolution Method***)**

Proving a theorem – 4

1 Given a set of non contradictory clauses
– assume the set of clauses is true

P
~P or Q
~ Q or ~R

2 Add the negation of the theorem, ~R , to be proven true

R

– Clause set now contains a contradiction

3 Find [] – showing that a contradiction exists, (see the slide *Resolution Method*)

4 Finding [ ] implies ~R is false, hence the theorem, R, is true

Resolution method problems

- In general resolution leads to longer and longer clauses
  - » Length 2 & length 2 –> length 2 **no shorter**
  - » Length 3 & length 2 –> length 3 **no shorter**
  - » In general
    length p & length q –> length p + q – 2 **longer**

Resolution method problems – 2

- ◊ In general resolution leads to longer and longer clauses
  - » Length 2 & length 2 --> length 2
  - » Length 3 & length 2 –> length 3
  - » In general length p & length q --> length p + q - 2
- ◊ **Non trivial to find the sequence of resolution rule applications needed to find [ ]**

Resolution method problems – 3

- In general resolution leads to longer and longer clauses
  - Length 2 & length 2 --> length 2 (see earlier slide) – no shorter
  - Length 3 & length 2 –> length 3 (longer)
  - In general length p & length q --> length p + q - 2 (see earlier slide)
- Non trivial to find the sequence of resolution rule applications needed to find [ ]
- **But at least there is only one rule to consider, which has helped automated theorem proving**

The Big Question

How does all this relate to Prolog?

If A then B – Propositional case

◊ Example 1: In Prolog we write

A :- B.

◊ Which in logic is

A if B ==> if B then A

==> A or ~B

Clausal form
A ← B

◊ Example 2

A :- B , C , D.

A if B and C and D

==> if B and C and D then A

==> A or ~B or ~C or ~D

Clausal form
A ← B, C, D

If A then B – Propositional case – 2

◊ Example 3

if B and C and D then P and Q and R

==> ~B or ~C or ~D or (P and Q and R)

==> (~B or ~C or ~D) or (P and Q and R)

==> ~B or ~C or ~D or P
~B or ~C or ~D or Q
~B or ~C or ~D or R

distribution

> **In Prolog**

```
P :- B , C , D.
Q :- B , C , D.
R :- B , C , D.
```

Clausal form

P ← B, C, D
Q ← B, C, D
R ← B, C, D

If A then B – Propositional case – 4

◊ Example 4

if B and C and D then P or Q or R

==> ~B or ~C or ~D or P or Q or R

Clausal form P, Q, R ← B, C, D

No single statement in Prolog for such an if ... then ... Choose one or more of the following depending upon the expected queries and database

```
P :- B , C , D , ~Q , ~R
Q :- B , C , D , ~P , ~R
R :- B , C , D , ~P , ~Q
```

If A then B – Propositional case – 5

◊ Example 5

if the_moon_is_made_of_green_cheese

then pigs_can_fly

==>

~ the_moon_is_made_of_green_cheese or pigs_can_fly

> **In Prolog**

pigs_can_fly :-
the_moon_is_made_of_green_cheese

Prolog facts – propositional case

◊ Prolog facts are just themselves.

```
a.
b.
the_moon_is_made_of_green_cheese.
pigs_can_fly.
```

◊ Comes from

if true then pigs_can_fly

==> pigs_can_fly or ~true
==> pigs_can_fly or false
==> pigs_can_fly

◊ In Prolog

pigs_can_fly :- true **:- true is implied, so it is not written**

Query

◊ A query "**A and B and C**", when negated is equivalent to

if A and B and C then false

> **insert the negation into the database, expecting to find a contradiction**

◊ Translates to

false or ~A or ~B or ~C

==> ~A or ~B or ~C

Is it true pigs_fly?

- Add the negated query to the database

  If pigs_fly then false

  ==> ~pigs_fly or false ==> ~pigs_fly

- If the database contains

  pigs_fly

- Then resolution obtains **[ ]**, the contradiction, so the negated query is false, so the query is true.

Fact or Query?

◊ Prolog distinguishes between facts and queries depending upon the mode in which it is being used. In **(re)consult** mode we are entering facts. Otherwise we are entering queries.

A longer example

1 pigs_fly :- pigs_exist , animals_can_fly.

==> pigs_fly ∨ ~pigs_exist ∨ ~animals_can_fly

2 pigs_are_pink.

==> pigs_are_pink

3 pigs_exist.

==> pigs_exist

4 birds_can_fly.

==> birds_can_fly

5 animals_can_fly.

==> animals_can_fly

Hypothesize that pigs can fly

6 :- pigs_fly.

==> ~pigs_fly

A longer example – 2

Resolve 6 & 1 ==>

7 ~pigs_exist ∨ ~animals_can_fly

Resolve 7 & 3 ==>

8 ~animals_can_fly

Resolve 8 & 5 ==>

9 [ ]

We have the empty clause – a refutation
As a consequence, the negated statement is false, the original statement, pigs_fly, is true.

Predicate Calculus

◊ **Step up to predicate calculus as resolution is not interesting at the propositional level.**

Predicate Calculus – 2

- Step up to predicate calculus as resolution is not interesting at the propositional level.
- **We add**
  - **the universal quantifier – for all x – $\forall$x**
  - **the existential quantifier – there exists an x – $\exists$ x**

Predicate Calculus – 3

- Step up to predicate calculus as resolution is not interesting at the propositional level.
- We add
  - the universal quantifier – for all x – $\forall$ x
  - the existential quantifier – there exists an x – $\exists$ x
- **But in Prolog there are no quantifiers?**
  - **They are represented in a different way**

Forall x – $\forall$ x

◊ The universal quantifier is used in expressions such as the following

$\forall$**x • P (x)**

> **For all x it is the case that P (x) is true**

$\forall$ **x • lovesBarney (x)**

> **For all x it is the case that lovesBarney (x) is true**

Forall x – ∀ x – 2

- The universal quantifier is used in expressions such as the following

  $\forall$ x • P (x)

  > For all x it is the case that P (x) is true

  $\forall$ x • lovesBarney (x)

  > For all x it is the case that lovesBarney (x) is true

- **The use of variables in Prolog takes the place of universal quantification – a variable implies universal quantification**

  P (X)

  > **For all X it is the case that P (X) is true**

  lovesBarney (X)

  > **For all x it is the case that lovesBarney (X) is true**

Exists x – ∃x

◊ **The existential quantifier is used in expressions such as the following**

  ∃ x • P(x)

  > **There exists an x such that P (x) is true**

  ∃ x • lovesBarney (x)

  > **There exists an x such that lovesBarney (x) is true**

Exists x – ∃x – 2

- The existential quantifier is used in expressions such as the following

  $\exists x \bullet P(x)$

  > There exists an x such that P(x) is true

  $\exists x \bullet lovesBarney(x)$

  > There exists an x such that lovesBarney(x) is true

- **Constants in Prolog take the place of existential quantification**

  The constant is a value of x that satisfies existence

  P (a) **a is an instance such that P (a) is true**

  lovesBarney (elliot) **elliot is an instance such that lovesBarney (elliot) is true**

Nested quantification

- ◊ $\exists x \exists y \cdot sisterOf (x , y)$
  - > There exists an x such that there exists a y such that x is the sister of y
  - > In Prolog introduce two constants

sisterOf (mary , eliza)

- ◊ $\exists x \forall y \cdot sisterOf (x , y)$
  - > There exists an x such that forall y it is the case that x is the sister of y

sisterOf (leila , Y)

  - > One constant for all values of Y

Nested quantification – 2

◊ $\forall x\ \exists y \cdot$ **sisterOf (x , y)**

> For all x there exists a y such that x is the sister of y

> The value of y depends upon which X is chosen, so Y becomes a function of X

sisterOf (X , f (X))

◊ $\forall x\ \forall y \cdot$ **sisterOf (x , y)**

> For all x and for all y it is the case that x is the sister of y

sisterOf (X , Y)

> Two independent variables

Nested quantification – 3

- ◊ $\forall x\ \forall y\ \exists z \cdot P(z)$
  - > For all x and for all y there exists a z such that P(z) is true
  - > The value of z depends upon both x and y, and so becomes a function of X and Y

  $P(g(X, Y))$

- ◊ $\forall x\ \exists y\ \forall z\ \exists w \cdot P(x, y, z, w)$
  - > For all x there exists a y such that for all z there exists a w such that P(x, y, z, w) is true
  - > The value of y depends upon x, while the value of w depends upon both x and z

  $P(X, h(X), Z, g(X, Z))$

Skolemization

- Removing quantifiers by introducing variables and constants is called skolemization
  - Named after the Norwegian mathematician Thoralf Skolem

Skolemization – 2

◊ Removing quantifiers by introducing variables and constants is called skolemization

◊ **Removal of $\exists$ gives us functions, and constants, which are functions with no arguments.**

  » **Functions in Prolog are the compound terms**

Skolemization – 3

◊ Removing quantifiers by introducing variables and constants is called skolemization

◊ Removal of $\exists$ gives us functions and constants – functions with no arguments.

  » Functions in Prolog are the compound terms

◊ **Removal of $\forall$ gives us variables**

Skolemization – 4

- Removing quantifiers by introducing variables and constants is called skolemization
- Removal of $\exists$ gives us functions and constants – functions with no arguments.
  - Functions in Prolog are called structures or compound terms
- Removal of $\forall$ gives us variables
- **Each predicate is called a literal**

Herbrand universe

◊ The transitive closure of the constants and functions is called the **Herbrand universe**

> **In general it is infinite**

Herbrand universe – 2

◊ The transitive closure of the constants and functions is called the **Herbrand universe**

> **– In general it is infinite**

◊ A Prolog database defines predicates over the Herbrand universe defined by the database

Herbrand universe – 3

◊ The transitive closure of the constants and functions is called the **Herbrand universe**

> – **In general it is infinite**

◊ A Prolog database defines predicates over the Herbrand universe defined by the database

> **The compound terms in the database determine the Herbrand universe**

Herbrand universe – Determination

◊ It is the union of all constants and the recursive application of functions to constants

Herbrand universe – Determination – 2

- It is the union of all constants and the recursive application of functions to constants
  - **Level 0 – Base level – is the set of constants**

Herbrand universe – Determination – 3

- It is the union of all constants and the recursive application of functions to constants
  - » Level 0 – Base level – is the set of constants
  - » **Level 1 constants are obtained by the substitution of level 0 constants for all the variables in the functions in all possible ways**

Herbrand universe – Determination – 4

◊ It is the union of all constants and the recursive application of functions to constants

» Level 0 – Base level – is the set of constants

» Level 1 constants are obtained by the substitution of level 0 constants for all the variables in the functions in all possible ways

» **Level 2 constants are obtained by the substitution of level 0 and level 1 constants for all the variables in the functions in all possible ways**

Herbrand universe – Determination – 5

◊ It is the union of all constants and the recursive application of functions to constants

» Level 0 – Base level – is the set of constants

» Level 1 constants are obtained by the substitution of level 0 constants for all the variables in the functions in all possible ways

» Level 2 constants are obtained by the substitution of level 0 and level 1 constants for all the variables in the functions in all possible ways

» **Level n constants are obtained by the substitution of all level 0 .. n-1 constants for all variables in the functions in all possible ways**

Back to Resolution

◊ **Predicate calculus case is similar to the propositional case in that resolution combines two clauses where two literals cancel each other**

Back to Resolution – 2

◊ Predicate calculus case is similar to the propositional case in that resolution combines two clauses where two literals cancel each other

◊ **With variables and constants we use pattern matching to find the most general unifier (binding list for variables) between two literals**

Back to Resolution – 3

◊ Predicate calculus case is similar to the propositional case in that resolution combines two clauses where two literals cancel each other

◊ With variables and constants we use pattern matching to find the most general unifier (binding list for variables) between two literals

◊ **The unifier is applied to all the literals in the two clauses being resolved**

Back to Resolution – 4

- Predicate calculus case is similar to the propositional case in that resolution combines two clauses where two literals cancel each other
- With variables and constants we use pattern matching to find the most general unifier (binding list for variables) between two literals
- The unifier is applied to all the literals in the two clauses being resolved
- **All the literals, except for the two which were unified, in both clauses are combined with "or"**

Back to Resolution – 5

- ◊ Predicate calculus case is similar to the propositional case in that resolution combines two clauses where two literals cancel each other
- ◊ With variables and constants we use pattern matching to find the most general unifier (binding list for variables) between two literals
- ◊ The unifier is applied to all the literals in the two clauses being resolved
- ◊ All the literals, except for the two which were unified, in both clauses are combined with “or”
- ◊ **The new clause is added to the set of clauses**

Back to Resolution – 6

◊ Predicate calculus case is similar to the propositional case in that resolution combines two clauses where two literals cancel each other

◊ With variables and constants we use pattern matching to find the most general unifier (binding list for variables) between two literals

◊ The unifier is applied to all the literals in the two clauses being resolved

◊ All the literals, except for the two which were unified, in both clauses are combined with "or"

◊ The new clause is added to the set of clauses

◊ **When [ ] is found, the bindings in the path back to the query give the answer to the query**

Example

◊ Given the following clauses in the database

person (bob).

~person (X) or mortal (X).

forall X • if person (X) then mortal (X)

◊ Lets make a query asking if bob is a person

◊ The query adds the following to the database

~person (bob).

◊ Resolution with the first clause is immediate with no unification required

◊ The empty clause is obtained
So ~person(bob) is false, therefore person(bob) is true

Example – 2

◊ Given the following clauses in the database

person (bob).

~person (X) or mortal (X).

forall X • if person (X) then mortal (X)

◊ Lets make a query asking if bob is mortal

◊ The query adds the following to the database

~mortal (bob).

◊ Resolution with the second clause gives with **X_1 = bob** (renaming is required!)

~person (bob).

◊ Resolution with the first clause gives [ ]
So **~mortal(bob)** is false, therefore **mortal(bob)** is true

Example – 3

◊ Given the following clauses in the database

person (bob).
~person (X) or mortal (X).

◊ Lets make a query asking does a mortal exist
The query adds the following to the database

~mortal (X).   **~ (∀ x • mortal (x)) -- negated query**

◊ Resolution with the second clause gives with **X_1 = X** (renaming is required!)

~person (X_1).

◊ Resolution with the first clause gives [] with **X_1 = bob**
So **~mortal(X)** is false, therefore **mortal(X)** is true with
bob = X_1 = **X**

Example – 4

- Given the following clauses in the database

  person (bob).
  ~person (X) or mortal (X).

- Lets make a query asking if alice is mortal

  ~mortal (alice).

- Resolution fails with the first clause but succeeds with the second clause gives with **X_1 = alice**

  ~person (alice).

- Resolution with the first clause and second clause fails, searching the database is exhausted without finding [ ]

- So **~mortal(alice)** is true, therefore **mortal(alice)** is false

Example – 4 cont'd

◊ Actually all that the previous query determined is that **~mortal(alice)** is consistent with the database and resolution was unable to obtain a contradiction

Prolog searches are based on a **closed universe**

Truth is relative to the database

Unification

- In order to use the resolution method with predicate calculus we need to be able to find the most general unifier (mgu) between two literals.
- **p(a, b, c)** and **p(X, Y, Z)**
  - » **mgu = { X / a , Y / b , Z / c }**
- **f(g(a, b), a, g(a, b)** and **f(g(X, Y, X, g(X, y))**
  - » **mgu = { X / a , Y / b , Z / a }**
- **p(a, f(b, a), c)** and **p(X, f(X, Y), Z)**
  - » **mgu does not exist**
- **p(X, a, b)** **and** **p(Y, Y, b)**
  - » **mgu = { X / Y , Y / a}**

Factoring

- General resolution permits unifying several literals at once by **factoring**
  > **unifying two literals within the same clause, if they are of the same "sign", both positive, P(...) or P(...), or both negative, ~P(...) or ~P(...)**

- Why factor?
  > **Gives shorter clauses, making it easier to find the empty clause**

Factoring – 2

◊ For example given the following clause

loves (X , bob) or loves (mary , Y)

◊ We can factor (obtain the common instances) by unifying the two loves literals

loves (mary , bob) **X = mary and Y = bob**

◊ The factored clause is implied by the un-factored clause as it represents the subset of the cases that make the un-factored clause true

> **Can be added to the database without contradiction**

Creating a database

◊ A large part of the work in creating a database is to convert general predicate calculus statements into conjunctive normal form.

◊ Much of Chapter 10 of Clocksin & Mellish describes how this can be done.

Horn clauses

◊ Clauses where the consequent is a single literal.

> **For example, X is the consequent in**

If A and B and C then X

Horn clauses – 2

- ◊ Clauses where the consequent is a single literal.
  - > **For example, X is the consequent in**

    If A and B and C then X

- ◊ Horn clauses are important because, while resolution is complete, it usually leads to getting longer and longer clauses while finding contradiction means getting the empty clause

Horn clauses – 3

◊ Clauses where the consequent is a single literal.

> **For example, X is the consequent in**

If A and B and C then X

◊ Horn clauses are important because, while resolution is complete, it usually leads to getting longer and longer clauses while finding contradiction means getting the empty clause

» **Need to get shorter clauses or at least contain the growth in clause length**

Horn clauses – 4

◊ Clauses where the consequent is a single literal.

> **For example, X is the consequent in**

If A and B and C then X

◊ Horn clauses are important because, while resolution is complete, it usually leads to getting longer and longer clauses while finding contradiction means getting the empty clause

» **Need to get shorter clauses or at least contain the growth in clause length**

» **General resolution can lead to exponential growth**

Horn clauses – 5

◊ Clauses where the consequent is a single literal.

> For example, X is the consequent in

If A and B and C then X

◊ Horn clauses are important because, while resolution is complete, it usually leads to getting longer and longer clauses while finding contradiction means getting the empty clause

» **Need to get shorter clauses or at least contain the growth in clauses**

» **General resolution can lead to exponential growth in both**

> clause length

> size of the set of clauses

Horn clauses – 6

◊ Horn clauses have the property

> **Every clause has at most one positive literal (un-negated) and zero or more negative literals**

Horn clauses – 7

◊ Horn clauses have the property

> **Every clause has at most one positive literal (un-negated) and zero or more negative literals**

person (bob).
mortal (X) ~person (X)
binTree (t (D , L , R))
~treeData (D) ~binTree (L) ~binTree (R).

Horn clauses – 8

- Horn clauses have the property

  > **Every clause has at most one positive literal (un-negated) and zero or more negative literals**

person (bob).
mortal (X) ~person (X)
binTree (t (D , L , R))
~treeData (D) ~binTree (L) ~binTree (R).

- Facts are clauses with one positive literal and no negated literals, resolving with facts reduces the length of clauses

Horn clauses – 9

- Horn clauses have the property
  - **Every clause has at most one positive literal (un-negated) and zero or more negative literals**

person (bob).
mortal (X) ~person (X)
binTree (t (D , L , R))
~treeData (D) ~binTree (L) ~binTree (R).

- Facts are clauses with one positive literal and no negated literals, resolving with facts reduces the length of clauses
- Horn clauses can represent anything we can compute

Horn clauses – 10

◊ Horn clauses have the property

> **Every clause has at most one positive literal (un-negated) and zero or more negative literals**

person (bob).
mortal (X) ~person (X)
binTree (t (D , L , R))
~treeData (D) ~binTree (L) ~binTree (R).

◊ Facts are clauses with one positive literal and no negated literals, resolving with facts reduces the length of clauses

◊ Horn clauses can represent anything we can compute

» **Any database and theorem that can be proven within first order predicate calculus can be translated into Horn clauses**