

Prolog and the Resolution Method

The Logical Basis of Prolog

Chapter 10

Background

- ◇ 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 case

- ◇ 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** \sim)
 - » set of clauses implicitly **anded** together

- ◇ Example

A or B

C or D or \sim E

F

\Rightarrow

(A or B) and (C or D or \sim E) and F

Propositional case – 2

◇ What happens if there is a contradiction in the set of clauses

◇ Example – only one clause

P

◇ Add $\sim P$ to the set of clauses

P

$\sim P$

\implies

P and $\sim P$

\implies

[] -- null the empty clause is false

◇ Think of **P** and **$\sim P$** canceling each other out of existence

Resolution rule

◇ Given the clause

Q or \sim R

◇ and the clause

R or P

◇ then resolving the two clauses is the following

(Q or \sim R) and (R or P)

\Rightarrow

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 method

- ◇ Combine clauses using resolution to find the empty clause
 - » **Implying one or more of the clauses in the set is false.**

- ◇ Given the clauses

1 P

2 $\sim P$ or Q

3 $\sim Q$ or $\sim R$

4 R

- ◇ Can resolve as follows

5 P and ($\sim P$ or Q) \implies Q resolve 1 and 2

6 Q and ($\sim Q$ or $\sim R$) \implies $\sim R$ resolve 5 and 3

7 $\sim R$ and R \implies [] resolve 6 and 4

Resolution method – 2

◇ Using resolution to prove a theorem

> **1 Given the non contradictory clauses
– assuming original set of clauses is true**

P

$\sim P$ or Q

$\sim Q$ or $\sim R$

> **2 Add the negation of the theorem, $\sim R$, to be proven true**

R

– Clause set now contains a contradiction

> **3 Find \square – showing that a contradiction exists,
(see previous slide)**

> **4 implies R is false, hence the theorem, $\sim R$, is true**

Resolution method – 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**
- ◇ 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

If A then B – Propositional case

◇ Example 1: In prolog we write

A :- B.

◇ Which in logic is

A if B \implies if B then A

\implies A or \sim B

◇ Example 2

A :- B , C , D.

A if B and C and D

\implies if B and C and D then A

\implies A or \sim B or \sim C or \sim D

If A then B – Propositional case – 2

◇ Example 2

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

$\Rightarrow \sim B$ or $\sim C$ or $\sim D$ or (P and Q and R)

$\Rightarrow (\sim B$ or $\sim C$ or $\sim D)$ or (P and Q and R)

**$\Rightarrow \sim B$ or $\sim C$ or $\sim D$ or P
 $\sim B$ or $\sim C$ or $\sim D$ or Q
 $\sim B$ or $\sim C$ or $\sim D$ or R**

> In Prolog

P :- B , C , D.

Q :- B , C , D.

R :- B , C , D.

If A then B – Propositional case – 4

◇ Example 3

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

$\Rightarrow \sim B$ or $\sim C$ or $\sim D$ or P or Q or R

> 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 , $\sim Q$, $\sim R$

Q :- B , C , D , $\sim P$, $\sim R$

R :- B , C , D , $\sim P$, $\sim Q$

If A then B – Propositional case – 5

◇ Example 4

**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

P

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 $\sim A$ or $\sim B$ or $\sim C$
 $\implies \sim A$ or $\sim B$ or $\sim C$

Is it true pigs_fly?

- ◇ Add the negated question 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.
- ◇ 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.

Predicate Calculus

- ◇ 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 – $\exists x$**

forall x – $\forall x$

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

$\forall x \cdot P(x)$

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

$\forall x \cdot \text{lovesBarney}(x)$

> 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

$\text{lovesBarney}(X)$

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

Exists x – \exists x

- ◇ The existential quantifier is used in expressions such as the following
 - $\exists x \cdot P(x)$
 - > There exists an x such that P(x) is true
 - $\exists x \cdot \text{lovesBarney}(x)$
 - > There exists an x such that lovesBarney(x) is true
 - ◇ Constants in Prolog take the place of existential quantification
 - a constant implies existential quantification
 - The constant is a value of x that satisfies existence
- $P(a)$ a is an instance such that P(a) is true
- $\text{lovesBarney}(elliot)$ elliot is an instance such that lovesBarney(elliot) is true

Nested quantification

◇ $\exists x \exists y \cdot \text{sisterOf} (x , y)$

> There exists an x such that there exists a y such that x is the sister of y

> In Prolog introduce two constants

sister (mary , eliza)

◇ $\exists x \forall y \cdot \text{sisterOf} (x , y)$

> There exists an x such that for all y it is the case that x is the sister of y

sister (leila , Y)

> One constant for all values of Y

Nested quantification – 2

◇ $\exists x \exists y \cdot \text{sisterOf} (x , y)$

> Forall 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

$\text{sisterOf} (X , f (X))$

◇ $\exists x \exists y \cdot \text{sisterOf} (x , y)$

> Forall x and forall y it is the case that x is the sister of y

$\text{sisterOf} (X , Y)$

> Two independent variables

Nested quantification – 3

$$\diamond \forall x \forall y \exists z \cdot P(z)$$

- > 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

$$P(g(X, Y))$$

$$\diamond \forall x \exists y \forall z \exists w \cdot P(x, y, z, w)$$

- > 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**
- ◇ Removal of \forall gives us functions and constants – functions with no arguments.
 - » **Functions in Prolog are called structures or compound terms**
- ◇ Removal of \exists 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
- ◇ A Prolog database defines predicates over the Herbrand universe determined by the database

Herbrand universe – Determination

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

Back to Resolution

- ◇ 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).**
 - \sim 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
 - \sim person (bob).**
- ◇ Resolution with the first clause is immediate with no unification required
- ◇ The empty clause is obtained
So \sim 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).

\sim person (X) or mortal (X).

- ◇ Lets make a query asking does a mortal exists
The query adds the following to the database

\sim mortal (X). **$\sim (\exists x \cdot \text{mortal} (x))$ -- negated query**

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

\sim person (X₁).

- ◇ Resolution with the first clause gives [] with **$X_1 = \text{bob}$**
So \sim mortal(X) is false, therefore mortal(X) is true with
X = bob

Example – 4

- ◇ Given the following clauses in the database
 - person (bob).**
 - \sim person (X) or mortal (X).**
- ◇ Lets make a query asking is alice mortal
 - \sim mortal (alice).**
- ◇ Resolution fails with the first clause but succeeds with the second clause gives with **X_1 = alice**
 - \sim person (alice).**
- ◇ Resolution with the first clause and second clause fails, searching the database is exhausted without finding []
- ◇ So \sim mortal(alice) is true, therefore mortal(alice) is false

Example – 4 cont'd

- ◇ Actually all that the previous query determined is that $\sim\text{mortal}(\text{alice})$ 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

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, $P(\dots)$ or $\neg P(\dots)$, or both negative, $\neg P(\dots)$ or $\neg P(\dots)$**
- ◇ 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 unfactored clause as it represents a subset of the cases that make the unfactored 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 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 size**
 - > **size of the set of clauses**

Horn clauses – 2

- ◇ 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**