Normal Forms

Recall: Conjunctive Normal Form (CNF) and Disjunctive Normal Form (DNF) for propositional logic

Prenex Normal Form. A FOL formula φ is said to be in *Prenex Normal Form* iff all its quantifiers (if any) 'come first'. The quantifier-free part of φ is called the *matrix* of φ.

Every sentence can be transformed into a logically equivalent sentence in Prenex Normal Form.

Transformation into Prenex Normal Form

If necessary, rewrite the formula first to ensure that no two quantifiers bind the same variable and no variable has both a free and a bound occurrence (variables need to be 'named apart')!

$$\begin{array}{rclrcl} \neg(\forall x)A & \equiv & (\exists x)\neg A & \neg(\exists x)A & \equiv & (\forall x)\neg A \\ ((\forall x)A) \wedge B & \equiv & (\forall x)(A \wedge B) & ((\exists x)A) \wedge B & \equiv & (\exists x)(A \wedge B) \\ ((\forall x)A) \vee B & \equiv & (\forall x)(A \vee B) & ((\exists x)A) \vee B & \equiv & (\exists x)(A \vee B) \end{array}$$

etc.

To avoid mistakes, formulas involving $\rightarrow$ or $\leftrightarrow$ should first be translated into formulas involving only $\neg$, $\wedge$, and $\vee$ (besides quantifiers).

Skolemisation

Skolemisation is the process of removing existential quantifiers from a given formula in Prenex Normal Form (without affecting satisfiability).

Algorithm. Given: a formula in Prenex Normal Form.

(1) If necessary, turn the formula into a sentence by adding $(\forall x)$ in front for every free variable x ('universal closure').

(2) While there are still existential quantifiers, repeat: replace
- $(\forall x_1) \cdots (\forall x_n)(\exists y)\varphi$ with
- $(\forall x_1) \cdots (\forall x_n)\varphi[y \leftarrow f(x_1, \ldots, x_n)]$, where f is a new function symbol.

Skolemisation (2)

Definition 1 (Skolem Normal Form) *A formula φ is said to be in Skolem Normal Form iff it is of the following form:*

$$\varphi \quad = \quad (\forall x_1)(\forall x_2) \cdots (\forall x_n)\, \varphi',$$

where φ' is a quantifier-free formula in CNF (with $n \in \mathbb{N}^0$).

Theorem 1 (Skolemisation) *For every formula φ there is a formula φ_{sk} in Skolem Normal Form such that φ is satisfiable iff φ_{sk} is satisfiable. φ_{sk} can be obtained from φ through the process of Skolemisation.*

Proof. [omitted]

Clauses

Resolution is a deduction system for formulas in Skolem Normal Form. Traditionally, formulas are written as sets of so-called *clauses*.

Clauses. A *clause* is a set of literals. Logically, it corresponds to the *disjunction* of these literals.

Sets of clauses. A *set of clauses* logically corresponds to the *conjunction* of the clauses in the set.

This means, any formula in Skolem Normal Form can be written as a set of clauses. *Variables are understood to be implicitly universally quantified.*

Recall: In Prolog *rules*, *facts*, and *queries* are also called clauses.

Binary Resolution with Factoring

Aim. Show $\Delta \models \varphi$ (for a set of sentences Δ and a sentence φ).

Preparation. Compute Skolem Normal Form of formulas in Δ and of $\neg\varphi$ and write them as a set of clauses (variables named apart).

Input. Set of clauses (which we want to show to be unsatisfiable).

Rules.

Binary Resolution Rule
$\dfrac{\{L_1\} \cup C_1 \quad \{L_2^{\mathsf{C}}\} \cup C_2}{\mu(C_1 \cup C_2)}$
L_2^{C} is the complement of L_2 μ is an mgu of L_1 and L_2

Factoring
$\dfrac{\{L_1, \ldots, L_n\} \cup C}{\sigma(\{L_1\} \cup C)}$
σ unifies $\{L_1, \ldots, L_n\}$

Success. We succeed if we can derive the empty clause $\Box$.

Why Factoring?

Try to derive the empty clause from the following (obviously unsatisfiable) set of clauses *without using the factoring rule.*

$$\{\{P(x), P(y)\}, \{\neg P(u), \neg P(v)\}\}$$

$\Rightarrow$ It's not possible!

This means that our *binary* resolution rule alone (without factoring) would not be a *complete* deduction system for FOL.

Remark. The *general resolution rule* allows us to resolve using sub-clauses (rather than just literals). In that case we can do without factoring.

Horn Clauses

Same old problem. We still have the same problem of a sometimes huge search space (as before with Tableaux): There is no general strategy to tell us which two clauses to resolve next.

Horn fragment. The situation can be improved if we restrict ourselves to interesting *fragments* of first-order logic. An important fragment is the so-called *Horn fragment*, namely those formulas of FOL that correspond to (sets of) *Horn clauses*:

Definition 2 (Horn clause) *An (implicitly universally quantified) clause is called a Horn clause iff it contains at most one positive literal.*

We distinguish *positive* Horn clauses (one positive literal) and *negative* Horn clauses (no positive literals).

SLD Resolution for Horn Clauses

SLD Resolution stands for **S**elective **L**inear Resolution for **D**efinite clauses, where:

- *linear* means we always use the latest resolvent in the next step,
- we have a *selection function* telling us which literal to use, and
- the input is restricted to Horn clauses, exactly one of which has to be negative (*definite clause* is another word for positive Horn clause).

SLD Resolution is *complete* for the Horn fragment [proof omitted].

Logic Programming

Programs. As we will see, a *logic program* corresponds to a set of fist-order sentences Δ.

Submitting queries. Submitting a *query* φ (which also corresponds to a sentence) to a logic programming system means asking whether φ can be inferred from the information given in the program Δ, that is, whether the following holds:

$$\Delta \models \varphi \qquad \text{(i.e. whether } \Delta \cup \{\neg\varphi\} \text{ is unsatisfiable)}$$

$\Rightarrow$ Logic Programming is an application of Automated Deduction.

Prolog. *Prolog* is a logic programming language which extends this basic idea with additional control structures (such as *negation as failure* and *cuts*). In this course, however, we are only concerned with the theoretical foundations of *'pure' Prolog*.

Logic Programming (2)

Translation. Programs in 'pure' Prolog can be translated into FOL:

- Prolog *rules* of the form `q :- p1, ..., pn.` are translated as $(\forall x_1)\cdots(\forall x_n)(P_1 \wedge \cdots \wedge P_n \rightarrow Q)$, where x_1, ..., x_n are the variables appearing in the rule.
- Prolog *facts* of the form `q.` are translated as $(\forall x_1)\cdots(\forall x_n)Q$, where x_1, ..., x_n are the variables appearing in the fact.
- Prolog *queries* of the form `?- p1, ..., pn.` are translated as $(\exists x_1)\cdots(\exists x_n)(P_1 \wedge \cdots \wedge P_n)$, where x_1, ..., x_n are the variables appearing in the query. So the *negation of a query* has the form $(\forall x_1)\cdots(\forall x_n)\neg(P_1 \wedge \cdots \wedge P_n)$.

Observation. All the above formulas correspond to Horn clauses. In fact, rules and facts correspond to positive Horn clauses and negated queries correspond to negative ones.

Logic Programming (3)

Prolog implements SLD Resolution:

- *Linearity:* we start with the only negative clause (the negated query) and then always use the previous resolvent (new query).
- The *selection function* is very simple: it always chooses the first literal (in the current 'query').
- The input is restricted to *one negative Horn clause* (negated query) and a number of *positive Horn clauses* (rules and facts).

Backtracking. In practice, one problem remains: if there is more than one way to resolve with the selected literal (i.e. more than one matching rule or fact) then we don't know which one will eventually lead to a successful refutation. In Prolog, always the first one is chosen and if this turns out not to be successful, *backtracking* is used to try another one.

Worked Example

Consider the following Prolog program:

```
parent( elisabeth, charles).
parent( charles, harry).
ancestor( X, Y) :- parent( X, Y).
ancestor( X, Z) :- parent( X, Y), ancestor( Y, Z).
```

What will happen if we submit the following query after the above program has been consulted by Prolog?

```
?- ancestor( elisabeth, harry).
```


Step 1: Translate into FOL

For the program we get the following formulas:

(1) $P(e, c)$

(2) $P(c, h)$

(3) $(\forall x)(\forall y)(P(x, y) \rightarrow A(x, y))$

(4) $(\forall x)(\forall y)(\forall z)(P(x, y) \wedge A(y, z) \rightarrow A(x, z))$

For the *negation of the query* we get:

(5) $\neg A(e, h)$

Remark. The above query is very simple. In the general case, the translation of a query is the universal closure of the negation of the conjunction of the subgoals of the query.

Step 2: Rewrite Formulas as Clauses

Formulas we get from translating a Prolog program already are in Prenex Normal Form and we don't need to Skolemise either (because there are no existential quantifiers).

We have to rewrite the implications as disjunctions. Here, we directly give the clauses (which correspond to disjunctions!). Remember that variables have to be renamed!

(1) $\{P(e, c)\}$

(2) $\{P(c, h)\}$

(3) $\{\neg P(x_1, y_1),\ A(x_1, y_1)\}$

(4) $\{\neg P(x_2, y_2),\ \neg A(y_2, z_2),\ A(x_2, z_2)\}$

(5) $\{\neg A(e, h)\}$

Step 3: Use Resolution to Derive the Empty Clause

(1)	$\{P(e, c)\}$	
(2)	$\{P(c, h)\}$	
(3)	$\{\neg P(x_1, y_1),\ A(x_1, y_1)\}$	
(4)	$\{\neg P(x_2, y_2),\ \neg A(y_2, z_2),\ A(x_2, z_2)\}$	
(5)	$\{\neg A(e, h)\}$	
(6)	$\{\neg P(e, y_3),\ \neg A(y_3, h)\}$	from (4,5) with $[x_2 \leftarrow e, z_2 \leftarrow h]$
(7)	$\{\neg A(c, h)\}$	from (1,6) with $[y_3 \leftarrow c]$
(8)	$\{\neg P(c, h)\}$	from (3,7) with $[x_1 \leftarrow c, y_1 \leftarrow h]$
(9)	$\square$	from (2,8)

Remark. If there had been variables in our query, then the substitutions made to them would have been part of the answer.