Knowledge Representation Summary

Author: **Nicolò Brandizzi**

Contributors:

SAPIENZA
UNIVERSITÀ DI ROMA

DIAG

Sapienza

November 2018

Contents

Abstract

This is **free** material! You should not spend money on it.
This notes are about the *Knowledge Representation* part taught by professor Daniele Nardi in the Artificial Intelligence class. Everyone is welcome to contribute to this notes in any relevant form, just ask for a pull request and be patient.
Remember to add your name under the contributors list in the title page when submitting some changes (if you feel like it).

1 Logic based agents

A **knowledge base** [KB] is a set of sentences in a knowledge representation language that express some assertion about the world.
We can either:

- *Tell*: i.e. add new sentences to the KB or
- *Ask*: i.e. query what is known.

We can use both this actions to do **inference** in which we derive new sentences from old ones.

There are two ways of building a KB structure:

- **Declarative**: in which the agent is *told* various aspects of the world [1] until it is capable of working in the environment.
- **Procedural**: which encodes desired behavior directly in the program.

Finally an agent can be viewed at:

- *Knowledge level*, where we specify what the agents knows and what are its goals. Or at
- *Implementation level*, where we need to specify the data structures used in the KB and the way to manipulate them.

1.1 Logic

Sentences must follow two principles in order to be considered *correct*:

- **Syntax** holds when a sentence is *well formed*, e.g. in mathematics "$x + y = 4$" is correct while "$x3y+ =$" is not.
- **Semantic** that defines the *truth* of a sentence in respect to each possible world. For example the sentence $x + y = 4$ is true in a world where $x = 2, y = 2$ but false when $x = 1, y = 4$.

Other than saying *each possible world* when referring to the KB, we can use the word **interpretation**. Whereas possible worlds might be thought of as (potentially) real environments that the agent might or might not be in, interpretations are mathematical abstractions, each of which simply fixes the truth or falsehood of every relevant sentence.

[1] That is the designer add new sentences.

Moreover a **model** m is an interpretation of a sentence α if α is true in m [2]. Given a set of the models of α, $M(\alpha)$, we can derive the concept of *entailment*:
A KB *entails* a sentence α [3] if and only if, in every interpretation in which the KB is true [4], α is also true:

$$KB \models \alpha \Leftrightarrow M(KB) \subseteq M(\alpha)$$

Note that $M(KB) \subseteq M(\alpha)$ means that KB is a *stronger assertion* than α since it *rules out* more interpretation or possible worlds [5].

Example Lets make an example to better understand this concepts.
We have the following sentences:

- α = Overwatch is better than Fortnite.
- β = Everything is better than Diablo immortal [6].

Our KB is equal to $KB = \alpha \wedge \beta$, that is the KB knows that *Overwatch is better than Fortnite* ***and*** *that Diablo immortal sucks*. We want to know:

$$KB \models \beta$$

i.e. we can derive from KB that Diablo immortal sucks.
We first need to check if $M(KB) \subseteq M(\beta)$, in other words if the KB has fewer true interpretation [7] than β. We know that the entailment holds since:

- The KB has two independent sentences, α, β, correlated by an *and* logic relationship, which result in fewer models.
- We are asking the KB for the truth of a sentence which is directly part of the KB itself; so that the set of models $M(KB)$ is a *subset* of $M(\beta)$.

Model checking To know if $KB \models \beta$ we need to prove that $M(KB) \subseteq M(\beta)$. This can be done by **model checking**, that is enumerating all the possible interpretation where KB is true and check if those are also models of β [8]. For the above example we have:

	KB	β
$\alpha \wedge \beta$	True	True
$\neg\alpha \wedge \beta$	False	True
$\alpha \wedge \neg\beta$	False	False
$\neg\alpha \wedge \neg\beta$	False	False

Table 1: Model checking for $KB \models \beta$

[2] Here the Russell-Norvig has a different concept of *model* which is equal to the above *interpretation (= each possible world = model)* . But Nardi prefer to make the distinction between the two, so we will go with the Nardi flow (*each possible world=interpretation ≠ model=true interpretation*).
[3] That is the sentence α follows logically/is derived from KB.
[4] That is in every *model* of the KB.
[5] There are less models in KB than in α.
[6] Don't you have phones?
[7] Models.
[8] This is a direct implementation of the definition of entailment.

Deduction To better understand the difference between entailment and inference we should think of the *set of all consequences* of KB as a haystack and β as a needle. Entailment is like the needle being in the haystack; inference is like finding it.

Another way of computing the knowledge entailed by a KB is by a **deduction procedure**:

$$KB \vdash_i \beta$$

Which denotes that β can be driven from KB by an **inference algorithm** i.

Sound and Completeness Given an inference algorithm i, if i derives **only entailed** sentences from KB then it is considered **sound** [9], otherwise i would make things up as it goes along (discovering of non-existing needles).

On the other hand **completeness** is also desirable: an inference algorithm is complete if it can derive any sentence that is entailed. For real haystacks, which are finite in extent, it seems obvious that a systematic examination can always decide whether the needle is in the haystack. For many knowledge bases, however, the haystack of consequences is infinite, and completeness becomes an important issue.

Difference between $\models$ and $\vdash$ Having

- $KB \models \alpha$
- $KB \vdash \alpha$

The first symbol $\models$ is called **entailment** and means that α must be true in all of KB's models, that is KB are true when α's interpretations are true [10].

The other symbol $\vdash$ is read **derives** (KB derives α) it can be joint with a sign denoting an inference rule, such as $KB \vdash_{MP} \alpha$ [11] or $KB \vdash_R \alpha$ [12] or both $KB \vdash_{R,MP} \alpha$. You can even ignore the set of inference rules and just write $KB \vdash \alpha$, in this case you are saying that *it must exist a derivation* [13] *so that α is the last element of the chain.* You can verify the entailment with the derivation if and only if the inference rules you are applying are sound and complete.

Difference between *deduction* and *inference* Broadly speaking they are the same thing.

More specifically, the *deduction* is always referred to as a syntactic derivation, while *inference* is a more generic term which means that starting from KB we can conclude α. So *deduction* is tied to the $\vdash$ symbol, while *infer* is more generic and can be used in both $\vdash, \models$

[9] Or truth preserving.
[10] We can have some models of α which are not model in KB.
[11] Modus Ponens Section 2.1.2.
[12] Resolution Section 2.2.
[13] Obtained with some inference rule.

2 Propositional Logic

Propositional logic is the simplest logic! [14]

Syntax An **atomic sentence** is made of *one* **propositional symbol** (for example S), which can be either *True or False*.
True and False are propositional symbols that are always True/False.
Complex sentences are propositional symbols joint together by the following **logical connective** (Figure 1) :

- $\neg$ (not): $\neg S$ is the **negation** of S.
- $\wedge$ (and): $S_1 \wedge S_2$ is a **conjunction**.
- $\vee$ (and): $S_1 \vee S_2$ is a **disjunction**.
- $\Rightarrow$ (implies): $S_1 \Rightarrow S_2$ is an **implication**, where S_1 is the *antecedent/premise* and S_2 is the *consequent/conclusion*. Implication are known as **if-then** statement [15].
- $\Leftrightarrow$ (if and only if): $S_1 \Leftrightarrow S_2$ is a **biconditional**.

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
 & \mid & \neg\ \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \wedge \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \vee \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \Rightarrow \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \Leftrightarrow \mathit{Sentence} \\
 & & \\
\text{OPERATOR PRECEDENCE} & : & \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 1: Syntax summary for propositional logic

Semantic The semantic defines the truth of a sentence in respect to a particular interpretation, that assign a truth value to every propositional symbol (Figure 2).

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Figure 2: Truth Table for propositional logic.

For what regard the *implies* symbol ($P \Rightarrow Q$) we need to specify that:

[14] Not to understand...
[15] If premise then conclusion.

- It *does not* need to have any causal/relevance link between the antecedent and the consequent. For example *'5 id odd implies Tokyo is the capital of Japan'* is syntactically correct.
- Any implication is true whenever the antecedent is false. This because we are saying *'If P is true, then I am claiming that Q is true. Otherwise I am making no claim'.*
- The only way for $P \Rightarrow Q$ to be false is if P is true and Q is false.

Inference Our goal is to prove that

$$KB \models \alpha$$

With the model checking approach we just need to enumerate all the possible interpretation and check that, when there is a model of KB then there is also a model of α. This is done by assigning either *true* or *false* to every propositional symbol in any interpretation. But given n symbols there are 2^n interpretations, thus the time complexity is $O(2^n)$, while the space complexity is $O(n)$ since we're using a depth-first approach.

2.1 Theorem Proving

We can use a technique known as *theorem proving* that consist in applying rules of inference directly to the sentences in our KB to construct a proof of the desired sentence without consulting models. If the number of models is large but the length of the proof is short, then theorem proving can be more efficient than model checking. We first need to introduce some concepts.

Logical equivalence Two sentences α and β are logically equivalent if they are true in the same set of interpretations, i.e. they have the **same set of models**. We write this as $\alpha \equiv \beta$. We can formalize this propriety by writing:

$$\alpha \equiv \beta \iff \alpha \models \beta \land \beta \models \alpha$$

Following there are some standard logic equivalences (Figure 3):

$$\begin{aligned}
(\alpha \land \beta) &\equiv (\beta \land \alpha) \quad \text{commutativity of } \land \\
(\alpha \lor \beta) &\equiv (\beta \lor \alpha) \quad \text{commutativity of } \lor \\
((\alpha \land \beta) \land \gamma) &\equiv (\alpha \land (\beta \land \gamma)) \quad \text{associativity of } \land \\
((\alpha \lor \beta) \lor \gamma) &\equiv (\alpha \lor (\beta \lor \gamma)) \quad \text{associativity of } \lor \\
\neg(\neg\alpha) &\equiv \alpha \quad \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\beta \Rightarrow \neg\alpha) \quad \text{contraposition} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\alpha \lor \beta) \quad \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha)) \quad \text{biconditional elimination} \\
\neg(\alpha \land \beta) &\equiv (\neg\alpha \lor \neg\beta) \quad \text{De Morgan} \\
\neg(\alpha \lor \beta) &\equiv (\neg\alpha \land \neg\beta) \quad \text{De Morgan} \\
(\alpha \land (\beta \lor \gamma)) &\equiv ((\alpha \land \beta) \lor (\alpha \land \gamma)) \quad \text{distributivity of } \land \text{ over } \lor \\
(\alpha \lor (\beta \land \gamma)) &\equiv ((\alpha \lor \beta) \land (\alpha \lor \gamma)) \quad \text{distributivity of } \lor \text{ over } \land
\end{aligned}$$

Figure 3: Standard logic equivalences.

Validity A sentence is *valid* if it is true in all interpretations [16]. For example: $A \lor \neg A$, $True$, $A \Rightarrow A$...
Validity can be tied to the deduction problem in the following way:
For every sentences α and β, $\alpha \models \beta$ if and only if the sentence $\alpha \Rightarrow \beta$ is valid.

[16] Also known as tautologies.

Satisfiability A sentence is **satisfiable** [17] if it is true in *some* interpretation, i.e. it has some models. A sentence can be proved to be satisfiable by enumerating the possible implementations until a model is found [18]. We can say that:

- α *is valid iff* $\neg\alpha$ *is unsatisfiable*, or
- α *is satisfiable iff* $\neg\alpha$ *is not valid.*

Hence we can give another interpretation for entailment:

$$\alpha \models \beta \iff (\alpha \wedge \neg\beta) \quad is \quad unsatisfiable$$

Which is also known as the **reductio ad absurdum** (proof by contradiction).

2.1.1 Deduction in Propositional Logic

We can use *inference rules* to derive a proof [19] of the interpretation truthfulness. The idea behind finding a proof rather than using model checking is that the proof can ignore irrelevant proposition, no matter how many of them there are.
Usually this kind of rules are written in the form:

$$\frac{premisies}{conclusions} = \frac{A_1, A_2, ..., A_n}{A}$$

Suppose we want to derive some formula [20] α from the KB[21] ($KB \vdash \alpha$), there must be a sequence of formulas $\alpha_1, ..., \alpha_n$ such that:

- For every i between $1...n$ either:
  - $\alpha_i \in KB$, that is the formula α_i is in the KB. Or
  - α_i is a *direct derivation* of $\alpha_{i-k},\ k \in [1, i-1]$ [22]
- $\alpha = \alpha_n$, the chain of direct derivations brings to the formula we want to infer.

Hence we can say that $\alpha_1, ..., \alpha_n$ is a proof of α from the KB. Finding a proof for a formula can implemented as a search where:

- *Initial State*: is the KB .
- *Operators*: are *inference rules* (mentioned earlier).
- *Final state*: is the formula to be proven.

Basic proprieties We need to describe the proprieties of a inferring method $\mathcal{R}$ given a set of formulas KB and a formula A. It is important to understand that by writing $\models A$ we are referring to the truthfulness of A in all its interpretation, thus referring to the *validity* of A.

[17] This problem is called SAT and it has been shown to be NP-complete.
[18] Model checking technique.
[19] A proof is a chain of conclusion which leads to a desired goal.
[20] Or sentence.
[21] Which is a set of formulas.
[22] α_i is a direct derivation of the previous formulas. In general α_i can depend on any subset of previous α, it depends on the resolution rule used.

- First we need to prove that an infer rule $\mathcal{R}$ is **sound**, to do so we need to prove that A is **valid** given the fact that it can be derived with $\vdash_{\mathcal{R}}$.
  First thing is to notice that there is no KB before the symbol since we want to prove A being valid regardless of the existence of any KB [23].
  But if we can derive A with a deduction $\vdash$ then A must be valid, so $\mathcal{R}$ is **sound** and we have: $KB \vdash_{\mathcal{R}} A$ implies $KB \models A$.

- On the other hand, if A is valid then it exist a derivation $\vdash_{\mathcal{R}}$ than let me derive it, so that $KB \models A$ implies $KB \vdash_{\mathcal{R}} A$, thus $\mathcal{R}$ is **complete**.

2.1.2 Inference and proof

First thing first the truth tables we cited before are not associated with the inference rules. They are associated with the formulas! So you cannot apply any truth table to Modus Ponens, And Elimination and Resolution since it does not make sense.

Moreover atom and literal are the two basic elements for the construction of the formulae, in propositional logic A is a propositional variable. But since we can have bot A and $\neg A$ we say that A is a literal which can be positive or negative.

The syntax (Section 2) allows you to build a formula regardless of the amount of free variable [24], when you have a formula with no free variables then its called a **sentence**. For example if we say:

$$A \wedge B \Rightarrow C$$

then its a formula, but if we associate a meaning to each literals:

$$Student \wedge SteamSales \Rightarrow Happy$$

Then it becomes a sentence.

On the other hand we used atom to indicate a formula for the estimations of predicates that has a predicates and some arguments. While in the propositional logic we have the propositional variable as a base element and an atom is a literal, in First Order Logic we have atoms.

Modus Ponens We can ether use Modus Ponens:

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

Which given $\alpha \Rightarrow \beta$ and α it can infer β. For example

$$\frac{man \Rightarrow mortal, \quad man}{mortal}$$

or

$$\Gamma = \{feline \Rightarrow animal, cat \Rightarrow feline, cat\}$$

Result in

$$\Gamma \vdash_{MP} animal$$

Which mean that *animal* can be derived from Γ using the inference algorithm MP (Modus Ponens).

[23] That is a formula which is true in all models, for example $A \vee \neg A$.
[24] This is use in general cases.

And Elimination On the other hand we can use and elimination:

$$\frac{\alpha \wedge \beta}{\alpha}$$

more in general:

$$\frac{\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n}{\alpha_i}$$

Which works for any chain of conjunction and means that *"if the rule $A = \{\alpha_1 \wedge \alpha_2 \wedge ... \wedge \alpha_n\}$ is true, then for any $\alpha_i \in A$, α_i must be true as well"*, since we have an *and* logical chain.

Monotonicity Finally **monotonicity** is the propriety of logical system which says that the set of entailed sentences can only increase as information is added to the knowledge base:

$$if\ KB \models \alpha\ then\ KB \wedge \beta \models \alpha$$

Which means that inference rules can be applied whenever suitable premises are found in the KB; conclusion of the rule must follow regardless of what else is in the knowledge base [25].

Inference rules We can use any of the equivalences from Figure 3 as inference rules, for example:

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad and \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}$$

2.2 Resolution

So far we did not provide any algorithm which can be considered **complete**, since the lack of some *inference rules* may prevent the algorithm from reaching the goal. So we introduce the inference rule called **resolution** that yields a complete inference algorithm when coupled with any complete search algorithm (Section 2.4).

Definitions Following there are some definitions:

- *Formula*: is a set of literals joint by some logic connectives (e.g. a complex sentence).
- *Literals*: can be either one propositional symbol (or atom) or a negated atom. They are the same as an atomic formulae that is a formula that contains no logical connectives.
- *Clause*, is a *disjunction* of literals, for example $L_1 \vee L_2 \vee ... \vee L_n$
- Moreover we introduce the constants $\bot$ and $\top$, that are *False* and *True* respectively.

2.2.1 Conjunctive Normal Form [CNF]

First thing first note that if we have a KB on which we *can* use Modus Ponens *then* we can use Resolution too. On the other hand if we can use Resolution then we may be not able to use Modus Ponent, since generalize Modus Ponens can be only used for definite horn clauses.

[25] Nonmonotonic logics, which violate the monotonicity property, capture a common property of human reasoning: changing one's mind.

Definition The key idea is that: *every sentence of propositional logic is logically equivalent to a conjunction of clauses* ($Formula \equiv CNF(Formula)$) ; hence sentences expressed as a conjunction of clauses are said to be in CNF. You can either preserve equivalence when converting to CNF, but the number of clauses will be 2^n, where n is the number of literals; or you can preserve *satisfiability* introducing new literals and linearly increase the size of the formula.

Example Let's make an example using:

$$A \Leftrightarrow (B \vee C)$$

In the following steps we make use of the formulae in Figure 3

1. Use biconditional elimination and get:
   $(A \Rightarrow (B \vee C)) \wedge ((B \vee C) \Rightarrow A)$
2. Use implication elimination end get :
   $(\neg A \vee B \vee C) \wedge (\neg(B \vee C) \vee A)$
3. CFN requires the negation to be applied only to literals [26], hence we need to use the De Morgan formula to move $\neg$ inwards:
   $(\neg A \vee B \vee C) \wedge ((\neg B \wedge \neg C) \vee A)$
4. Finally we use the distributive law of $\vee$ over $\wedge$ and get:
   $(\neg A \vee B \vee C) \wedge (\neg B \vee A) \wedge (\neg C \vee A)$

Alpha-Beta Formulas There are some cases in which we need to separate dis/conjunction depending on the following formulas:

C=$\{\alpha\}$	C$_1 = \{\alpha_1\}, C_2 = \{\alpha_2\}$	C=$\{\beta\}$	C=$\{\beta_1, \beta_2\}$
$\alpha = A \wedge B$	$\alpha_1 = A,\ \alpha_2 = B$	$\beta = A \vee B$	$\beta_1 = A,\ \beta_2 = B$
$\alpha = \neg(A \vee B)$	$\alpha_1 = \neg A,\ \alpha_2 = \neg B$	$\beta = A \Rightarrow B$	$\beta_1 = \neg A,\ \beta_2 = B$
$\alpha = \neg(A \Rightarrow B)$	$\alpha_1 = A,\ \alpha_2 = \neg B$	$\beta = \neg(A \wedge B)$	$\beta_1 = \neg A,\ \beta_2 = \neg B$

Table 2: Alpha-Beta formulas for CNF

As you can see from Table 2 the alpha formulas result into two clauses [27] C_1, C_2, while the beta formulas terminate in a single clause C with two formulas [28].

Algorithm Given a formula F the algorithm for having the latter in CNF is the following:

1. Let $I = \{F\}$[29] be the initial state.
2. At the step $n + 1$ we have that $I = \{D_1, ..., D_n\}$ where D_i is a disjunction containing some formulas $\{A^i_1, ..., A^i_k\}$.
3. If A^i_j is not a literal then choose D_i and do the following $\forall X_j \in D_i$:

[26] When a sentence has negation symbols applied directly to literal then it is in **Negation normal form** [NNF]. For example $\neg(A \vee B)$ is not in NNF while $\neg A \wedge \neg B$ is.
[27] That is because there is a conjunction symbol.
[28] Note that I said formulas and not literals since both $\alpha_1, \alpha_2, \beta_1, \beta_2$ can be further simplified if they are not literals.
[29] The braces indicates that the I is a set of clauses which will be joint by conjunctions.

- if X_j is $\neg\top$ then replace it with $\bot$.
- if X_j is $\neg\bot$ then replace it with $\top$.
- if X_j is $\neg\neg X_j$ then replace it with X_j.
- if X_j is β formula then replace it with β_1, β_2.
- if X_j is *alpha* formula then replace D_i with two clauses D_i^1, D_i^2 and add them to I.

4. Continue until every D_i is made of literals.

Expansion rules Finally note that the above rules can be written in the form of *expansion rules* [30]:

$$\frac{\neg\neg A}{A} \quad \frac{\neg\top}{\bot} \quad \frac{\neg\bot}{\top} \quad \frac{\beta}{\beta_1, \beta_2} \quad \frac{\alpha}{\alpha_1|\alpha_2}$$

Another Example Given the formula:

$$(P \Rightarrow (Q \Rightarrow (S \vee T))) \Rightarrow (T \Rightarrow Q)$$

We have that:

1. Use implication elimination (from now on it will be taken for granted) and get:

$$\neg(P \Rightarrow (Q \Rightarrow (S \vee T))) \vee (T \Rightarrow Q)$$

2. Use β rule on disjunction:

$$C = \{\beta_1 = \neg(P \Rightarrow (Q \Rightarrow (S \vee T))), \beta_2 = (T \Rightarrow Q)\}$$

3. Use α rule on β_1 (negation of implication) and get:

$$C_1 = \{P, (T \Rightarrow Q)\}, \quad C_2 = \{\neg(Q \Rightarrow (S \vee T)), (T \Rightarrow Q)\}$$

4. Use β rule on C_1:

$$C_1 = \{P, \neg T, Q\}, \quad C_2 = \{\neg(Q \Rightarrow (S \vee T)), (T \Rightarrow Q)\}$$

5. Use β rule on C_2:

$$C_1 = \{P, \neg T, Q\}, \quad C_2 = \{\neg(Q \Rightarrow (S \vee T)), \neg T, Q\}$$

6. Use α rule on C_2:

$$C_1 = \{P, \neg T, Q\}, \quad C_2 = \{\neg Q, \neg T, Q\} \quad C_3 = \{\neg(S \vee T), \neg T, Q\}$$

7. Use α rule on C_3:

$$C_1 = \{P, \neg T, Q\}, \quad C_2 = \{\neg Q, \neg T, Q\} \quad C_3 = \{\neg S, \neg T, Q\} \quad C_4 = \{\neg T, \neg T, Q\}$$

[30] Using the form introduced in Section 2.1.1.

2.2.2 Horn clause

Definitions The definition for understanding Horn clauses are both reported in Figure 4 as well as in the following list:

- *Definite clause*: is a disjunction of literals of which *exactly one is positive*. For example:

$$(\neg L_1 \vee \neg L_2 \vee L_3)$$

- *Goal clause*: is a disjunction of literals of which *none is positive*. For example:

$$(\neg L_4 \vee \neg L_5 \vee \neg L_6)$$

- *Horn clause*: is a disjunction of literals of which *at most one is positive*. So you can either have *definite clauses* or *goal clauses*. For example:

$$(\neg L_1 \vee \neg L_2 \vee L_3) \quad or \quad (\neg L_4 \vee \neg L_5 \vee \neg L_6)$$

- *Horn Form*: a KB is in Horn form if it is a conjunction of Horn clauses. For example:
  1. $C \wedge (B \Rightarrow A) \wedge (C \wedge D \Rightarrow B)$
  2. $C \wedge (\neg B \vee A) \wedge (\neg(C \wedge D) \vee B)$
  3. $C \wedge (\neg B \vee A) \wedge (\neg C \vee \neg D \vee B)$

Motivation So why are we interested in definite/goal/Horne clauses?

- When we have a definite clause we can write it as an implication of conjunctions. For example:

$$(\neg L_1 \vee \neg L_2 \vee L_3) \quad becomes \quad (L_1 \wedge L_2) \Rightarrow L_3$$

  In this case the premise is called the *body* and the conclusion is called *head*. Moreover a single positive literal is a *fact* [31].

- what is the advantage of having a goal clause??????????
- Moreover inference in horn clauses can be done with the forward/backward chaining algorithm (Section 2.3).
- Finally deciding entailment with Horn clauses can be done in time that is linear in the size of the knowledge base.

$$\begin{array}{rcl}
\mathit{CNFSentence} & \rightarrow & \mathit{Clause}_1 \wedge \cdots \wedge \mathit{Clause}_n \\
\mathit{Clause} & \rightarrow & \mathit{Literal}_1 \vee \cdots \vee \mathit{Literal}_m \\
\mathit{Literal} & \rightarrow & \mathit{Symbol} \mid \neg \mathit{Symbol} \\
\mathit{Symbol} & \rightarrow & P \mid Q \mid R \mid \ldots \\
\mathit{HornClauseForm} & \rightarrow & \mathit{DefiniteClauseForm} \mid \mathit{GoalClauseForm} \\
\mathit{DefiniteClauseForm} & \rightarrow & (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{Symbol} \\
\mathit{GoalClauseForm} & \rightarrow & (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{False}
\end{array}$$

Figure 4: Different types of clauses

[31] Since a single literal can be viewed as a disjunction of one literal.

2.2.3 Propositional Resolution

The key idea of the propositional resolution is that if I have two clauses C_1, C_2 [32] and one literal L that appears with different sign in both $L \in C_1$, $\neg L \in C_2$, then I can generate a new clause by joining $C_1 \vee C_2$ and removing $L, \neg L$ from them. That is because if I have a clause where a symbol appears with both sign, e.g. $C_3 = L_1 \vee L_2 \vee \neg L_1$, then every interpretation I can give to L_1 will be indifferent since it will result in a model.

English Example Consider the following KB, with two clauses, trying to understand why can't I play Overwatch on Monday:

- C_1=*"There is an exam the next day"* or *"My pc is broken"* or *"I'm tired"*
- C_2=*"There is no exam the next day"*

This is a special case of resolution called **unit resolution** in which I have a clause C_1 and a fact C_2. So, by examining the two clauses, I can say that *There is no exam the next day*, so the alternatives are either that *my pc is broken* or that *I'm tired*.
If we add another fact to the KB:

- C_3= *I'm never tired on Mondays!*

Then the only logical conclusion why I can't play Overwatch on Monday is because *my pc is broken.* [33].

Formalization We can formalize the previous example for any given set of clauses:
Let there be two clauses:

$$L = L_1 \vee L_2 \vee ... \vee L_n = \{L_1, ..., L_n\} \tag{1}$$

$$P = P_1 \vee P_2 \vee ... \vee P_m = \{P_1, ..., P_m\} \tag{2}$$

for any n, m.
Let there be a set of literals, $O = \{O_1, ..., O_k\}$, which appears in both L and P but with opposite sign:

$$\forall O_i,\ O_i \in L, \neg O_i \in P$$

Se we can use resolution to join P and L [34] and remove O from the union:

$$\frac{L \quad P}{(L \cup P) \setminus O}$$

Which result will look like:

$$(L_1 \vee L_2 \vee ... \vee L_n \vee P_1 \vee P_2 \vee ... \vee P_m) \setminus (O_1 \vee ... \vee O_k)$$

Formal Example Let's use the following KB:

$$KB = \{C_1 = \{\neg P, Q, \neg P\}, \quad C_2 = \{P, \neg L\}\}$$

We first apply **factoring** to remove the double $\neg P$ in C_1, thus having:

$$KB = \{C_1 = \{\neg P, Q\}, \quad C_2 = \{P, \neg L\}\}$$

[32] You should know by now that a clause is a disjunction of literals.
[33] Sad me.
[34] In this instance by *joining* we mean to merge together the two set of disjunction with a disjunction symbol

Given a formula $\alpha = \{Q, \neg L\}$ we want to know if $KB \models \alpha$, that is $(KB \vee \neg\alpha)$ is unsatisfiable, that is $(KB \vee \neg\alpha) \vdash_{\mathcal{R}} \{\}$ [35]. We have that:

$$(KB \vee \neg\alpha) = \{C_1 = \{\neg P, Q\}, \quad C_2 = \{P, \neg L\}, \quad C_3 = \{\neg Q\}, \quad C_4 = \{\neg\neg L = L\}\}$$

The procedure works as follow:

1. Resolve C_1 with C_2 which outputs:

$$C_5 = \frac{\{\neg P, Q\} \quad \{P, \neg L\}}{\{\neg L, Q\}} = \{\neg L, Q\}$$

   So that we have:

$$C_3 = \{\neg Q\}, \quad C_4 = \{L\}, \quad C_5 = \{\neg L, Q\}$$

2. Resolve C_3 and C_5 in the same way and get:

$$C_4 = \{L\}, \quad C_6 = \{\neg L\}$$

3. Finally resolve C_4 and C_6 and get the empty clause $\{\}$

We just demonstrated that $(KB \vee \neg\alpha) \vdash_{\mathcal{R}} \{\}$ thus that $KB \models \alpha$. Notice that these passages can be written in a tree form as shown in Figure 5.

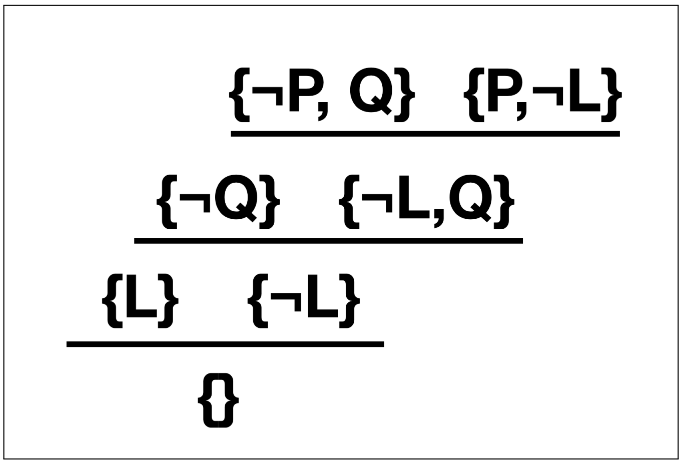

Figure 5: Resolution steps shown in tree form

Satisfiability of Resolution We start by having two clauses $C_1 = \{L_1, ..., L_n\}$, $C_2 = \{P_1, ..., P_m\}$ and a literal that is the same in each clause but with a different sign $L_i = \neg P_j = K$, i.e. if L_i is *True* then P_j is *False*. We want to prove that $\{C_1 \cup C_2\} \setminus \{K\}$ is satisfiable.
Suppose C_1 and C_2 are satisfiable, i.e. there exists a model $\mathcal{M}$; such that $\mathcal{M} \models C_1$ and $\mathcal{M} \models C_2$, and that L_i is true in $\mathcal{M}$; it follows that $\neg L_1$ is *False* in $\mathcal{M}$, thus P_j is *True* in C_2 and therefore C_2 must be *True* in $\mathcal{M}$. Consequently, $C_1 \cup C_2$ is true in $\mathcal{M}$. Since $\mathcal{M}$ is a model of C_1, C_2 by hypothesis, $C_1 \cup C_2$ is true in M.
Same goes with $\mathcal{M} \models \neg L_i$.

[35] Note that $\{\}$ is the empty clause which is equivalent to *False*.

Completeness of resolution We introduce the notion of **resolution closure** $RC(S)$ of a set of clauses S that is the set of all clauses derivable by repeated application of the resolution rule to clauses in S or their derivatives.
It is easy to understand that $RC(S)$ is finite because there are only finitely many distinct clauses that can be constructed out of the symbols $P_1, ..., P_k$ that appear in S [36].

Now let us consider the **ground resolution theorem** which states: *If a set of clauses is unsatisfiable, then the resolution closure of those clauses contains the empty clause*; that is :

$$S \quad unsatisfiable \quad iff \quad \{\} \in RC(S)$$

Now let's prove this the other way around, we want to prove that:

$$S \quad satisfiable \quad iff \quad \{\} \notin RC(S)$$

First we build a model of $RC(S)$ with the following procedure.
For i in $range(k)$:

- Given a clause $C_k \in S$ that contains $\neg P_i \in C_k$ and all its other literals are set to *False*: $C_k = false \lor ... \lor false \lor \neg P_i$; assign $False$ to P_i, so to make C_k true.
- Otherwise assign *True* to P_i.

So now we have a model of S. Since we must prove that $RC(S)$ is satisfiable, thus have some models, let's assume that what we just got is not a model, then we must have been wrong at some iteration i. That is setting the symbol P_i causes some clause C to becomes false, since we *do not have a model* all the other clauses must be already false. So C can be one of the following possibilities:

- $C = false \lor ... \lor false \lor P_i$
- $C = false \lor ... \lor false \lor \neg P_i$

If *only one* of the two is in $RC(S)$ then the algorithm would assign the right value to make C true. So the only way to have C false is that *both* possibilities are in $RC(S)$, but this is impossible since $RC(S)$ is **closed under resolution**, that is the two possibilities would have been resolved by the algorithm.

2.3 Chaining

2.3.1 Forward Chaining

Definition Determines if a single proposition symbol q [37] is entailed by a KB of definite clauses [38]. It begins from known facts in the knowledge base [39]. If all the premises of an implication are known, then its conclusion is added to the set of known facts. This process continues until the query q is reached or until no further inferences can be made. The main point to remember is that it runs in linear time.

[36] This is true only if we are using the factoring step to remove duplicate literals.
[37] What is asked to the KB, i.e. the query.
[38] Definite clauses means the KB contains either facts (positive literals) or implication with an *atomic* conclusion and, for premise, either an atom or a conjunction of literals.
[39] For this reason it is called **data driven**.

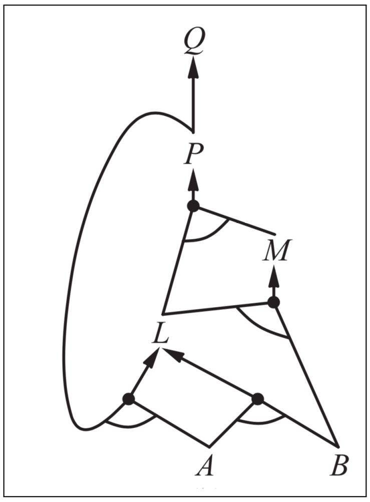

Figure 6: AND-OR graph for definite clause KB

Example Given the following KB:

- $P \Rightarrow Q$, equal to $\neg P \lor Q$ (definite clause)
- $L \land M \Rightarrow P$, equal to $\neg L \lor \neg M \lor P$ (definite clause)
- $B \land L \Rightarrow M$, equal to $\neg B \lor \neg L \lor M$ (definite clause)
- $A \land P \Rightarrow L$, equal to $\neg A \lor \neg P \lor L$ (definite clause)
- $A \land B \Rightarrow L$, equal to $\neg A \lor \neg B \lor L$ (definite clause)
- A, (known fact, since positive literal)
- B, (known fact, since positive literal)

We query Q [40].
The KB can bee seen as an **AND-OR Graph** where the implications are OR arches while the $\land$ are AND arches (Figure 6).

The flow of the algorithm works as illustrated in Figure 7.

[40] That is we ask the KB about the truthfulness of Q.

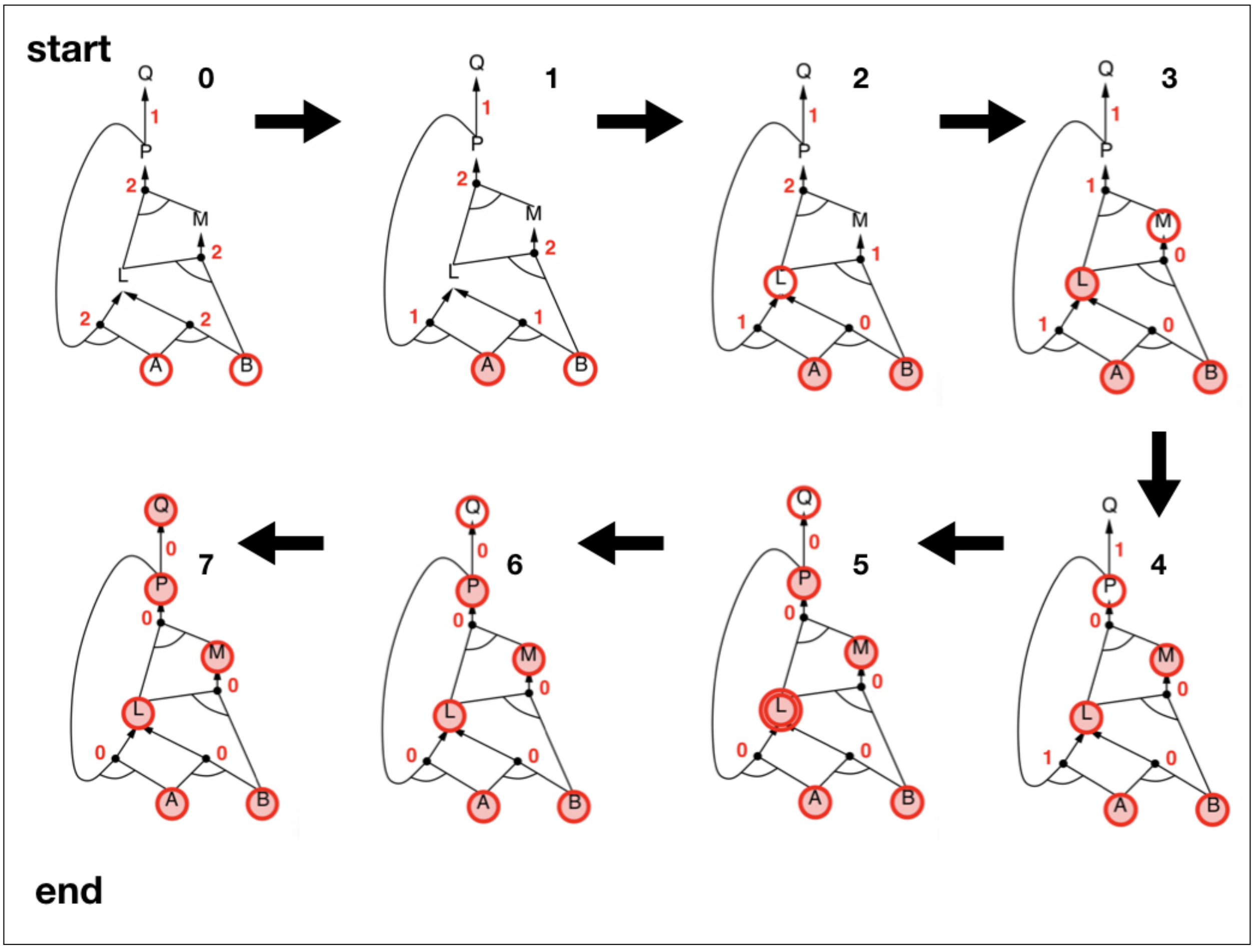

Figure 7: Forward Chaining for KB graph

Proof We can affirm that the algorithm is **sound** since every step is an application of Modus Ponens.
For the **completeness** we need to prove that every possible atomic sentence can be derived from the KB. We first introduce the notion of **fixed point** which is a state of the algorithm in which no more inference can be done; then we consider to be in the final state m that is a model [41] because we cannot apply Modus Ponens to any other clause.
Since the final state is in Horn form [42] we have a conjunction of disjunctions, so to be a model [43] every clause must be *True*. To prove this let's consider the opposite:

- Let's take a generic clause $C = a_1 \wedge ... \wedge a_n \Rightarrow b$ in m. This clause have a general number of conjunctions n in its premise.
- Suppose that C is false in m.
- For an implication to be *False* we need the premise $a_1 \wedge ... \wedge a_n$ to be *True* and the conclusion b to be *False*.
- But if that is the case I would apply Modus Ponens again $\frac{a_1 \wedge ... \wedge a_n \Rightarrow b \quad a_1 \wedge ... \wedge a_n}{b}$ and conclude b True, so in fact I'm not in a fixed point and this contradicts the assumption.

[41] In which there has been various assignment of *True/False* for the literals.
[42] As the whole KB.
[43] True interpretation.

We can conclude, therefore, that the set of atomic sentences inferred at the fixed point defines a model of the original KB. Furthermore, any atomic sentence q that is entailed by the KB must be true in all its models and in this model in particular. Hence, every entailed atomic sentence q must be inferred by the algorithm.

2.3.2 Backward Chaining

On the other hand the backward chaining works its way from the query q [44] and find those implication which conclusions are equal to q. If all the premises of a given implication which resolve in q are true then we can conclude that q is true itself.
This kind of reasoning is called **goal directed** reasoning and it usually works in linear size in respect to the size of the KB.

2.4 Proposal for model checking

In this section, we describe two families of efficient algorithms for general propositional inference based on model checking: One approach based on backtracking search, and one on local hill-climbing search. Notice that these algorithms are used for checking the **satisfiability** (SAT) of a problem.

2.4.1 DPPL

Also known as David-Putnam, Logemann, Loveland algorithm, it takes as input a sentence in conjunctive normal form (Section 2.2.1) and uses a recursive depth-first enumeration of all possible interpretations. To speed up the algorithm we can introduce some improvements.

Early Termination A clause is true if *any* literal is true [45], hence if we encounter a true literal in a clause we can stop looking at it and give it the value *true* [46].
For example having $(A \lor B) \land (A \lor C)$, if A is true then we do not need to look for the values of B, C.

Pure Symbol is a symbol which always appears with the same "sign", i.e. negated or not, so it can be assigned a value once for all the clauses. For example:

$$(A \lor \neg B) \land (\neg B \lor \neg C) \land (C \lor A)$$

In this case the literal A is a pure positive symbol, B is a pure negative while C is impure.

Unit clause is a clause with only one literal. This can be either because

- We have a clause with just one literal (fact). Or
- We have a clause with multiple literals that are false [47] except for this last one [48].

Component analysis We can divide clauses into independent subset when they *do not share* any common literal. By dividing them we can parallelize the job as well as prune large part of the state space [49]

[44] Note that if the query is true the algorithm stops immediately.
[45] Remember that a clause is a disjunction of literals of the form $C_1 \lor C_2 \lor ... \lor C_n$.
[46] Similarly a sentence (conjunction of clauses) is false if *any* clause is false.
[47] Since clauses are disjunction, false literals can be removed if there are some other that can assume the value true.
[48] This is also called **unit propagation**.
[49] No necessity to check the constraint of a literal in other clauses which do not have it.

Variable and Value ordering A general rule is to always try the value *true* before *false*. While the **degree heuristic** suggests choosing the variable that appears most frequently over all remaining clauses [50].

Intelligent Backtracking Also discussed in the Planning part of the course, intelligent backtracking keeps tracks of conflicts [51] so to cut off useless searches steps.

Random restarts When the algorithm gets stuck for a long time execute a restart from the root point.

Clever indexing at implementation level.

2.4.2 Local Search Algorithm

These algorithms works by flipping the truth value of one symbol at a time. The search space usually contains many local minima, to escape from which various forms of randomness are required.

GSAT Missing

WALKSAT The algorithm picks an unsatisfied clause and picks a symbol in the clause to flip. It chooses randomly between two ways to pick which symbol to flip:

- A *min-conflicts* step that minimizes the number of unsatisfied clauses in the new state.
- A *random walk* step that picks the symbol randomly.

This type of algorithm is very similar to simulate annealing studied in the Local search part. WALKSAT may end with the following outcomes:

- A model; thus the input sentence is satisfiable.
- A failure; then there are two possibilities:
  - Either the sentence is unsatisfiable, or
  - The algorithm needs more time to complete the search [52].

2.4.3 Random SAT problem

Depending on the number of clauses m and symbols n we can either have an *under-constraint* problem ($n > m$) or a *over-constraint* problem ($m > n$) [53]. Given the ration $r = m/n$, when the ratio approaches zero then the probability of the problem to be satisfiable approaches one and vice versa. As you can see from Figure 8 around the value $r = m/n = 4.3$ the probability for the sentence to be satisfiable approaches zero.

[50] Everything we saw in the planning part of the course.
[51] Literals that may create a conflict with other literals.
[52] Note that if there is no limit to the number of flips and the sentence is unsatisfiable then the algorithm may never end.
[53] Giving that each symbol may not appear twice in a clause and no clauses may appear twice in the sentence.

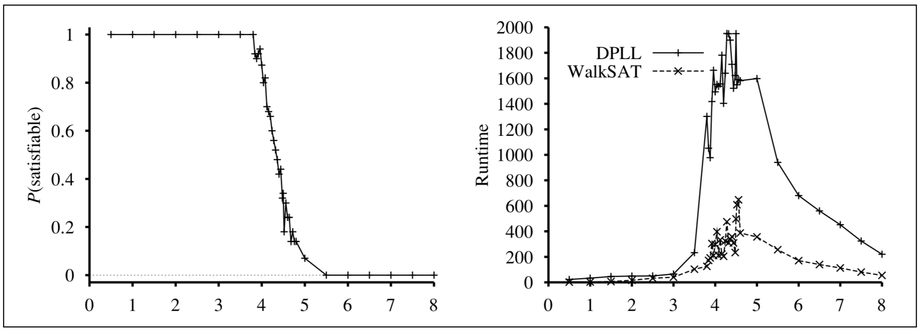

Figure 8: (Left) Graph showing the probability that a random sentence with n = 50 symbols is satisfiable, as a function ratio m/n. (Right) Graph of the median run time on random sentences. The most difficult problems have a ratio of about 4.3.

3 First Order logic

3.1 Syntax and Semantic

One of the most important proprieties about propositional logic that holds for First Order Logic (FOL from now on) is the **compositionality**, that is the meaning of a sentence is a function of the meaning of its parts. For example given S_1 =*I have don't want to write this* and S_2 =*I would love to watch Netflix* it would be weird if $S_1 \wedge S_2$ =*Tomorrow will be cold.*
Moreover propositional logic assumes that there are facts that either hold or do not hold in the world [54], while FOL assumes that the world consists of objects with certain relations among them that do or do not hold. Hence in both logic a sentence represents a fact and the agent either believes the sentence to be true, believes it to be false, or has no opinion.

Structure FOL assumes the world contains:

- **Objects** like: pc, tv, beer....
- **Relations** are sets of tuples of objects that are related. Depending on the number of tuple in the set we can have:
  - *Proprieties* (or unary relationship) that have just one tuple in the set; such as:*red, round, big...*
  - *General* (or n-ary relationship) which may have n tuples; like: *bigger than, has color, owns...*
- **Functions** For example: *father of, best friend...*

Example Given the sentence *"The next MidTerm will be easier then the first one"*[55] I have that:

- *Objects*: Midterm, one
- *Proprieties*: next, first (unary)
- *Functions*: easier than

Symbols For each one on the structures we have a related symbol:

- **Constants**: stands for objects.
- **Predicates**: stands for relations.
- **Functions**: stands for functions [56]

The last two comes with an **arity** that is the number of arguments they are referring to [57].

[54] Each fact can be in one of two states: true or false.
[55] Probably not a model.
[56] You don't say.
[57] For example the function *Sibling(x,y)* has arity 2 (since we need a subject and a brother), while the function *Human(x)* has arity 1.

Interpretation vs Model A model in first-order logic consists of a set of objects and an interpretation that maps constant symbols to objects, predicate symbols to relations on those objects, and function symbols to functions on those objects.
Given a domain $\mathcal{D}$, an interpretation is a function $\mathcal{I}$ which maps:

- Every constant symbol c into an element of $\mathcal{D}$.
- Every n-ary[58] function symbol f into a function f^I: $\mathcal{D}^n \to \mathcal{D}$. More specifically f is a function which may assume any value in the domain $\mathcal{D}^n$, while f^I assumes just one ($\mathcal{D}$).
- Every predicate symbol P into a n-ary relation P^I: $P^I \subset \mathcal{D}^n$. This means that a predicate symbol, e.g. ????????????????

Terms are a logical expressions that refers to an object. That is what is used to refer to a particular object in the domain; they can be:

- **Constants** symbols; for example the color red, or that guy Eric over there.
- **Functional** terms; which are functions applied to constant symbols such as: *BestFriendOf(Eric)* or *Mother(Marta)*.

Notice that both refer to an "object" in the domain, one directly (constants), while the other indirectly. A functional term can be used both for not known constants, like the name of Marta's mother, but they can also be used for thing we do not bother to name, e.g. *LeftArmOf(Edoardo)*.

Atomic Formulas An atomic formula (or atom for short) is a predicate symbol with a number $n \in [0, \infty]$ of terms. For example given $t_1, ..., t_n$ terms, the following are all atoms:

- *Sibling*(t_i, t_j): predicate symbol with arity 2.
- $Friend(t_1, ..., t_n)$: predicate symbol with arity n.
- $t_i = t_j$: predicate symbol = with arity 2, can be written as $Equals(t_i, t_j)$
- $ParentOf(ParentOf(x))$ is the grandparent of x and has arity 1.
- $\bot, \top$ are atoms.

Terms vs Atoms So what is the difference between this two symbols?
The difference is in the use of relationships, for example:

- *BatCave, CaveOf(Batman)* are terms, while *Big(BatCave),Big(CaveOf(Batman))* are atoms.
- Equal symbols generate atoms, for example *MobileOf(Batman)* and *BatMobile* are terms, where *MobileOf(Batman)=BatMobile* is an atom.

Moreover a term is called **ground** if it does not contain variables [59], like *CaveOf(Batman)*.

Complex Sentences When we have atoms joint by some logic connectives then we call them **complex sentence**, for example:

- $\neg Friends(Superman, LexLutor)$
- $Superman \wedge Kryponite \Rightarrow Weak(Superman)$

[58] A function with n arguments.
[59] Later explained

Equality The equations symbol = is used to indicate that two terms refer to the same object [60], while the negation symbol $\neq$ is used otherwise.

Database Semantics

- **Unique name assumption**: every constant symbol refers to a distinct object.
- **Close-world assumption**: atomic sentences not known to be true are false.
- **Domain closure**: each model contains no more domain elements than those named by the constant symbols.

3.1.1 Quantifiers

We first introduce the concept of **variables** [61] that are equals to terms, thus can be used as arguments for functions. Moreover a **free** variable is one that is not in the scope of any quantifier, elsewhere the variable is called **bounded**.
Then we have to talk about **extended interpretations** that specifies a domain [62] element for which a variable exists

Universal Quantifier If we consider the sentence *Every guy literally only want one thing* [63], instead of enumerating every possible guy we can use:

$$\forall x \quad Guy(x) \Rightarrow WantOneThing(x)$$

that is translated in:

For all x, if x is a guy, then x wants one thing.

Implication vs Conjunction The question that arises is why do we have to use implication in the universal quantifier and not a conjunction?

Consider the following example:

- *A(X)= is an apple.*
- *D(X)= is delicious.*

We want to consider an universe U in which *all* the apples are delicious, i.e. if the object is an apple then it is delicious, we must write:

$$\forall x \in U,\ A(x) \Rightarrow D(x)$$

That means for all objects in the universe, if x is an apple $A(x)$, then x is delicious $D(x)$. If we take a look at the truth table (Figure 2) we notice that the implication is has three rows in which is $True$, for every x:

1. If it is an apple and it is delicious.
2. If it is not an apple and it is delicious.

[60] Can be used to state fact about the world.
[61] Variables will be in lowercase.
[62] The domain is the set of individual objects.
[63] And it's fucking disgusting!

3. If it is not an apple and it is not delicious.

This means that if we have an avocado as x then the above formula will be true. So why is this? The important thing to understand is that we only care about an universe in which *all* apples are delicious, i.e. there can not be any non-delicious apple [64], nothing matters when it comes down to avocado. So the fact than an implication becomes vacuously for non-apple object is not in our scope of interest.
On the other hand, if we want to exclude the other two possibilities (2,3), we are saying that *"any fruit is an apple and is delicious"*[65], hence *all fruits are delicious apples*:

$$\forall x \in U,\ A(x) \land D(x)$$

Existential Quantifier On the other hand, if we do not want to specify a propriety for each object of a domain, but either we want to say that it exists *at least one* object with a certain propriety we use the **existential quantifier**, e.g. *Some student will pass the midterm*:

$$\exists x \quad Student(x) \land Pass(x)$$

Which is read as:

There exists some x such that x is a student and x will pass the Midterm (hopefully).

Conjunction vs Implication Same as before the question is : *why do we use conjunction for existential quantifier*?
Like previously said, the conjunction for an universal quantifier makes the formula overly strong [66], while the use of the implication for the existential quantifiers makes the formula overly week.
Going back to the apple example, we want to say that there are some apples which are delicious, if we use the implication symbol then the formula will return $True$ as soon as there is an object which is either a delicious non-apple or a non-delicious non-apple, so the look for some delicious apple stops immediately.
It is useful to imagine the existential quantifier to be like a disjunction in which as soon as we encounter a $True$ element we can stop the look for other and just return $True$. On the other hand the implication needs to look for all the objects since its behavior is closer to the one of a conjunction.

Nested Quantifiers We can use multiple quantifiers in the same formula to express different kind of proprieties. For example, given the function $Love(x, y)$ that can be read as, x loves y, we can say the following things:

- Everybody loves somebody:
$$\forall x \exists y Loves(x, y)$$
- Somebody loves us
$$\forall x \exists y Loves(y, x)$$
- X loves everybody
$$\exists x \forall y Loves(x, y)$$
- Everybody loves x
$$\exists x \forall y Loves(y, x)$$

[64] First line of the truth table.
[65] Overly strong statement.
[66] All objects are delicious apples

Connection between ∀ and ∃ Saying that *everyone likes professor Nardi* is like saying that *there is no one who dislikes him.* So we can make use of this information to switch between universal and existential quantifiers as shown in Table 3.

Quantifiers	Equivalences
$\forall x\ \neg P \equiv \neg\exists x\ P$	$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$
$\neg\forall x\ P \equiv \exists x\ \neg P$	$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$
$\forall x\ P \equiv \neg\exists x\ \neg P$	P $\wedge Q \equiv (P \vee \neg Q)$
$\exists x\ P \equiv \neg\forall x\ \neg P$	P$\vee Q \equiv \neg(\neg P \wedge \neg Q)$
$\forall x\ (P_1 \vee P_2) \equiv \forall x\ P_1 \vee P_2$	if x $\notin var(P_2)$
$\exists x\ (P_1 \wedge P_2) \equiv \exists x\ P_1 \wedge P_2$	if x $\notin var(P_2)$
$\forall x P(x) \equiv \forall y P(y)$	$\exists x P(x) \equiv \exists y P(y)$
$\forall x \forall y P(x,y) \equiv \forall y \forall x P(x,y) \equiv \forall x, y P(x,y)$	$\exists x \exists y P(x,y) \equiv \exists y \exists x P(x,y) \equiv \exists x, y P(x,y)$
$\forall x P(y) \equiv P(y)$	$\exists x P(y) \equiv P(y)$
$(\forall x P_1) \Rightarrow P_2 \equiv \exists x(P_1 \Rightarrow P_2)$	x not free in P_2
$(\exists x P_1) \Rightarrow P_2 \equiv \forall x(P_1 \Rightarrow P_2)$	x not free in P_2
$P_2 \Rightarrow \forall x P_1 \equiv \forall x(P_2 \Rightarrow P_1)$	x not free in P_2
$P_2 \Rightarrow \exists x P_1 \equiv \exists x(P_2 \Rightarrow P_1)$	x not free in P_2

Table 3: Quantifiers equivalences

3.2 Using First Order Logic

Sentences which are added to the KB using *TELL* are called **assertion** that states some truth about the world. On the other hand, queries are information *ASK*ed to the KB, if they are logically entailed by the KB they should be answered positively. Similarly the *AskVars* action ask the KB for the values of the formula that make it true, such answers are called substitutions (Section 3.4) [67].
Depending on the type of knowledge the KB is made out of we may have:

- **Intentional Knowledge**: which are general laws about the domain, such as $\forall x, y\ Mother(x, y) \Leftrightarrow Famale(x) \wedge Parent(x, y)$.
- **Extensional Knowledge**: facts about a specific instance, like $Parent(Marco, Ugo)$.

FOL formulas In FOL formulas follows this rules, given the formulas A, B:

- Every atom is a formula.
- $\neg A$ is a formula.
- If $\circ$ is a binary operator, then $A \circ B$ is a formula.
- Given a variable x, $\forall x\ A$ and $\exists x\ A$ are formulas.

[67] Usually reserved for KB of only Horn Clauses.

3.3 Truth, Interpretation and Models

Remember that a *closed formula*, also called a sentence, is a formula without free variables. So given a sentence ϕ and a structure $\mathcal{U} = \langle D, I \rangle$ we denote the truth of ϕ as:

$$\mathcal{U} \models \phi$$

Meaning that the $\mathcal{U}$ satisfies [68] ϕ if ϕ is true in the $\mathcal{U}$. The following cases are used for satisfiability:

- $\mathcal{U} \models \top$ and $\mathcal{U} \not\models \bot$
- If A is a closed formula $P(t_1, ..., t_n)$ then

$$\mathcal{U} \models A \quad iff \quad \langle t_1^I, ..., t_n^I \rangle \in P^I$$

- $\mathcal{U} \models \neg A$ iff $\mathcal{U} \not\models A$
- $\mathcal{U} \models A \wedge B$ iff $(\mathcal{U} \models A) \wedge \mathcal{U} \models B$
- $\mathcal{U} \models (A \Rightarrow B)$ iff $(\mathcal{U} \models A) \Rightarrow (\mathcal{U} \models B)$
- $\mathcal{U} \models (A \leftrightarrow B)$ iff $(\mathcal{U} \models A) \wedge (\mathcal{U} \models B)$ or $(\mathcal{U} \not\models A) \wedge (\mathcal{U} \not\models B)$
- $\mathcal{U} \models \forall x \ A$ iff $\forall d \in D, \mathcal{U} \models \{d \rightarrow x\}$
- $\mathcal{U} \models \exists x \ A$ iff $\exists d \in D, \mathcal{U} \models \{d \rightarrow x\}$

For what regard models we have that if $\mathcal{U} \models A$ then $\mathcal{U}$ is a **model** of A [69].

Validity and satisfiability We have that if a formula $A \in \mathcal{L}$ is true in every structure of a language $\mathcal{L}$, then A is **valid**.
On the other hand, given a set of formulas Γ, if there exists a structure $\mathcal{U}$ such that $\mathcal{U} \models A$ for every $A \in \Gamma$ then Γ is **satisfiable**, mathematically:

$$\Gamma = \{A_1, ..., A_n\}, \ iff \ \exists \, \mathcal{U} : \mathcal{U} \models A_i, \forall A_i \in Gamma \ \Rightarrow \Gamma \ satisfiable$$

We can derive the notion of **logic entailment** $\gamma \models A$, where A is a closed formula, form the satisfiability, in the following way:
For every structure $\mathcal{U}$ of the language $\mathcal{L}$, $\forall B \in \Gamma : \mathcal{U} \models B$, we have $\mathcal{U} \models A$ then $\gamma \models A$.

3.4 Propositional vs First Order Logic

First-order inference can be done by converting the KB to propositional logic and using propositional inference. This can be done by replacing the quantifiers as explained in the following paragraphs.

[68] Note that in this case $\models$ is read satisfies and it is red.
[69] A is true in $\mathcal{U}$.

3.4.1 Universal instantiation

UI for short, is used for the universal quantifier and it says that we can substitute a variable with a ground term [70] to infer the sentence.
The **substitution** is written an follow:

$$Subst(\theta, \alpha) = \frac{\forall v \ \alpha}{Subst(\{v/g\}, \alpha)}$$

where θ is the substitution applied to the term α for any variable v and ground term g.
The UI can be applied many times to produce multiple consequences that will be added to the KB preserving its logical equivalence.

An Example Given the formula:

$$\forall x, y \quad Loves(x, y)$$

We can substitution like this:

$$Subst(\{x/Anna, y/Enrico\}, Loves(x, y)) = Loves(Anna, Enrico)$$

And like this:

$$Subst(\{x/Enrico, y/Anna\}, Loves(x, y)) = Loves(Enrico, Anna)$$

3.4.2 Existential instantiation

EI for short, in this case the variable is replace with *a single* constant symbol C that **does not** appear in the KB. The idea is saying that there is an object C that satisfy the existential quantifier, but we are not specifying which is it.
After applying EI we need to remove the existential quantifier [71] and our new KB will *not* be logically equivalent to the old one, but it can be shown that it is satisfiable exactly when the original KB is satisfiable.

An Example Given the formula:

$$\exists x \ Game(x) \land PlayOnPc(x, Nicolo)$$

We have *the only* substitution:

$$Game(G_1) \land PlayOnPc(G_1, Nicolo)$$

Where G_1 is a generic constant symbol referring to some cool game I want to play.

3.4.3 Propositionalization

Is the technique of reducing first-order inference to propositional inference. The problem is that with function symbols we may have an infinite domain [72], so how can we preserve the entailment?
Thanks to Jacques Herbrand (1930) we have a theorem which states the following: *if a sentence is entailed by the original, first-order KB, then there is a proof involving just a finite subset of the propositionalized KB*. The problem with propositionalization is that it generates a lot of irrelevant sentences, with p predicates of $k-$arity and n constants we have $p \cdot n^k$ instantiations.
The question of entailment for first-order logic is *semidecidable*, that is, algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every non-entailed sentence

[70] Term without variables.
[71] Skolemization.
[72] For example *Father(Father(Father(Father(Jhon))))*.

An Example Given the following KB:

$$\forall x\ Student(x) \wedge Stressed(x) \Rightarrow StudyingAI(x)$$

$$Student(Edoardo), Student(Marco), Professor(Nardi)$$

We ca use UI to have:

$$Student(Edoardo) \wedge Stressed(Edoardo) \Rightarrow StudyingAI(Edoardo)$$

$$Student(Marco) \wedge Stressed(Marco) \Rightarrow StudyingAI(Marco)$$

$$Student(Edoardo), Student(Marco), Professor(Nardi)$$

So that the new propositionalized KB is:

$$Student(Edoardo), Stressed(Edoardo), StudyingAI(Edoardo), Student(Marco),$$

$$Stressed(Marco), StudyingAI(Marco), Professor(Nardi)$$

Horn Clauses We can have the following for every element of the FOL:

- *Facts*: $\top(x)$ ($A(x)$ for simplicity).
- *Rules*: $A_1(x) \wedge ... \wedge A_n(x) \Rightarrow B(x)$
- *Goals*: $A_1(x) \wedge ... \wedge A_n(x) \Rightarrow \bot$ [73]

3.4.4 Generalized Modus Ponens

When we have an implication of the kind:

$$p_1 \wedge ... \wedge p_n \Rightarrow q$$

If there is a substitution θ which makes every conjunct of the premises identical to a sentence in the KB, then we can infer q.

The **Generalized Modus Ponens** [GMP] works in the following way: given n atomic sentences $p_1, ..., p_n$ which are the premises of an implication $p_1 \wedge ... \wedge p_n \Rightarrow q$, and given n atomic sentences already in the KB $p'_1, ..., p'_n$ so that $Subst(\theta, p_i) = Subst(\theta, p'_i)$ [74] for all i, we can use *GMP*:

$$\frac{p'_1, ..., p'_n, \quad (p_1 \wedge ... \wedge p_n \Rightarrow q)}{Subst(\theta, q)}$$

And infer a valid substitution for the query q.

Example Given a KB with the following information:

1. $\forall y\ Tired(y)$, everyone [75] in the domain is tired.
2. $\forall x\ Student(x) \wedge Stressed(x) \wedge Tired(x) \Rightarrow NeedCoffe(x)$
3. $Student(Edoardo), Student(Luca), Stressed(Luca)$

Considering the second line of the KB, we have that:

[73] We are using $\bot$ since want to prove the falsity of the premise is impossible so the premise is true.

[74] This is that given the substitution θ we can transform a sentence p into a known sentence p' in the KB.

[75] Actually every object.

- $p_1 = Tired(x)$
- $p_2 = Student(x)$
- $p_3 = Stressed(x)$
- $q = NeedCoffe(x)$

So now we need a substitution θ such that $Subst(\theta, p') = Subst(\theta, p)$. Trying with:

$$\theta = \{x/Edoardo, y/Edoardo\}$$

Will not work since the KB has no notion about $Stressed(Edoardo)$ [76]. On the other hand, Luca is pretty stressed so we can use:

$$\theta = \{x/Luca, y/Luca\}$$

So we can conclude that $Subst(\theta, q) \longrightarrow Luca$.

Soundness It is easy to prove that the *GMP* is sound since we need to prove that: given a sentence p and a substitution θ, we have:

$$p \models Subst(\theta, p)$$

But this holds since it exactly what we have using *UI*.

3.4.5 Unification

As before mentioned we need to find a substitution such that different logical expression looks identical ($Subst(\theta, p) = Subst(\theta, p')$). We can use the **Unification** process that takes as input two sentences and returns an unifier for them[77]:

$$Unify(p, q) = \theta \; Subst(\theta, p) = Subst(\theta, q)$$

We can have problems when having variables with the same name [78], so we need to use a technique called **standardizing apart** which changes one of the two sentence's variables to avoid name clashes.
Finally we can say that, for every unifiable pair of expressions, there is a single **most general unifier** that is unique up to renaming and substitution of variables, for example $P(A, B)$ is less general than $P(A, x)$ that is less general than $P(y, x)$.

Example Given the following KB:

1. $Knows(Homer, Marge)$, Homer knows Marge.
2. $\forall y \; Knows(y, Bart)$, everyone knows Bart.
3. $\forall y \; Knows(y, Mother(y))$, everyone knows their mother.
4. $\forall x \; Knows(x, Lisa)$, everyone knows Lisa.
5. $\forall y, z \; Knows(y, z)$, everyone knows everyone.

We ask the KB which are the persons that *Homer* knows: $q = AskVars(Knows(Homer, x))$. So we will have the following:

[76] Since the first line of the KB, implies that every one in the domain is tired the formula $Tired(x)$ will always be true.
[77] If exists.
[78] As you will see in the next example.

1. $Unify(Knows(Homer, x), Knows(Homer, Marge) = \{x/Marge\}$
2. $Unify(Knows(Homer, x), Knows(y, Bart) = \{x/Bart, y/Homer\}$
3. $Unify(Knows(Homer, x), Knows(y, Mother(y)) = \{y/Homer, x/Mother(y)\}$
4. $Unify(Knows(Homer, x), Knows(x, Lisa) = Fail$, because the variable x cannot be both $Homer$ and $Lisa$ at the same time [79], so we use **standardizing apart**.
5. $Unify(Knows(Homer, x), Knows(y, z)$, returns two substitutions:
   (a) $\theta = \{y/Homer, x/z\}$, which is the **most general unifier**.
   (b) $\theta = \{y/Homer, x/Homer, z/Homer\}$, which may be generated by the previous substitution with $\{z/Homer\}$.

3.5 Chaining

Consider the following problem:

The law says that it is a crime for an American to sell kinder eggs to hostile nations. The Easter Island, an enemy of America, has some kinder eggs, and all of its kinder eggs were sold to it by Roger Rabbit, who is American.

We want to prove that Roger Rabbit is a criminal. First we need to convert the problem to a first-order definite clause KB:

1. "...it is a crime for an American to sell kinder eggs to hostile nations" results in:

   $$American(x) \land KinderEgg(y) \land Sells(x, y, z) \land Hostile(z) \Rightarrow Criminal(x)$$

   [80]

2. "The Easter Island...has some kinder eggs", the sentence can be written as an existential formula of the form:

   $$\exists x\ Owns(EasterIsland, x) \land KinderEgg(x)$$

   We can use *EI* 3.4.2 to remove the existential quantifier by adding some new constant K_1 resulting in:

   $$Owns(EsterIsland, K_1), \quad KinderEgg(K_1)$$

3. "... all of its kinder eggs were sold to it by Roger Rabbit":

   $$KinderEgg(x) \land Owns(EsterIsland, x) \Rightarrow Sells(RogerRabbit, x, EsterIsland)$$

4. We must know that an hostile object is an enemy of America:

   $$Enemy(x, America) \Rightarrow Hostile(x)$$

5. "Roger Rabbit is American":

   $$American(RogerRabbit)$$

6. "The Easter Island, an enemy of America":

   $$Enemy(EsterIsland, America)$$

[79] It should follow the behavior of point 2.
[80] From now on we will not write the $\forall x, y, z$ symbol. Note that this kind of KB is called **Datalog**.

3.5.1 Forward Chaining

Starting from the known facts, the algorithm triggers all the rules whose premises are satisfied, adding their conclusions to the known facts. The process repeats until the query is answered or no new facts are added. Notice that a fact is not new if it is just a **renaming** of a known fact [81]. The steps for the forward chaining are the following:

- Rule 3 is satisfied with $Subst(\theta, x) = \{x/K_1\}$, so that $Sells(RogerRabbit, K_1, EsterIsland)$ is added to the KB.
- Rule 4 is satisfied by $Subst(\theta, x) = \{x/EsterIsland\}$, so that $Hostile(EsterIsland)$ is added.
- Finally rule 1 is satisfied with $Subst(\theta, x, y, z) = \{x/RogerRabbit, y/K_1, z/EsterIsland\}$ so we can add $Criminal(RogerRabbit)$.

  Notice that we reached a **fixed point** in which no more inferences can be induced, but the difference with the propositional fixed point is that we may have universally quantified atomic sentence.
  Moreover the algorithm is **sound** since every step is an application on *GMP* and complete for definite clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses.
  Finally let k be the maximum arity (number of arguments) of any predicate, p be the number of predicates, and n be the number of constant symbols. Clearly, there can be no more than pn^k distinct ground facts, so after this many iterations the algorithm must have reached a fixed point.

  Efficiency The previous mentioned process may be improved with :

  - **Conjunction ordering**: the problem of finding an ordering to solve the conjuncts of the rule premise so that the total cost is minimized. It arises when we have many facts in the KB that need to be matched against a premise.
    Given the rule 3 we have to find objects which are both $KinderEggs$ owned by $EsterIsland$; so we must iterate through all the n $KinderEggs$ and all the m objects owned by $EsterIsland$. It is logical that if $n \ll m$ it will be faster to find the $KinderEggs$ first and then the object owned by $EsterIsland$.
    This problem is exactly as the CSP-problem studied before [82], thus being NP-Hard.
  - **Incremental Forward Checking**: another problem is the one of matching redundant rules. It can be avoided if we consider that a rule is *"new"* at iteration t if it has been inferred by some facts generated in the previous iteration $t-1$, otherwise it would have been generated earlier.
    Another problem is given by **partial matching** rules which are generated and discarded as soon as some atoms in the premise do not hold. Instead of discarding them we could leave them semi-generated and gradually complete them as new facts arrives.
  - **Irrelevant Facts**: finally, the algorithm does not distinguish between relevant and irrelevant facts. To address this issue we can either use backward chaining or deductive databases that uses forward chaining as the standard inference rule.

[81] For example *Likes(x,ice-scream)* and *Likes(y,ice-scream)* are renaming of each other with the same meaning (everybody likes ice-scream).

[82] We can use the studied techniques such as *minimum-remaining-value...*

3.5.2 Backward Chaining

The backward chaining is a kind of AND/OR search, the OR part because the goal query can be proved by any rule in the knowledge base, and the AND part because all the conjuncts in a list of clauses must be proved.
It is a depth-first search so it suffers from problem such as repeated states and incompleteness.
It is the base for **Logic programming [PROLOG]**

4 Resolution in First Order Logic

For applying resolution we need to convert a sentence in *Conjunctive Normal Form* CNF [83], fortunately every sentence in FOL can be converted into an inferentially equivalent sentence in CNF; the difference between propositional CNF and FOL CNF is the presence of the existential quantifier.

4.1 Skolem Normal Form

When removing existential quantifiers using EI (Section 3.4.2) we are assuming there exist an element, which we do not know, that satisfy the existential property.
Now consider the sentence *"Everyone has a heart"*, that is translated in FOL with:

$$\forall x[Person(x) \Rightarrow \exists y \ Heart(y) \wedge Has(x, y)]$$

If we use EI on thi s we obtain:

$$\forall x[Person(x) \Rightarrow Heart(H) \wedge Has(x, H)]$$

The meaning of the sentence becomes *"Everyone has the heart H"*, which is not the same as before since we make no distinction between hearts.
For this kind of problem we need a function which takes as input a person and returns the person's heart. Let us denote this function as $F(x)$ [84], so the above sentence becomes:

$$\forall x[Person(x) \Rightarrow Heart(F(x)) \wedge Has(x, F(x))]$$

which now links every person to his own heart, $F(x)$ is called a **Skolem function**.
Formally given a formula $\exists x\psi(x)$, where $\psi(x)$ is another formula that depends on x [85], we have that the skolemization of the formula ($\exists x\psi^{sko}$) is given by substituting the dependence on x with a skolem function $\psi[F(x_1, ..., x_n)/x]$ of arity n.

4.1.1 Prenex Normal Form

Before introducing the *Skolem Normal Form* [SNF] we must describe the *Prenex Normal Form* [PNF].
A for the CNF that needs the negation symbols $\neg$ to be pushed inward [86], the PNF wants the quantifier symbols to be moved *outwards* to the left side of the formula. For example, given some quantifier free formulas $\phi(x), \psi(y), \varphi(z)$ [87] which appears in a formula as:

$$[\forall x \ \phi(x) \wedge \forall y \ \psi(y)] \Rightarrow \exists z \ \varphi(z)$$

We transform it in PNF just by pushing the quantifiers to the left:

$$\forall x \forall y \exists z[[\phi(x) \wedge \psi(y)] \Rightarrow \varphi(z)]$$

[83] Section 2.2.1.
[84] Note that this function must not appear in the KB, similarly as for EI.
[85] Note that the dependence can be of any kind of *arity*. The heart example mentioned before has arity 2 for the $Has(x, y)$ predicate, but we may have predicates in which the variable of the existential quantifier x may depend on a huge number of other variables $v_1, ..., v_n$ so to have arity n.
[86] That is inside the formulas, to the left of literals.
[87] These formulas are made of other formulas like : $\phi(x) = P(x) \vee Q(x) \vee A$. The important thing is that they do now contain any quantifier and they must depend on the specified variable.

It is easy to understand that the two different implementation are equivalent $\equiv$.

Generally speaking a formula ψ in in PNF if:

- Does not contain quantifiers:
  - no variable occurs in ψ, e.g. $\top$.
  - ψ has only free variables, e.g. $A \vee \neg B$.
- All the quantifiers are pushed to the left-most side. Formally given $Q = \{\vee, \wedge\}$, A is a quantifier free formula and we have $x_1, x_2, ..., x_n$ variables; ψ is in the form:

$$Q_1 x_1 Q_2 x_2 ... Q_n x_n A$$

Transformation to PNF Given a generic formula ϕ the steps for transforming it into PNF are the following:

1. Build a formula ϕ' where only $\vee, \wedge, \exists, \forall$ occur, negation is pushed inwards and double negations are eliminated.
2. Rename bound variables so that each quantifier uses a different variable [88].
3. Build the Prenex formula moving all quantifiers to the left.

4.1.2 Complete Example

We make the following example:

Everyone who loves all animal is loved by someone.

This sentence is translated into FOL as:

$$\forall x[\forall y\ Animal(y) \Rightarrow Loves(x, y)] \Rightarrow [\exists y\ Loves(y, x)]$$

The steps for reducing the sentence into SNF are the following:

1. **Eliminate implication**:

$$\forall x[\neg\forall y\ \neg Animal(y) \vee Loves(x, y)] \vee [\exists y\ Loves(y, x)]$$

2. **Move $\neg$ inwards**:

$$\forall x[\exists y\ \neg\neg Animal(y) \wedge \neg Loves(x, y)] \vee [\exists y\ Loves(y, x)]$$

$$\forall x[\exists y\ Animal(y) \wedge \neg Loves(x, y)] \vee [\exists y\ Loves(y, x)]$$

The sentence now reads "either there is some animal that x doesn't love, or (if this is not the case) someone loves x".

3. **Standardize variables**, that is removing variables with the same name:

$$\forall x[\exists y\ Animal(y) \wedge \neg Loves(x, y)] \vee [\exists z\ Loves(z, x)]$$

[88] Technique known as standardization.

4. **Skolemize** to remove the existential quantifier, using two skolem function $F(x), G(z)$ not present in the KB:
$$\forall x[Animal(F(x)) \land \neg Loves(x, F(x))] \lor Loves(G(z), x)$$
5. **Drop universal quantifier** using UI 3.4.1 and get:
$$[Animal(F(x)) \land \neg Loves(x, F(x))] \lor Loves(G(z), x)$$
6. **Distribute** over $\lor, \land$ and obtain a SNF with two clauses:
$$[Animal(F(x)) \lor Loves(G(z), x)] \land [\neg Loves(x, F(x)) \lor Loves(G(z), x)]$$

4.1.3 Proprieties

Let ψ be a formula and ψ^{sko} be its SNF, we have the following proprieties:

- $\mathcal{M}$ model of ψ **does not imply** $\mathcal{M}$ model of ψ^{sko}.
- $\mathcal{M}$ model of ψ^{sko}**implies** $\mathcal{M}$ model of ψ.
- ψ valid **does not imply** ψ^{sko} valid.
- ψ^{sko} valid **implies** ψ valid.
- ψ satisfiable **iff** ψ^{sko} satisfiable.
- ψ unsatisfiable **iff** ψ^{sko} unsatisfiable.
- ψ contradictory **iff** ψ^{sko} contradictory.
- ψ valid **iff** $\neg\psi^{sko}$ valid.

4.2 Resolution Inference Rule

In propositional logic two clauses can be resolved if they contain complementary literals, in FOL two clauses are complementary if one *unifies* with the negation of the other.

4.2.1 Binary Resolution

Given :
$$\frac{l_1 \lor ... \lor l_k, \quad m_1 \lor ... \lor m_n}{Subst(\theta, l_1 \lor ... \lor l_{i-1} \lor l_{i+1} \lor ... \lor l_k \lor m_1 \lor ... \lor m_{i-1} \lor m_{i+1} \lor ... \lor m_n)}$$
Where $Unify(l_i, \neg m_i) = \theta$. For example:
$$[Animal(F(x)) \lor Loves(G(x), x)] \quad and \quad [Loves(u, v) \lor \neg Kills(u, v)]$$
can be resolved with $\theta = [u/G(x), v/x]$ and generate:
$$[Animal(F(x)) \lor \neg Kills(u, v)]$$

Incompleteness of binary resolution This type of resolution is called a *binary* since it resolves two literals, l_i, m_i or $Loves(G(x), x), Loves(u, v)$, and it can be shown to be incomplete (why???????????). But it can be made complete with the use of **factoring** that removes two identical literals [89].

[89] In FOL two literals are identical if they are unifiable.

4.2.2 General Resolution

Same as for preopositional logic.

4.2.3 Completeness of Resolution

The objective is to show that resolution is **refutation-complete**, which means that if a set of sentences is unsatisfiable, then resolution will always be able to derive a contradiction. Hence we can use resolution to establish that a certain sentence is entailed by a set of sentences (KB) [90] effectively answering a query $Q(x)$ by proving that $KB \wedge \neg Q(x)$ is unsatisfiable.
The proof follows the steps of Figure 9.

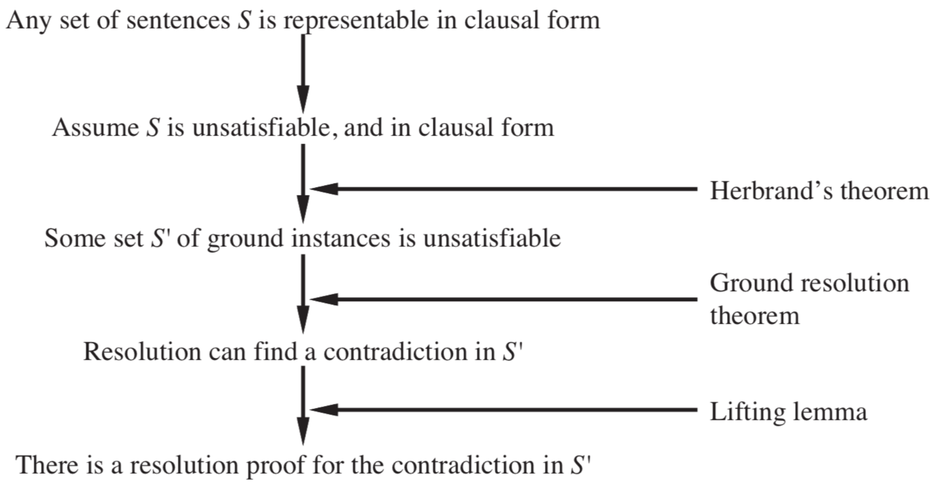

Figure 9: Structure of a completeness proof for resolution.

Before explaining the proof we need to introduce some concepts.

Herbrand Universe Given a set of clauses S the Herbrand universe H_S of those clauses is constructed in the following way:

- Following a function symbol if any.
- Following a constant symbol if any.
-

Per example given $Parent(x, F(x, A)) \vee Sibling(x, B)$ we can construct $Parent(Parent(A), F(F(A), A)), Parent(Paren$
That is we construct all the possible **ground terms** with the given function symbols ($Parent(x, y), Sibling(x, y)$), constant symbols ($A, B, F(A), Sibling(A, B)...$) and skolem formulas ($F(A, B), F(B, B), F(F(A), B)...$).

Saturation Given the set of ground terms P obtained earlier, we generate the saturation of S in respect to P, called $P(S)$, by applying all consistent substitution of ground term in P with variables in S.

Herbrand Base The saturation of a set of clauses S in respect to a Herbrand Universe is called a Herbrand Base, denoted $H_S(P)$ [91].

[90] Although we can not use resolution to derive all possible consequences from a KB.
[91] Which is infinite.

Herbrand's Theorem The theorem says that:

If a set S of clauses is unsatisfiable, then there exists a finite subset of $H_S(S)$ that is also unsatisfiable.

Let S' be this finite subset of $H_S(S)$, we can use the ground resolution theorem (Section 2.2.3) to show that the resolution closure $RC(S')$ contains the empty clause. Hence we have demonstrated that there is *always* a resolution proof involving some subset of $H_S(S)$, now we need to establish that there exists some proof involving S itself [92].

Lifting Lemma This lemma is used as a proof step from ground clauses up to general first-order clauses. The lemma says the following:

Let C_1 and C_2 be two clauses with no shared variables, and let C_1' and C_2' be ground instances of C_1 and C_2. If C' is a resolvent of C_1 and C_2 , then there exists a clause C such that (1) C is a resolvent of C_1 and C_2 and (2) C' is a ground instance of C.

For example, given the two clauses:

$$C_1 = Heart(x, F(x, A)) \vee Human(x)$$

$$C_2 = \neg Heart(y, F(y, B)) \vee Cyborg(y) \vee Oil(y, G(y, C))$$

We instantiate them with some ground terms (constant symbols only for simplicity):

$$C_1' = Heart(Goku, F(Goku, A)) \vee Human(Goku)$$

$$C_2' = \neg Heart(C18, F(C18, B)) \vee Cyborg(C18) \vee Oil(C18, G(C18, C))$$

Next we apply the resolution rule with $Human(x)$:

$$C' = Cyborg(C17) \vee Oil(C17, G(C17, C)) \vee Human(C17)$$

And finally we generalize to get:

$$C' = Cyborg(y) \vee Oil(y, G(y, C)) \vee Human(y)$$

4.3 Resolution Strategies

In the following we will analyze some strategies to make resolution more efficient.

Unit preference Is the technique of preferring to use a single literal (unit clause) [93] to resolve against other clauses. This is because the length [94] of the resolution will be less than the other clause we used. For example, resolving:

$$\frac{P \vee \neg Q \vee R, \quad F \vee \neg R \vee A}{P \vee \neg Q \vee F \vee A}$$

Which has length 4, higher than the two clauses used for resolution, while resolving:

$$\frac{P, \quad F \vee \neg P \vee A}{F \vee A}$$

That has length 2, shorter than the second clause.

[92] Note that the difference is that S may not contain only ground clauses.
[93] For example P, not like $Q \vee R$.
[94] Counted as the number of element in the disjunction.

Set of Support We keep a set of clauses which are considered useful for the resolution inference, when we resolve something using an element from this set we add the resolvent to the set. Usually this set contains the negated query and generate a goal-directed proof tree.

Input Resolution Is the technique of using a sentence from the input (either the query or the KB) and resolving it with another sentence. The **linear resolution** strategy is a slight generalization that allows P and Q to be resolved together either if P is in the original KB or if P is an ancestor of Q in the proof tree. Linear resolution is complete.

5 Prolog

Prolog is the major logic-based programming language [95], it uses a set of definite clauses written in a notation somewhat different from FOL.
For example:

$$American(x) \wedge (y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$$

becomes

$$criminal(X)\ :-\ american(X), kinderEgg(Y), sells(X, Y, Z), hostile(Z)$$

Note how the variables have *uppercase* letters while the relationships are in *lowercase*. As for the FOL case $criminal(X)$ is true if every atom in the premise $american(X), kinderEgg(Y), sells(X, Y, Z), hostile(Z)$ is also true.

An Example We write a program with the following rules:

1. $append([\,], Y, Y)$: if I append to an empty list $[\,]$ another list Y, then the result will be the same list Y.
2. $append([A|X], Y, [A|Z])\ :-append(X, Y, Z)$: the premises (on the right) says that appending X to Y produces Z, if this is true then appending Y to $[A|X]$ [96] will output $[A|Z]$.

And we query $append(X, Y, [1, 2])$ (what can be the values of X, Y so that, if I append them, I have $[1, 2]$?) and obtain:

- $X = [\,] \quad Y = [1, 2]$
- $X = [1] \quad Y = [2]$
- $X = [1, 2] \quad Y = [\,]$

5.1 Efficient Implementation

There are two modes of operation for the Prolog program.

[95] It is a sub-set of FOL.
[96] This notation denotes a list whose first element is A and rest is X.

function FOL-BC-ASK(KB, $query$) **returns** a generator of substitutions
  return FOL-BC-OR(KB, $query$, { })

---

generator FOL-BC-OR(KB, $goal$, θ) **yields** a substitution
  for each rule ($lhs \Rightarrow rhs$) in FETCH-RULES-FOR-GOAL(KB, $goal$) **do**
    (lhs, rhs) $\leftarrow$ STANDARDIZE-VARIABLES((lhs, rhs))
    for each θ' **in** FOL-BC-AND(KB, lhs, UNIFY(rhs, $goal$, θ)) **do**
      yield θ'

---

generator FOL-BC-AND(KB, $goals$, θ) **yields** a substitution
  if $\theta = failure$ **then return**
  else if LENGTH($goals$) = 0 **then yield** θ
  else do
    $first$,$rest$ $\leftarrow$ FIRST($goals$), REST($goals$)
    for each θ' **in** FOL-BC-OR(KB, SUBST(θ, $first$), θ) **do**
      for each θ'' **in** FOL-BC-AND(KB, $rest$, θ') **do**
        yield θ''

Figure 10: Abstract Interpreted Prolog Mode

Interpreted The algorithm works as shown in Figure 10 with two major improvements:

- The presence of a **global stack**[97] of choice points to keep track of multiple possibilities to consider .
- The use of a **trail** in which logic variables [98] are kept. So when a branch fails the algorithm removes the last bounded variable and continues with the trail *without* re-instantiating a substitution for each variable.

Compiled In this mode the Prolog program becomes an inference procedure for a specific set of clauses, so it knows what clauses match the goal. Prolog basically generates a miniature theorem prover for each different predicate, thereby eliminating much of the overhead of interpretation. ??????????????????

Parallelization There are two principal sources of parallelism. The first, called **OR** parallelism, comes from the possibility of a goal unifying with many different clauses in the knowledge base. Each gives rise to an independent branch in the search space that can lead to a potential solution, and all such branches can be solved in parallel. The second, called textbfAND parallelism, comes from the possibility of solving each conjunct in the body of an implication in parallel.

5.2 Redundant inference and infinite loops

Since Prolog works with backward chaining, hence using a depth-first search, some issues may arise. Specifically the problem of infinite paths makes Prolog **incomplete** as a theorem prover for definite clauses because it fails to prove sentences that are entailed.

[97] Remember that a stack is the implementation of a depth-first search.
[98] Variables that remember their current bindings .

An example Given two implementation of the same problem:

1. The first one:
   - path(X,Z) :- link(X,Z)
   - path(X,Z) :- path(X,Y), link(Y,Z)
2. The second one:
   - path(X,Z) :- path(X,Y), link(Y,Z)
   - path(X,Z) :- link(X,Z)

Since Prolog chooses the clauses in the order of implementation, the resulting search tree for the second implementation will be an infinite loop, as shown in Figure 11.

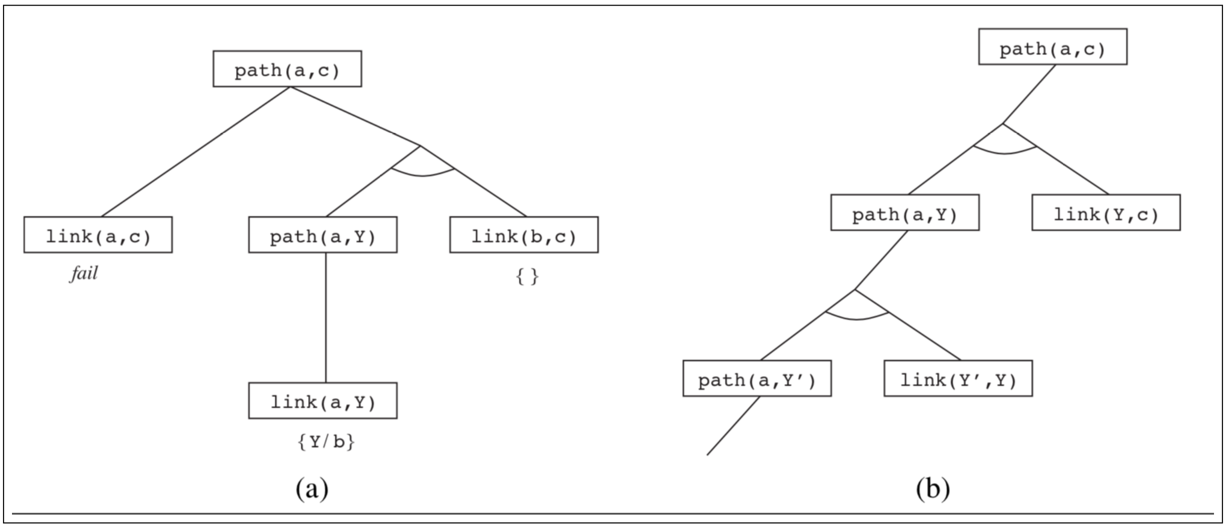

Figure 11: Abstract Interpreted Prolog Mode

6 Exercises

6.1 First Order Logic

6.1.1 Ex1

Choose a suitable vocabulary of constant and predicate symbols then represent the following sentences in FOL:

1. Steve likes easy classes
2. Violin classes are not easy
3. Every class of Percussion is easy
4. The class of "Afro Drums" is a class of Percussion
5. The class of "Violin for pre-school Kids" is a class of Violin

The KB will look something like this:

1. $\forall x \; Class(x) \wedge Easy(x) \Rightarrow Likes(Steve, x)$
2. $\forall x \; Class(x) \wedge Violin(x) \Rightarrow \neg Easy(x)$
3. $\forall x \; Class(x) \wedge Percussion(x) \Rightarrow Easy(x)$
4. $Class(AfroDrums), Percussion(AfroDrums)$
5. $Class(ViolinKids), Violin(ViolinKids)$

Notice that the second formula is not a definite clause but a goal clause (Section 2.2.2). This makes the KB in Horn normal Form.

Can we infer that Steve likes the class of Afro Drums ($Likes(Steve, x), \; \{x/AfroDrums\}$)?

- using Modus Ponens
- using Resolution

Using Modus Ponens The steps are the following:

- Use MP for 3,4:

$$\frac{Class(AfroDrums), Percussion(AfroDrums), \quad [Class(x) \wedge Percussion(x) \Rightarrow Easy(x)]}{Easy(AfroDrums)}$$

  With the substitution $\theta = \{x/AfroDrums\}$. Then add $Easy(AfroDrums)$ to the KB.

- Finally use MD on 1 using the ground term we just deducted from the previous step:

$$\frac{Easy(AfroDrums), Class(AfroDrums), \quad [\forall x \; Class(x) \wedge Easy(x) \Rightarrow Likes(Steve, x)]}{Likes(Steve, AfroDrums)}$$

Resolution We first need to bring the KB in Skolem Normla Form SNF (Section 4.1) [99].

1. $\neg Class(x) \vee \neg Easy(x) \vee Likes(Steve, x)$

[99] Note that there are no existential quantifiers, so there is no need for skolem unctions. On the other hand SNF is not only about skolem functions, universal quantifiers are treated too with UI (Section 3.4.1).

2. $\neg Class(y) \vee \neg Violin(y) \vee Easy(y)$
3. $\neg Class(z) \vee \neg Percussion(z) \vee Easy(z)$
4. $\neg q = \neg Likes(Steve, g)$

So we end up with three clauses and can start the resolution mechanism. We want to derive the query $q = Likes(Steve, AfroDrums)$ from the KB, $KB \vdash_{\mathcal{R}} q$, hence we need to prove that $(KB \vee \neg q) \vdash_{\mathcal{R}} \{\}$. First we use the 1,4, with the substitution $\theta = \{x/AfroDrums\}$ and get:

$$\frac{\{\neg Class(x), \neg Easy(x), Likes(Steve, x)\} \quad \{\neg Likes(Steve, g)\}}{\{\neg Class(AfroDrums), \neg Easy(AfroDrums)\}}$$

Note that this is a case of **binary resolution** (Section 4.2.1). Next we use the facts in the KB.

$$\frac{\{\neg Class(AfroDrums), \neg Easy(AfroDrums)\}, \quad \{Class(AfroDrums)\}}{\{\neg Easy(AfroDrums)\}}$$

And finally we use:

$$\frac{\{\neg Easy(AfroDrums)\}, \quad \{Easy(AfroDrums)\}}{\{\}}$$

Which entails the empty clause, thus $(KB \vee \neg q) \vdash_{\mathcal{R}} \{\}$ is true, thus $KB \vdash_{\mathcal{R}} q$.