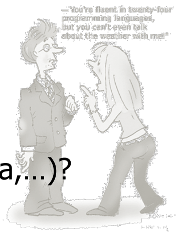
Artificial Intelligence 1

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

Knowledge Representation: First-Order Logic

- We are designing **knowledge-based agents** – they combine and recombine information about the world with current observations to uncover hidden aspects of the world and use them for action selection.
- How to represent **knowledge?**
  - so far **propositional logic**
  - today **first-order logic**
- How to **reason** over that knowledge?
  - so far **model checking** (DPLL, WalkSAT)
  - today **resolution**
    - forward and backward chaining

We are looking for a **formal language** that can
- **represent knowledge**
- **reason with knowledge**



What about **programming languages** (C++, Java,…)?
- this is the most widely used class of formal languages
- facts are described via **data structures**
  - array world[4,4]
- **programs** describe how to do computations (changing data structures)
  - world[2,2] ← pit
- How to infer new information from existing facts?
  - ad-hoc procedures changing data structures → a **procedural approach**
  - a **declarative approach** separates knowledge and inference mechanism (moreover, inference is general and problem independent)
- How to represent knowledge such as „pit at [2,2] or [3,1]"?
  - variables in computer programs have unique values

- Can we use **natural languages** (English, Czech, ...) to represent knowledge?
  - That would be great but there is no precise formal semantics for these languages!
  - Currently, natural languages are seen as a **medium for communication** rather than for pure representation.
    - the sentence itself does not code information, it also depends on **context**
      - „Look!"
    - another problem is **ambiguity** of natural languages
      - spring, …

Back to logic

- **Propositional logic** is declarative with compositional semantic that is context-independent and unambiguous.
- However, some properties are cumbersome (not easy to model).
  - Wumpus: there is breeze next to a pit
    - $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$
    - $B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$
    - …
- Let us take inspiration from natural languages:
  - we have nouns representing **objects** (pit, square,…)
  - verbs express **relations** between the objects (is next to, …)
  - some relations are in fact **functions** (is a father of)
- Instead of pure facts (propositional logic) we will work with objects, relations, and functions. We will also express facts about some or all objects (**first-order predicate logic - FOL**).

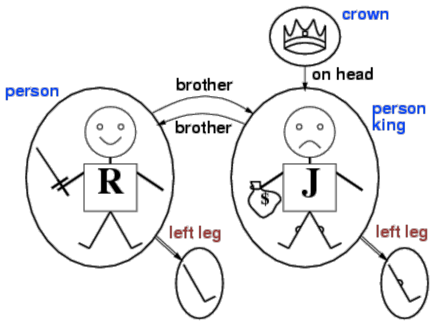
Logical frameworks - a survey

Propositional logic	facts that hold or not
First-order logic	objects and relations between them
Temporal logic	facts and times when they hold
Higher-order logic	relations between objects are used as objects (there are claims on relations)

First-order logic: syntax

- constants — John, 2, Crown,...
- predicates — Brother, >,...
- functions — Sqrt, LeftLeg,...
- variables — x, y, a, b,...
- connectives — $\neg$, $\Rightarrow$, $\wedge$, $\vee$, $\Leftrightarrow$
- equality — =
- quantifies — $\forall$, $\exists$

First-order logic: an example



- **constants** (names of objects):
  - Richard, John, TheCrown
- **function symbols**:
  - LeftLeg
- **terms** (another form to name objects)
  - LeftLeg(John)
- **predicate symbols**:
  - Brother, OnHead, Person, King, Crown
- **atomic sentences** (describe relations between objects):
  - Brother(Richard,John)
- **complex sentences**:
  - King(Richard) $\vee$ King(John)
  - $\neg$King(Richard) $\Rightarrow$ King(John)
- **quantifiers** (help to define sentences over more objects):
  - $\forall$x King(x) $\Rightarrow$ Person(x)
  - Beware: $\forall$x King(x) $\wedge$ Person(x) !!!
  - $\exists$x Crown(x) $\wedge$ OnHead(x,John)
  - Beware: $\exists$x Crown(x) $\Rightarrow$ OnHead(x,John) !!!

- $\forall$x,y Brother(x,y) $\Rightarrow$ Brother(y,x)
- $\exists$x,y Brother(x,Richard) $\wedge$ Brother(y,Richard)
- $\exists$x,y Brother(x,Richard) $\wedge$ Brother(y,Richard) $\wedge$ $\neg$(x=y)

Equality says that two terms refer to the same object (Father(John) = Henry).

Quantifiers

- **Universal quantifier ∀x P**
  - P is true for any object x
  - corresponds to a conjunction of all formulas P
    - P(John) ∧ P(Richard) ∧ P(TheCrown) ∧ P(LeftLeg(John)) ∧ …
  - Typically connected with implication (to select the objects for which the sentence holds)
    - ∀x King(x) ⇒ Person(x)
- **Existential quantifier ∃x P**
  - there is an object x such that P holds for it
  - corresponds to a disjunction of all formulas P
    - P(John) ∨ P(Richard) ∨ P(TheCrown) ∨ P(LeftLeg(John)) ∨ …
- **Relations between quantifiers**
  - ∀x ∀y is identical to ∀y ∀x
    ∃x ∃y is identical to ∃y ∃x
  - ∃x ∀y is not identical to ∀y ∃x (∃x ∀y Loves(x,y) vs.∀y ∃x Loves(x,y))
  - ∀x P is identical to ¬∃x ¬P
    ∃x P is identical to ¬∀x ¬P

FOL and knowledge base

- Similarly to propositional logic we will use **operations TELL** to add a sentence to knowledge base:
  - TELL(KB, King(John))
  - TELL(KB,∀x King(x) ⇒ Person(x))
  - We are typically adding **axioms** (**facts** as atomic sentences, **definitions** using ⇔ and other complex sentences) and sometime even **theorems** (can be deduced from axioms, but they "speed up" further inference).
- and **operations ASK** for querying the sentences entailed by KB:
  - ASK(KB, King(John))
  - ASK(KB, Person(John))
  - ASK(KB, ∃x Person(x))

a database query

we need some inference here

in addition to YES/NO answers we also ask for the value of x for which the sentence holds – substitution {x/John}

Inference in FOL

- We can do inference in propositional logic. Let us extend it to first-order logic now.

- The main differences:
  - quantifiers→ **skolemization**
  - functions and variables → **unifications**
- The rest techniques are known:
  - **forward chaining** (deduction databases, production systems)
  - **backward chaining** (logic programming)
  - **resolution** (theorem proving)

Reducing FOL to PL

- **Reasoning in first-order logic can be done by conversion to propositional logic and doing reasoning there.**
  - Grounding (propositionalization)
    - instantiate variables by all possible terms
    - atomic sentences then correspond to propositional variables
  - And what about quantifiers?
    - universal quantifiers: each variable is substituted by a term
    - existential quantifier: skolemization (variable is substituted by a new constant)

Reducing FOL to PL: quantifiers

Universal instantiation

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

For a variable v and a grounded term g, apply substitution of g for v.
Can be applied more times for different terms g.

– Example: ∀x King(x) ∧ Greedy(x) ⇒ Evil(x) leads to:
  King(John) ∧ Greedy(John) ⇒ Evil(John)
  King(Richard) ∧ Greedy(Richard) ⇒ Evil(Richard)
  King(LeftLeg(John)) ∧ Greedy(LeftLeg(John)) ⇒ Evil(LeftLeg(John))
  …

Existential instantiation

$$\frac{\exists v\ \alpha}{\text{Subst}(\{v/k\}, \alpha)}$$

For a variable v and a new constant k, apply substitution of k for v.
Can be applied once with a new constant that has not been so far (Skolem constant)

– Example: ∃x Crown(x) ∧ OnHead(x,John) leads to:
  $\text{Crown}(C_1) \wedge \text{OnHead}(C_1,\text{John})$

Reducing FOL to PL – an example

- Let us start with a knowledge base in FOL (no functions yet):
  ∀x King(x) ∧ Greedy(x) ⇒ Evil(x)
  King(John)
  Greedy(John)
  Brother(Richard,John)
- By assigning all possible constants for variables we will get a knowledge base in propositional logic:
  King(John) ∧ Greedy(John) ⇒ Evil(John)
  King(Richard) ∧ Greedy(Richard) ⇒ Evil(Richard)
  King(John)
  Greedy(John)
  Brother(Richard,John)
- Inference can be done in propositional logic then.

Problem: having even a single function symbol gives infinite number of terms LeftLeg(John), LeftLeg(LeftLeg(John)),…
– Herbrand: there is an inference in FOL from a given KB if there is an inference in PL from a finite subset of a fully instantiated KB
– We can add larger and larger terms to KB until we find a proof.
– However, if there is no proof, this procedure will never stop ☹.

Inference in FOL

- We can modify the inference rules to work with FOL:
  – **lifting** – we will do only such substitutions that we need to do
  – **lifted Modus Ponens rule:**

$$\frac{p_1, p_2, \ldots, p_n, q_1 \wedge q_2 \wedge \ldots \wedge q_n \Rightarrow q}{\text{Subst}(\theta,q)}$$

  where θ is a substitution s.t. $\text{Subst}(\theta,p_i) = \text{Subst}(\theta,q_i)$
  (for **definite clauses** with exactly one positive literal - **rules**)
  – We need to find substitution such that two sentences will be identical (after applying the substitution)
    - King(John) ∧ Greedy(y)   King(x) ∧ Greedy(x)
    - substitution {x/John, y/John }

Unification

- **How to find substitution θ such that two sentences p and q are identical after applying that substitution?**
  – Unify(p,q) = θ , where Subst(θ,p) = Subst(θ,q)

p	q	θ
Knows(John,x)	Knows(John,Jane)	{x/Jane}
Knows(John,x)	Knows(y,OJ)	{x/OJ, y/John}
Knows(John,x)	Knows(y,Mother(y))	{y/John, x/Mother(John)}}
Knows(John,x)	Knows(x,OJ)	{fail}

  – **What if there are more such substitutions?**
    Knows(John,x)   Knows(y,z),
    ➯ θ = {y/John, x/z } or θ = {y/John, x/John, z/John}
    - The first **substitution is more general** than the second one (the second substitution can be obtained by applying one more substitution after the first substitution {z/John}).
  – There is a unique (except variable renaming) substitution that is more general than any other substitution unifying two terms – **the most general unifier** (mgu).

Unification algorithm

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
   inputs: x, a variable, constant, list, or compound
           y, a variable, constant, list, or compound
           θ, the substitution built up so far

   if θ = failure then return failure
   else if x = y then return θ
   else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
   else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
   else if COMPOUND?(x) and COMPOUND?(y) then
       return UNIFY(ARGS[x], ARGS[y], UNIFY(OP[x], OP[y], θ))
   else if LIST?(x) and LIST?(y) then
       return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
   else return failure
```

explore the sentences recursively and build mgu until obtaining trivially unifiable or different sentences

complex terms must have the same "name" and unifiable arguments

list are being unified separately to omit cycles when representing the list as a term (First,Rest)

```
function UNIFY-VAR(var, x, θ) returns a substitution
   inputs: var, a variable
           x, any expression
           θ, the substitution built up so far

   if {var/val} ∈ θ then return UNIFY(val, x, θ)
   else if {x/val} ∈ θ then return UNIFY(var, val, θ)
   else if OCCUR-CHECK?(var, x) then return failure
   else return add {var/x} to θ
```

Checking occurence of variable *var* in term *x*
- *x* and *f(x)* are not unifiable
- gives quadratic time complexity
- there are also linear complexity algorithms
- not always done (Prolog)

Standardizing apart

- Assume a **query** Knows(John, x).
- We can find an answer in the knowledge base by finding a fact **unifiable with the query**:

  Knows(John, Jane) → {x/Jane}
  Knows(y, Mother(y)) → {x/Mother(John)}
  Knows(x, Elizabeth) → fail
  - ???
  - Knows(x,Elizabeth) means that anybody knows Elizabeth (existential quantifier is assumed there), so John knows Elizabeth.
  - The problem is that both sentences contain variable x and hence cannot be unified.
  - ∀x Knows(x,Elizabeth) is identical to ∀y Knows(y,Elizabeth)
  - Before we use any sentence from KB, we rename its variables to new fresh variables not ever used before – **standardizing apart**.

Example

- According to US law, any American citizen is a criminal, if he or she sells weapons to hostile countries. Nono is an enemy of USA. Nono owns missiles that colonel West sold to them. Colonel West is a US citizen.
- Prove that West is a criminal.

… any US citizen is a criminal, if he or she sells weapons to hostile countries:
American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)

Nono … owns missiles, i.e. ∃x Owns(Nono,x) ∧ Missile(x):
Owns(Nono,M_1) and Missile(M_1)

… colonel West sold missiles to Nono
Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)

Missiles are weapons.
Missile(x) ⇒ Weapon(x)

Hostile countries are enemies of USA.
Enemy(x,America) ⇒ Hostile(x)

West is a US citizen …
American(West)

Nono is an enemy of USA …
Enemy(Nono,America)

Inference techniques

All sentences in the example are definite clauses and there are no function symbols there.

To solve the problem we can use:

- **forward chaining**
  - using Modus Ponens we can infer all valid sentences
  - this is an approach used in deductive databases (Datalog) and production systems
- **backward chaining**
  - we can start with a query **Criminal(West)** and look for facts supporting that claim
  - this is an approach used in logic programming

Forward chaining in FOL

function FOL-FC-ASK(KB, α) **returns** a substitution or $false$

  repeat until new is empty

    $new \leftarrow \{\}$

    for each sentence r **in** KB **do**

      $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-APART(r)

      for each θ such that $(p_1 \wedge \ldots \wedge p_n)\theta = (p'_1 \wedge \ldots \wedge p'_n)\theta$

        for some $p'_1, \ldots, p'_n$ in KB

        $q' \leftarrow$ SUBST(θ, q)

        if q' is not a renaming of a sentence already in KB or new **then do**

          add q' to new

          $\phi \leftarrow$ UNIFY(q', α)

          if ϕ is not $fail$ **then return** ϕ

    add new to KB

  return $false$

take a rule from KB and rename its variables (standardizing apart)

infer all conclusions not yet present in KB

if we infer a sentence unifiable with the query then we can return corresponding mgu

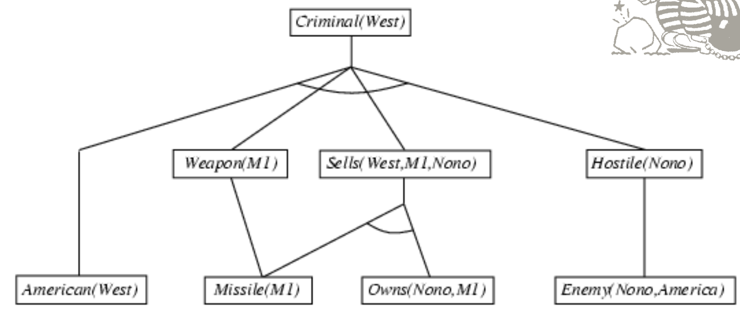
obtained sentences are placed to KB as theorems and the whole process is repeated

Forward chaining is a **sound** and **complete** inference algorithm.

– Beware! If the sentence is not entailed by KB the algorithm may not finish (if there is at least one function symbol).

Forward chaining – an example

American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
Owns(Nono,M1) a Missile(M1) (vzniklo z ∃x Owns(Nono,x) ∧ Missile(x))
Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)
American(West)
Enemy(Nono,America)



Forward chaining – pattern matching

for each θ such that $(p_1 \wedge \ldots \wedge p_n)\theta = (p'_1 \wedge \ldots \wedge p'_n)\theta$
for some $p'_1, \ldots, p'_n$ in KB

How to find (fast) a set of facts $p`_1,\ldots, p`_n$ unifiable with the body of the rule?

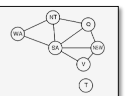
- This is called **pattern matching**.
- Example 1: *Missile(x) ⇒ Weapon(x)*
  - we can **index the set of facts** according to predicate name so we can omit failing attempts such as *Unify(Missile(x),Enemy(Nono, America))*
- Example 2: *Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)*
  1. we can find objects own by Nono which are missiles …
  2. or we can find missiles that are owned by Nono

  Which order is better?
  - Start with less options (if there are two missiles while Nono owns many objects then alternative 2 is faster) – **recall the first-fail heuristic** from constraint satisfaction

!

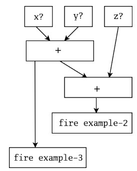
Pattern matching is an NP-complete problem.

Diff(wa,nt) ∧ Diff(wa,sa) ∧ Diff(nt,q) ∧ Diff(nt,sa) ∧ Diff(q,nsw) ∧ Diff(q,sa) ∧ Diff(nsw,v) ∧ Diff(nsw,sa) ∧ Diff(v,sa) ⇒ Colorable()
Diff(Red,Blue), Diff (Red,Green), Diff(Green,Red), Diff(Green,Blue), Diff(Blue,Red), Diff(Blue,Green)



Forward chaining – an incremental approch

- **Example**: *Missile(x) ⇒ Weapon(x)*
  - during the iteration, the forward chaining algorithm infers that all known missiles are weapons
  - during the second (and every other) iteration the algorithm deduces exactly the same information so KB is not updated
- When should we use the rule in inference?
  - if there is a new fact in KB that is also in the rule body
- **Incremental forward chaining**
  - a rule is fired in iteration t, if a new fact was inferred in iteration (t-1) and this fact is unifiable with some fact in the rule body
  - when a new fact is added to KB, we can verify all rules such that the fact unifies with a fact in rule body
  - **Rete algorithm**
    - the rules are pre-processed to a **dependency network** where it is faster to find the rules to be fired after adding a new fact



Forward chaining – a magic set

- Forward chaining algorithm deduces all inferable facts even if they are not relevant to a query.
  - to omit it we can use **backward chaining**
  - another option is modifying the rules to work only with relevant constants using a so called **magic set**

Example: query Criminal(West)

Magic(x) ∧ American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)

Magic(West)

  - The magic set can be constructed by backward exploration of used rules.

Production systems

TESTY

COMPUTERWORLD 2, 2007 17

Řízení obchodních pravidel levně a jednoduše

JAMES OWEN

- based on rete algorithm
- XCON (R1)
  - configuration of DEC computers
- OPS-5
  - programming language based on forward chaining
- CLIPS
  - A tool for expert system design from NASA
- Jess, JBoss Rules,…
  - business rules

Backward chaining in FOL

function FOL-BC-ASK(KB, goals, θ) **returns** a set of substitutions
  inputs: KB, a knowledge base
    $goals$, a list of conjuncts forming a query
    θ, the current substitution, initially the empty substitution { }
  local variables: ans, a set of substitutions, initially empty
  if $goals$ is empty **then return** $\{\theta\}$
  $q' \leftarrow$ SUBST(θ, FIRST($goals$))
  for each r **in** KB where STANDARDIZE-APART(r) = $(p_1 \land \ldots \land p_n \Rightarrow q)$
    and $\theta' \leftarrow$ UNIFY(q, q') succeeds
   $ans \leftarrow$ FOL-BC-ASK(KB, $[p_1, \ldots, p_n|$REST$(goals)]$, COMPOSE$(\theta, \theta')) \cup ans$
  return ans

take the first goal and apply the so-far found substitution

find a rule whose head is unifiable with the first goal (from query)

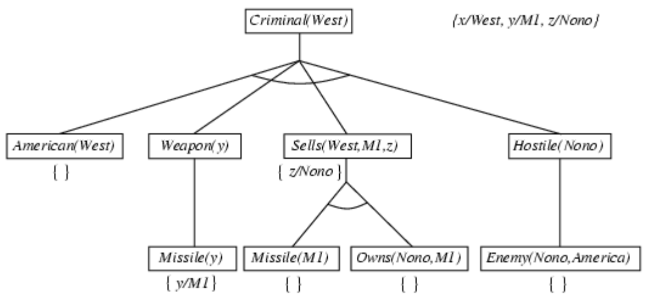
add the rule body among the goals and recursively continue in goal reduction until obtaining an empty goal

composition of substitutions
Subst(Compose(θ, θ'), p) = Subs(θ', Subst(θ,p))

Algorithm FOL-BC-Ask uses **depth-first search** to find **all solutions** (all substitutions) to a given query.
We need **linear space** (in the length of the proof).
This algorithm is **not complete** (the same goals can be explored again and again).

Backward chaining – an example

American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
Owns(Nono,M1) a Missile(M1) (vzniklo z ∃x Owns(Nono,x) ∧ Missile(x))
Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)
American(West)
Enemy(Nono,America)



Logic programming

- Backward chaining is a method used in **logic programming** (Prolog).

rule head   rule body

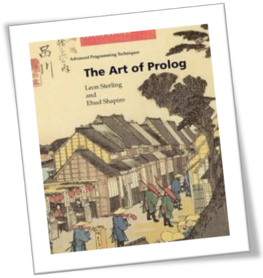
```
criminal(X) :-
    american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
owns(nono,m1).
missile(m1).
sells(west,X,nono) :-
    missile(X), owns(nono,X).
weapon(X) :-
    missile(X).
hostile(X) :-
    enemy(X,america).
american(west).
enemy(nono,america).

?- criminal(west).
```

```
?- criminal(west).
?- american(west), weapon(Y),
   sells(west,Y,Z), hostile(Z).
?- weapon(Y), sells(west,Y,Z),
   hostile(Z).
?- missile(Y), sells(west,Y,Z),
   hostile(Z).
?- sells(west,m1,Z), hostile(Z).
?- missile(m1), owns(nono,m1),
   hostile(nono).
?- owns(nono,m1), hostile(nono).
?- hostile(nono).
?- enemy(nono,america).
?- true.
```

Logic programming - properties

- **fixed computation mechanism**
  - goal is reduced from left to right
  - rules are explored from top to down
- returns a **single solution**, a next solution on request
  - possible cycling (`brother(X,Y) :- brother(Y,X)`)
- build-in **arithmetic**
  - `X is 1+2.`
  - (numerically) evaluates the expression on right and unifies the result with the term on the left
- **equality** gives explicit access to unification
  - `1+Y = 3.`
  - It is possible to naturally exploit constraints (CLP – Constraint Logic Programming)
- **negation as failure**
  - `alive(X) :- not dead(X).`
    - „everyone is alive, if we cannot prove he is dead "
  - *¬Dead(x) ⇒ Alive(x)* is not a definite clause!
    - *Alive(x) ∨ Dead(x)*
    - „Everyone is alive or dead"



Resolution – a conjunctive normal form

To apply a **resolution method** we first need a formula in a **conjunctive normal form**.

- ∀x [∀y Animal(y) ⇒ Loves(x,y)] ⇒ [∃y Loves(y,x)]
- **remove implications**
  ∀x [¬∀y ¬Animal(y) ∨ Loves(x,y)] ∨ [∃y Loves(y,x)]
- **put negation inside** (¬∀x p ≡ ∃x ¬p, ¬ ∃x p ≡ ∀x ¬p)
  ∀x [∃y ¬(¬Animal(y) ∨ Loves(x,y))] ∨ [∃y Loves(y,x)]
  ∀x [∃y ¬¬Animal(y) ∧ ¬Loves(x,y)] ∨ [∃y Loves(y,x)]
  ∀x [∃y Animal(y) ∧ ¬Loves(x,y)] ∨ [∃y Loves(y,x)]
- **standardize variables**
  ∀x [∃y Animal(y) ∧ ¬Loves(x,y)] ∨ [∃z Loves(z,x)]
- **Skolemize** (Skolem functions)
  ∀x [Animal(F(x)) ∧ ¬Loves(x,F(x))] ∨ [Loves(G(x),x)]
- **remove universal quantifiers**
  [Animal(F(x)) ∧ ¬Loves(x,F(x))] ∨ [Loves(G(x),x)]
- **distribute ∨ and ∧**
  [Animal(F(x)) ∨ Loves(G(x),x)] ∧ [¬Loves(x,F(x)) ∨ Loves(G(x),x)]

Resolution – inference rules

- A lifted version of the resolution rule for first-order logic:

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{(\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)\theta}$$

  where Unify(ℓ_i, $\neg m_j$) = θ.
- We assume standardization apart so variables are not shared by clauses.
- To make the method complete we need:
  - extend the binary resolution to more literals
  - use **factoring** to remove redundant literals (those that can be unified together)

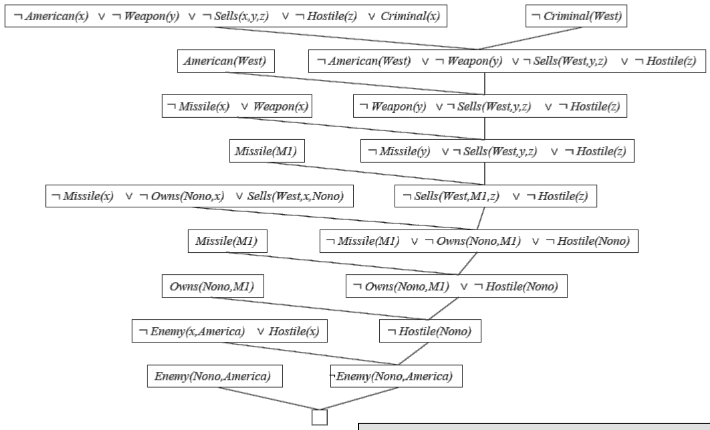
Example:

$$\frac{[\text{Animal}(F(x)) \vee \text{Loves}(G(x),x)], \qquad [\neg\text{Loves}(u,v) \vee \neg\text{Kills}(u,v)]}{[\text{Animal}(F(x)) \vee \neg\text{Kills}(G(x),x)]}$$

where θ = {u/G(x), v/x}

- Query α for KB is answered by applying the resolution rule to CNF(KB ∧ ¬α).
  - If we obtain an empty clause, then KB ∧ ¬α is not satisfiable and hence KB ⊢ α.
- This is a **sound** and **complete** inference method for first-order logic.

Resolution method – an example



Resolution method applied to definite clauses is actually **backward chaining**, where the clauses to resolve are determined.

American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)
Owns(Nono,M1) a Missile(M1) (vzniklo z ∃x Owns(Nono,x) ∧ Missile(x))
Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)
Missile(x) ⇒ Weapon(x)
Enemy(x,America) ⇒ Hostile(x)
American(West)
Enemy(Nono,America)

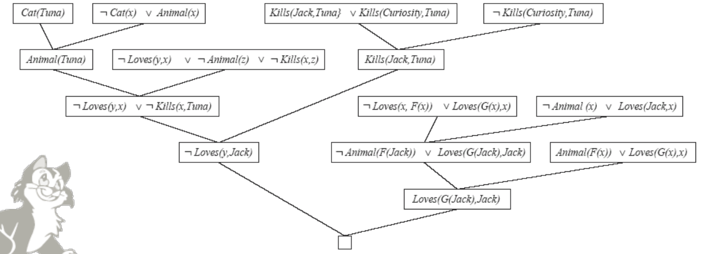
Resolution – a complex example

- Everyone, who likes animals, is loved by somebody. Everyone, who kills animals, is loved by nobody. Jack likes all animals. Either Jack or Curiosity killed cat named Tuna. Cats are animals.
- Did Curiosity kill Tuna?

∀x [∀y Animal(y) ⇒ Loves(x,y)] ⇒ [∃y Loves(y,x)]

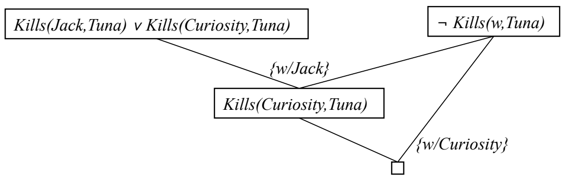
∀x [∃y Animal(y) ∧ Kills(x,y)] ⇒ [∀z ¬Loves(z,x)]
∀x Animal(x) ⇒ Loves(Jack,x)
Kills(Jack,Tuna) ∨ Kills(Curiosity, Tuna)
Cat(Tuna)
∀x Cat(x) ⇒ Animal(x)
¬ Kills(Curiosity,Tuna)

Animal(F(x)) ∨ Loves(G(x),x)
¬ Loves(x,F(x)) ∨ Loves(G(x),x)
¬ Animal(y) ∨ ¬Kills(x,y) ∨ ¬ Loves(z,x)
¬ Animal(x) ∨ Loves(Jack,x)
Kills(Jack,Tuna) ∨ Kills(Curiosity,Tuna)
Cat(Tuna)
¬ Cat(x) ∨ Animal(x)
¬ Kills(Curiosity,Tuna)



Resolution – non-constructive proofs

- What if the query is **„Who did kill Tuna?"**



- The answer is **„Yes, somebody killed Tuna** ".
- We can include an **answer literal** in the query.
  *¬ Kills(w,Tuna) ∨ **Answer(w)***
  - The previous non-constructive proof would give now:
    Answer(Curiosity) ∨ Answer(Jack)
  - Hence we need to use the original proof leading to:
    ¬ Kills(Curiosity,Tuna)

Resolution strategies

How to **effectively** find proofs by resolution?

- **unit resolution**
  - the goal is obtaining an empty clause so it is good if the clauses are shortening
  - hence we prefer a resolution step with a unit clause (contains one literal)
  - in general, one cannot restrict to unit clauses only, but for Horn clauses this is a complete method (corresponds to forward chaining)
- **set of support**
  - this is a special set of clauses such that one clause for resolution is always selected from this set and the resolved clause is added to this set
  - initially, this set can contain the negated query
- **input resolution**
  - each resolution step involves at least one clause from the input – either query or initial clauses in KB
  - this is not a complete method
- **subsumption**
  - eliminates clauses that are subsumed (are more specific than) by another sentence in KB
  - having P(x), means that adding P(A) and P(A) ∨ Q(B) to KB is not necessary