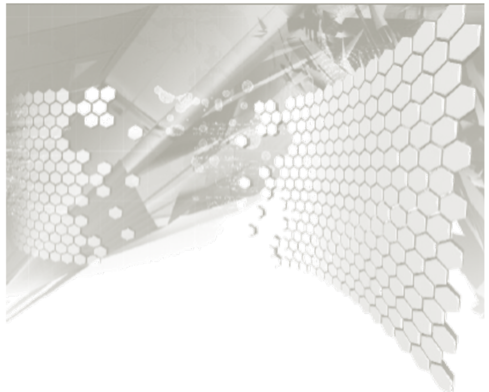
Artificial Intelligence 1

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

First-Order Logic: Inference Techniques

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



The main differences:

- quantifiers $\rightarrow$ **skolemization**
- functions and variables $\rightarrow$ **unification**

The core inference principles are known:

- **forward chaining** (deduction databases, production systems)
- **backward chaining** (logic programming)
- **resolution** (theorem proving)

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)

Universal instantiation

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

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: $\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$ leads to:

  King(John) $\wedge$ *Greedy(John)* $\Rightarrow$ *Evil(John)*

  King(Richard) $\wedge$ *Greedy(Richard)* $\Rightarrow$ *Evil(Richard)*

  King(LeftLeg(John)) $\wedge$ *Greedy(LeftLeg(John))* $\Rightarrow$ *Evil(LeftLeg(John))*

  …

Existential instantiation

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

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 used so far (Skolem constant)

– Example: $\exists x\ Crown(x) \wedge OnHead(x,John)$ leads to:

  $Crown(C_1) \wedge OnHead(C_1,John)$

Reducing FOL to PL: an example

Let us start with a knowledge base in FOL (**no functions** yet):

$\forall x\ (King(x) \wedge Greedy(x) \Rightarrow Evil(x))$

King(John)

Greedy(John)

Brother(Richard,John)

By assigning all possible constants for variables we will get a knowledge base in propositional logic:

$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$

$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$

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 ☹.

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:**

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

  where θ is a substitution s.t. $\mathrm{Subst}(\theta,p_i) = \mathrm{Subst}(\theta,q_i)$

  (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)* $\wedge$ *Greedy(y)*   *King(x)* $\wedge$ *Greedy(x)*
  - substitution *{x/John, y/John}*

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_1 = *{y/John, x/z}* or θ_2 = *{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=*empty*) **returns** a substitution to make x and y identical, or *failure*
  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*

function UNIFY-VAR(var, x, θ) **returns** a substitution
  if $\{var/val\} \in \theta$ for some val **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ for some val **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** *failure*
  else return add $\{var/x\}$ to θ

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

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

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

Checking occurrence 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)

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) $\rightarrow$ *{x/Jane}*

Knows(y, Mother(y)) $\rightarrow$ *{x/Mother(John)}*

Knows(x, Elizabeth) $\rightarrow$ *fail*

- ???
- *Knows(x,Elizabeth)* means that anybody knows Elizabeth (universal quantifier is assumed there), so John knows Elizabeth.
- The problem is that both sentences contain variable **x** and hence cannot be unified.
- $\forall$*x Knows(x,Elizabeth)* is identical to $\forall$*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**.

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.

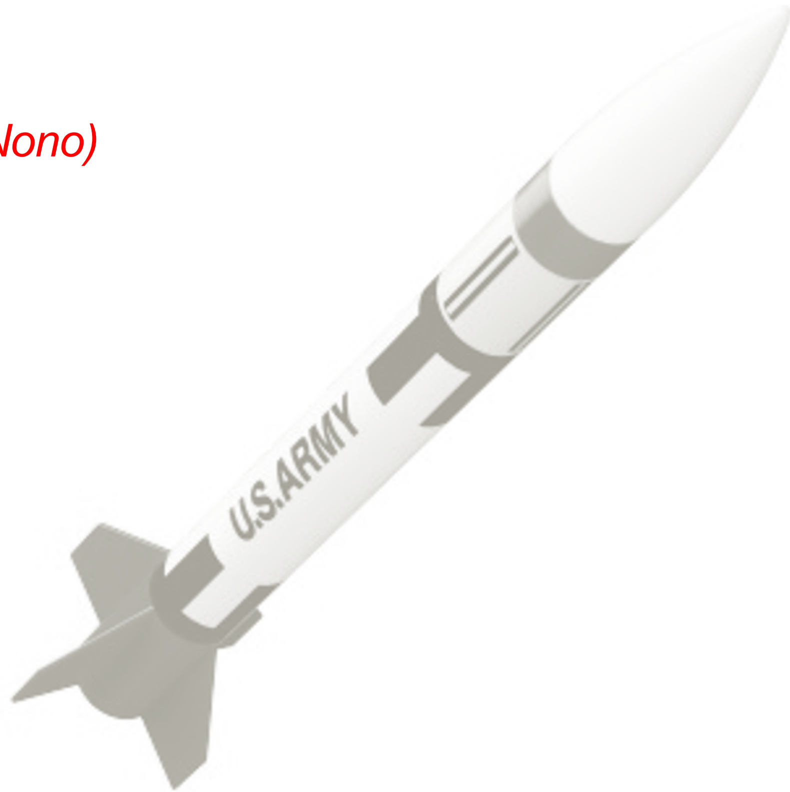
Enemy(x,America) ⇒ Hostile(x)

West is a US citizen …

American(West)

Nono is an enemy of USA …

Enemy(Nono,America)



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*
  inputs: KB, the knowledge base, a set of first-order definite clauses
    α, the query, an atomic sentence

  while *true* **do**
    $new \leftarrow \{\}$ // *The set of new sentences inferred on each iteration*
    for each $rule$ **in** KB **do**
      $(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES($rule$)
      for each θ such that SUBST($\theta, p_1 \wedge \ldots \wedge p_n$) = SUBST($\theta, p'_1 \wedge \ldots \wedge p'_n$)
        for some $p'_1, \ldots, p'_n$ in KB
        $q' \leftarrow$ SUBST(θ, q)
        if q' does not unify with some sentence already in KB or new **then**
          add q' to new
          $\phi \leftarrow$ UNIFY(q', α)
          if ϕ is not *failure* **then return** ϕ
    if $new = \{\}$ **then return** *false*
    add new to KB

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 the corresponding mgu

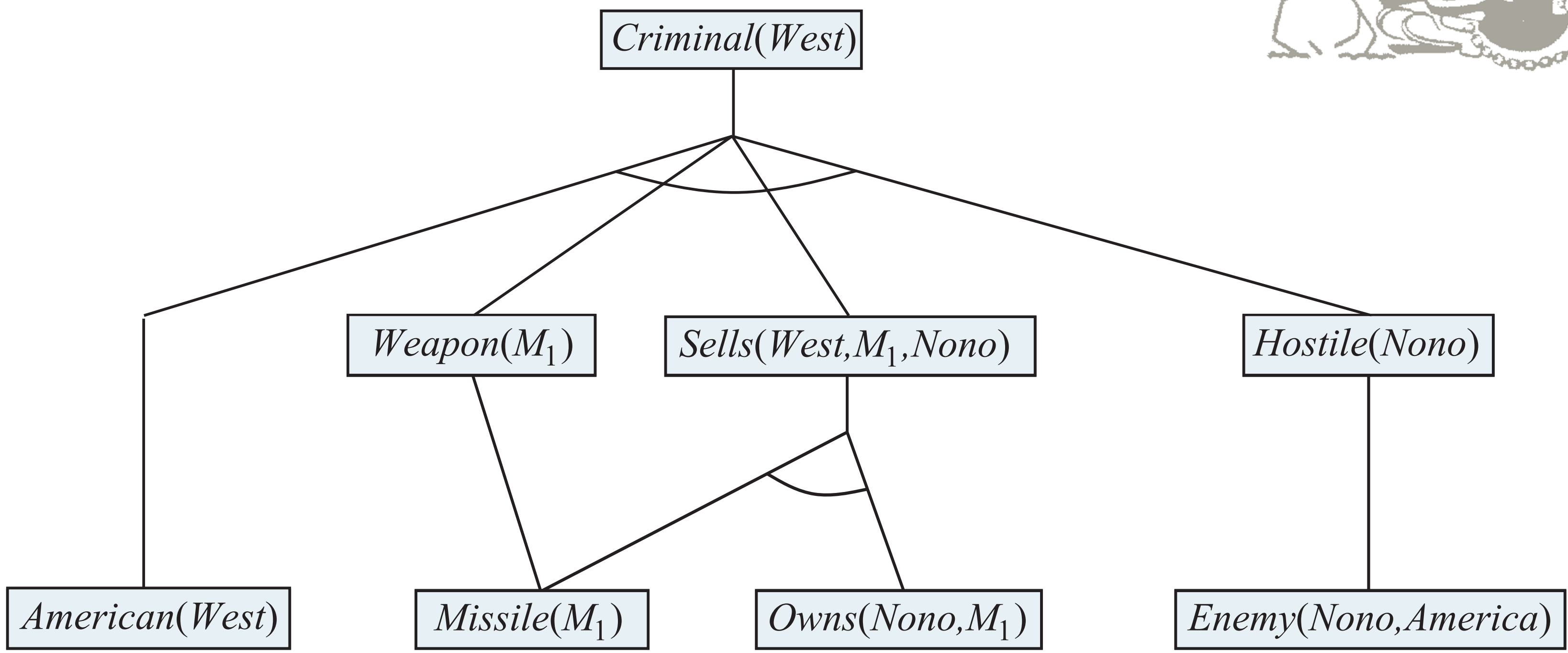
obtained sentences are placed to KB as theorems and the whole process is repeated

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

– Beware! If the sentence is not entailed by KB then 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) and Missile(M1) (from ∃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)



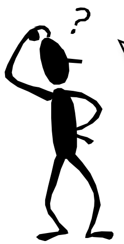
Forward chaining: pattern matching

> **for each** θ such that $\text{SUBST}(\theta, p_1 \wedge \ldots \wedge p_n) = \text{SUBST}(\theta, p'_1 \wedge \ldots \wedge p'_n)$
> 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

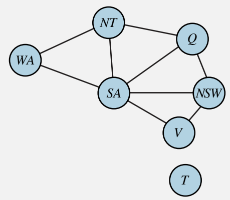
  Which order is better?
  - Start with less options (if there are two missiles while Nono owns many objects then alternative 2 is faster) – **recall the first-fail heuristic** from constraint satisfaction



!

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)



Example: *Missile(x) ⇒ Weapon(x)*

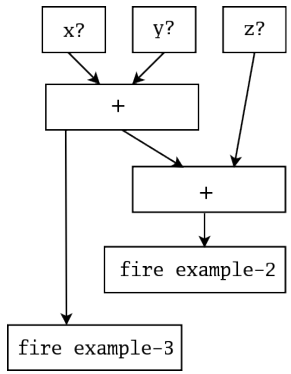
- during the iteration, the forward chaining algorithm infers that all known missiles are weapons
- during the second (and every other) iteration the algorithm deduces exactly the same information so KB is not updated

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

Magic(West)

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

TESTY

COMPUTERWORLD 2, 2007 17

Podnikoví správci pravidel
Máte-li možnost dosáhnout flexibility, výkonu a snadné údržby vašich firemních aplikací díky implementaci nákladově efektivního produktu, jako je JBoss Rules nebo Jess, naskýtá se otázka, v čem se tyto systémy liší od BRMS podnikové třídy. Několik rozdílů se mezi nimi přece jenom najde. Strana 18

Servery s architekturou x86
Servery postavené na architektuře x86, letitém a takřka nesmrtelném standardu, představují velmi efektivní řešení „hardwarového problému" pro mnoho firem a širokou škálu aplikací. Jejich výkon lze díky stále lepším komponentám škálovat až do nebetyčných výšin a pevně se zde zabydlely i 64bitové technologie. Strana 20

Řízení obchodních pravidel levně a jednoduše

JAMES OWEN

Uvážíme-li, že high-endové systémy pro řízení obchodních pravidel, BRMS (Business Rule Management System), vás vyjdou na zhruba 50 000 dolarů jen při zprovoznění a že roční údržba, provozní poplatky a profesionální služby mohou celkové náklady vytáhnout téměř až k půl milionu nebo více, mají organizace s těsnějším rozpočtem velmi dobrou motivaci poohlédnout se po alternativách. Dobré volby naštěstí existují – JBoss Rules a Jess představují solidní nástroje pro řízení pravidel a respektu hodný výkon za sympatickou cenu.

Dvěma z těch lepších BRMS nástrojů s nižší až nulovou cenou jsou Jess společnosti Sandia National Laboratories a JBoss Rules firmy JBoss, divize společnosti Red Hat. Stejně jako podnikové systémy jako Blaze Advisor společnosti Fair Isaac nebo JRules firmy ILog i Jess a JBoss odkrývají obchodní logiku komplexních javových aplikací jako sadu pravidel, která mohou být rychle a snadno změněna beze změn v základním Java kódu. Nicméně na rozdíl od těchto systémů třídy Enteprise ani Jess, ani JBoss Rules neposkytují uživatelsky přívětivá rozhraní (vizuální editor, diagramy toků či tabulkové GUI), jež dovolují běžným firemním/obchodním uživatelům stejně jako programátorům vkládat, měnit a mazat pravidla.

Na rozdíl od systémů Blaze Advisor a JRules postrádají Jess a JBoss Rules rovněž i plnohodnotný archiv pravidel (rule repository). Jess a JBoss Rules mohou být integrovány s CVS systémem pro kontrolu verzí, takové řešení však daleko zaostává za možnostmi řízení životního cyklu, granulární kontroly přístupů a rozsáhlého reportingu, poskytovanými sklady pravidel v podnikových produktech. Funkčně plně vybavený repository může být klíčem ke spolupráci mezi mnoha vývojáři a obchodními analytiky a bohaté možnosti reportingu mohou být nepostradatelnými prostředky pro ladění a optimalizaci.

Samozřejmě, přístup vycházející z filozofie open source, který reprezentují Jess a JBoss, má také své výhody. Jak Jess, tak JBoss Rules vyvíjejí vývojáři z celého světa, kteří nepřetržitě hledají a opravují chyby, navrhují nové fukce, píší nový kód a ve skutečnosti vlastně fungují jako neplacená inženýrská skupina starající se o tyto produkty. Váš IT personál by tak mohl – pod vedením uživatelské komunity Jess či JBoss Ruless nebo konzultantů třetích stran – vyvinout uživatelsky přívětivé tabulkové GUI, vizuální editor toků a dalších žádoucí prostředky, které si budete přát. Takové snahy ale budou klást značné nároky na personál, školení a investice po dobu několika měsíců až let, zatímco problém, který potřebujete řešit, existuje právě nyní.

V kostce se dá říci, že Jess a JBoss Rules jsou nejvhodnější pro menší projekty, kde archiv pravidel či rozsáhlé možnosti reportingu a ladění nepředstavují kritické požadavky a kde tvorba a údržba pravidel mohou být svěřeny jednomu nebo několika zasvěceným programátorům.

Sandia Labs Jess 7.0
Jess, systém společnosti Sandia Labs a Ernesta Friedmana-Hilla, byl, pokud je nám známo, první implementací na pravidlech založeného systému v Javě. Šlo o přímý výsledek portování dobře známých částí CLIPS (na jazyce C založeného rozhraní k Production Systems), projektu organizace NASA. Poté se začala objevovat řada high-endových systémů, jako je zmíněný JRules od ILog a Blaze Advisor od Fair Isaac. V následujících letech trval Fried-

- 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

```
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 {θ}
   q' ← Subst(θ, First(goals))
   for each r in KB where Standardize-Apart(r) = ( p1 ∧ ... ∧ pn ⇒ q)
           and θ' ← Unify(q, q') succeeds
     ans ← FOL-BC-Ask(KB, [p1, ..., pn|Rest(goals)], Compose(θ, θ')) ∪ 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

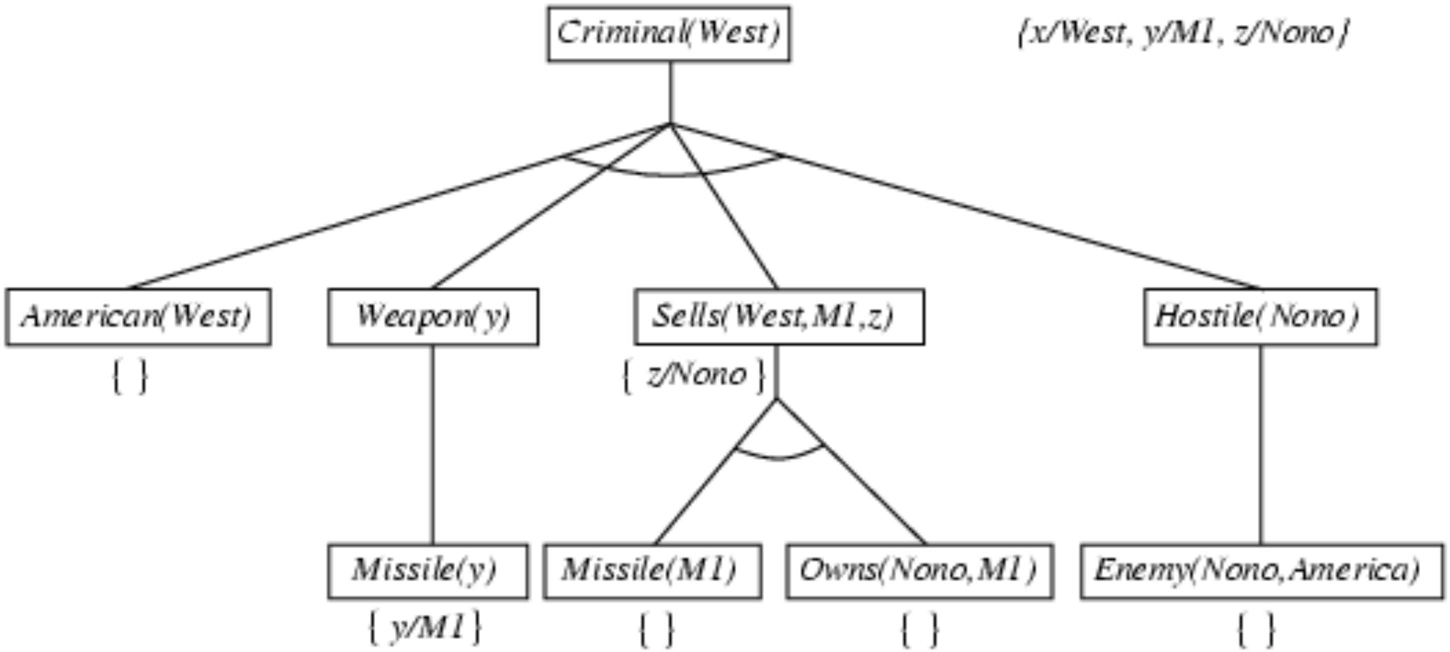
composition of substitutions
Subst(Compose(θ, θ`), p) = Subs(θ`, Subst(θ,p))

!

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) and Missile(M1) (from ∃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)



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

rule head

rule body

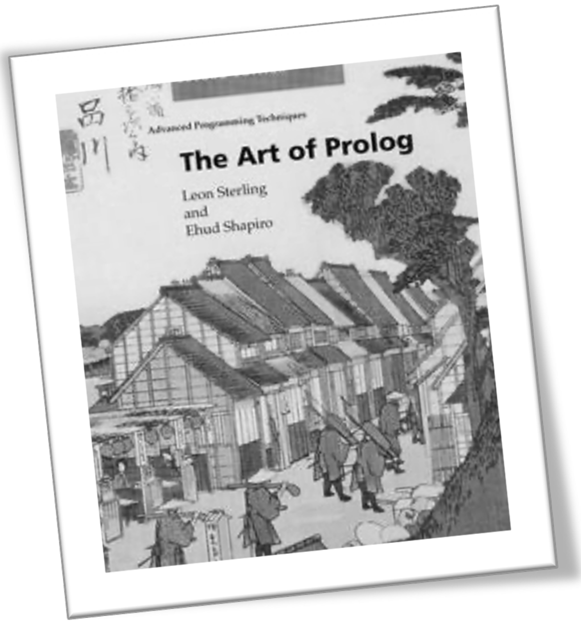
```
criminal(X) :-
    american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
owns(nono,m1).
missile(m1).
sells(west,X,nono) :-
    missile(X), owns(nono,X).
weapon(X) :-
    missile(X).
hostile(X) :-
    enemy(X,america).
american(west).
enemy(nono,america).

?- 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 "
  - $\neg$*Dead(x)* $\Rightarrow$ *Alive(x)* is not a definite clause!
    - *Alive(x)* $\vee$ *Dead(x)*
    - „Everyone is alive or dead"



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

- $\forall x\ [\forall y\ Animal(y) \Rightarrow Loves(x,y)] \Rightarrow [\exists y\ Loves(y,x)]$
- **remove implications**
  $\forall x\ [\neg\forall y\ \neg Animal(y) \vee Loves(x,y)] \vee [\exists y\ Loves(y,x)]$
- **put negation inside** ($\neg\forall x\ p \equiv \exists x\ \neg p$, $\neg\ \exists x\ p \equiv \forall x\ \neg p$)
  $\forall x\ [\exists y\ \neg(\neg Animal(y) \vee Loves(x,y))] \vee [\exists y\ Loves(y,x)]$
  $\forall x\ [\exists y\ \neg\neg Animal(y) \wedge \neg Loves(x,y)] \vee [\exists y\ Loves(y,x)]$
  $\forall x\ [\exists y\ Animal(y) \wedge \neg Loves(x,y)] \vee [\exists y\ Loves(y,x)]$
- **standardize variables**
  $\forall x\ [\exists y\ Animal(y) \wedge \neg Loves(x,y)] \vee [\exists z\ Loves(z,x)]$
- **Skolemize** (Skolem functions)
  $\forall x\ [Animal(F(x)) \wedge \neg Loves(x,F(x))] \vee [Loves(G(x),x)]$
- **remove universal quantifiers**
  $[Animal(F(x)) \wedge \neg Loves(x,F(x))] \vee [Loves(G(x),x)]$
- **distribute** $\vee$ **and** $\wedge$
  $[Animal(F(x)) \vee Loves(G(x),x)] \wedge [\neg Loves(x,F(x)) \vee Loves(G(x),x)]$

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 to:

- extend the binary resolution to more literals
- use **factoring** to remove redundant literals (those that can be unified together)

Example:

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

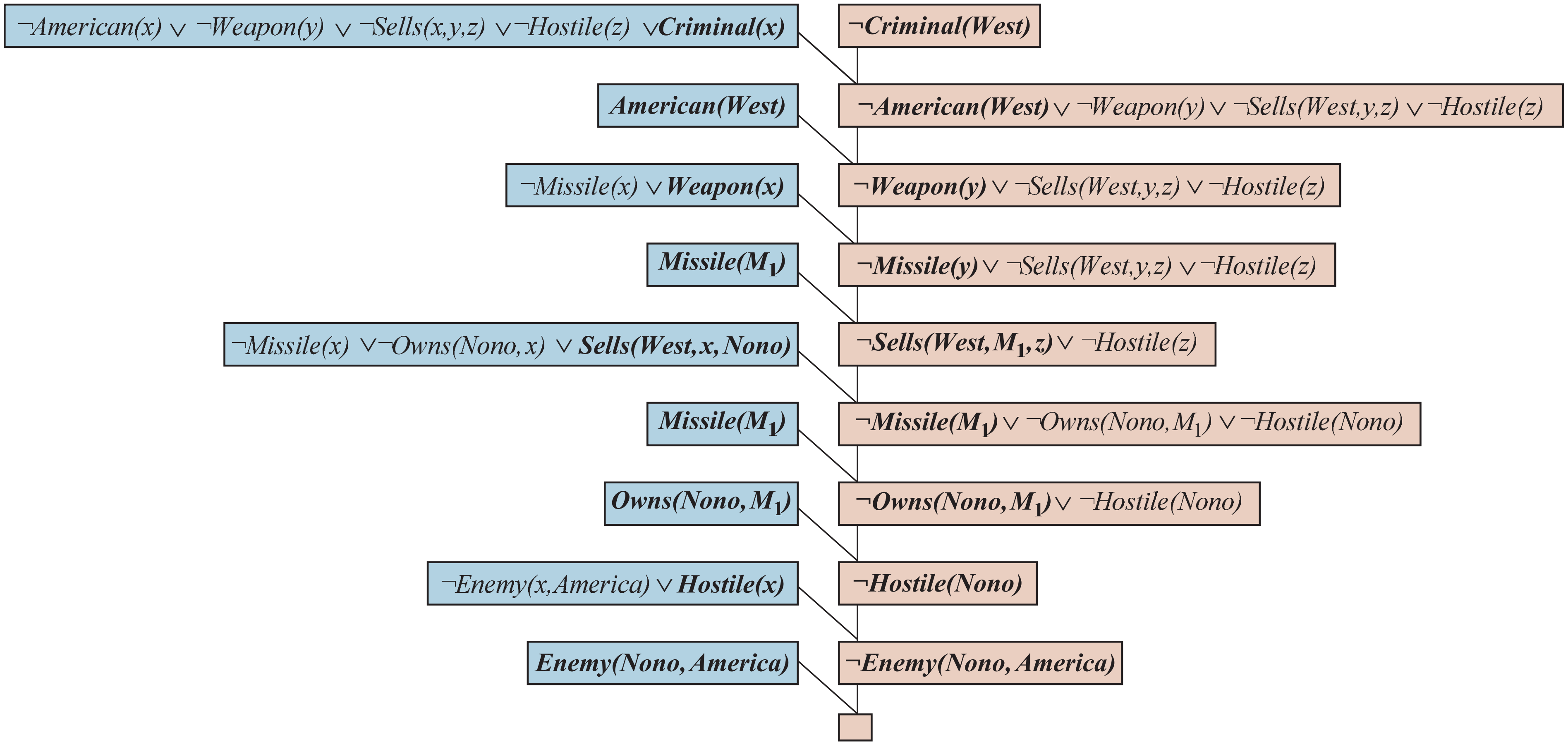
where θ = {u/G(x), v/x}

Query α for KB is answered by applying the resolution rule to CNF(KB $\wedge \neg\alpha$).

- If we obtain an empty clause, then KB $\wedge \neg\alpha$ is not satisfiable and hence KB $\models \alpha$.

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) and Missile(M1) (from ∃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

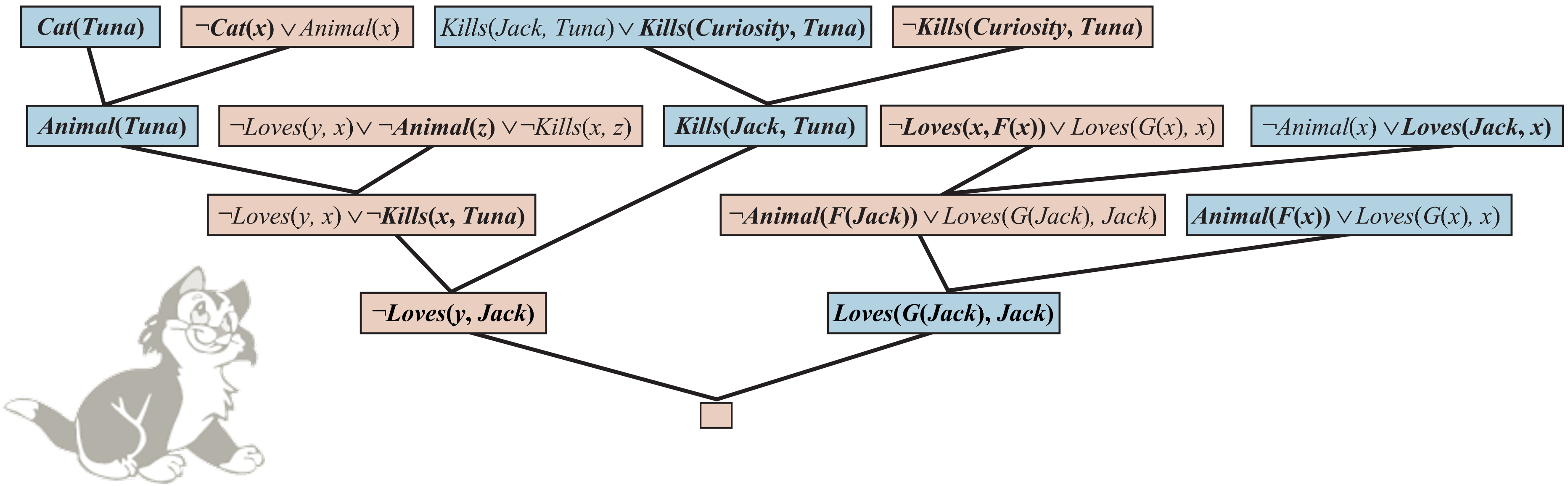
Everyone, who likes animals, is loved by somebody. Everyone, who kills animals, is loved by nobody. Jack likes all animals. Either Jack or Curiosity killed cat named Tuna. Cats are animals.

Did Curiosity kill Tuna?

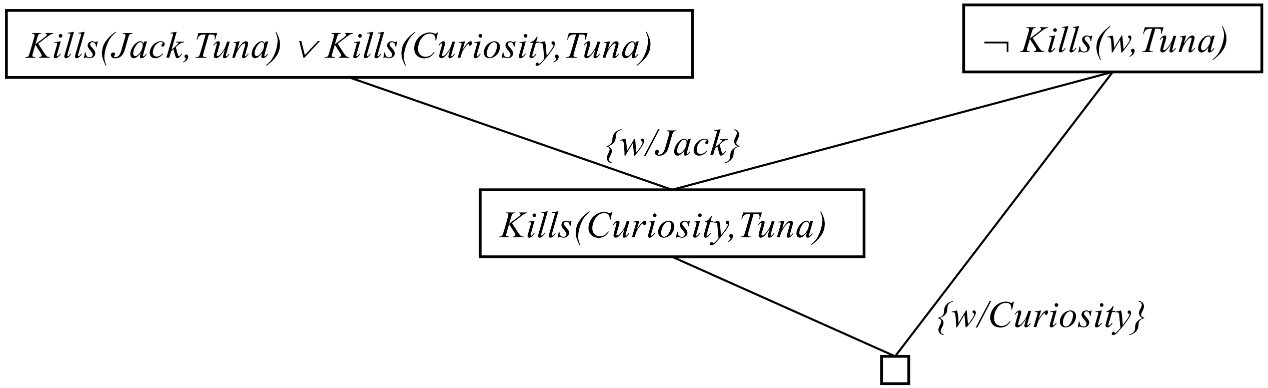
$\forall x\ [\forall y\ Animal(y) \Rightarrow Loves(x,y)] \Rightarrow [\exists y\ Loves(y,x)]$

$\forall x\ [\exists y\ Animal(y) \wedge Kills(x,y)] \Rightarrow [\forall z\ \neg Loves(z,x)]$
$\forall x\ Animal(x) \Rightarrow Loves(Jack,x)$
$Kills(Jack,Tuna) \vee Kills(Curiosity, Tuna)$
$Cat(Tuna)$
$\forall x\ Cat(x) \Rightarrow Animal(x)$
$\neg Kills(Curiosity,Tuna)$

$Animal(F(x)) \vee Loves(G(x),x)$
$\neg Loves(x,F(x)) \vee Loves(G(x),x)$
$\neg Animal(y) \vee \neg Kills(x,y) \vee \neg Loves(z,x)$
$\neg Animal(x) \vee Loves(Jack,x)$
$Kills(Jack,Tuna) \vee Kills(Curiosity,Tuna)$
$Cat(Tuna)$
$\neg Cat(x) \vee Animal(x)$
$\neg Kills(Curiosity,Tuna)$



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)*

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)
- **a 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) $\vee$ Q(B) to KB is not necessary

How can we handle **equalities** in the inference methods?

- **Axiomatizing equality**

  $\forall x\ \ x=x$

  $\forall x,y\ \ x=y \Rightarrow y=x$

  $\forall x,y,z\ \ x=y \wedge y=z \Rightarrow x=z$

  $\forall x,y\ \ x=y \Rightarrow P(x) \Leftrightarrow P(y)$

  $\forall x,y\ \ x=y \Rightarrow F(x) = F(y)$

  …

- **Special inference rules** such as **demodulation**

$$\frac{x=y \qquad m_1 \vee \cdots \vee m_n}{\text{sub}(x\theta, y\theta, m_1 \vee \cdots \vee m_n)}$$

where Unify(x, z) = θ, where appears somewhere in m_i, and sub(**x**, **y**, **m**) replaces **x** for **y** in **m**

$$\frac{\textit{Father(Father(x)) = PaternalGrandfather(x)} \qquad \textit{Birthdate(Father(Father(Bella)), 1926)}}{\textit{Birthdate(PaternalGrandfather(Bella), 1926}}$$

- **Extended unification**
  - handle equality directly by the unification algorithm

Department of Theoretical Computer Science and Mathematical Logic
bartak@ktiml.mff.cuni.cz