Umělá inteligence I

10

Roman Barták, KTIML

roman.bartak@mff.cuni.cz
http://ktiml.mff.cuni.cz/~bartak

Odvozování v PL

- Již umíme odvozovat ve výrokové logice a také umíme odvozovat v PL převodem na VL:
  - „uzemnění" výrazů (dosazení všech možných termů)
  - skolemizace pro odstranění existenčního kvantifikátoru
- Odvození převodem na VL ale není moc efektivní
  - generuje velké množství irelevantních tvrzení typu
    King(LeftLeg(John)) ∧ Greedy(LeftLeg(John)) ⇒ Evil(LeftLeg(John))
- Zkusíme odvozovat přímo v PL:
  - **liftování**
  - **unifikace**
  - **standardizace stranou**
- Zbytek už je nám známý:
  - **dopředné řetězení** (dedukční databáze, produkční systémy)
  - **zpětné řetězení** (logické programování)
  - **rezoluce** (dokazování vět)

Odvozování v PL

- Upravíme odvozovací pravidla pro práci v PL:
  - **liftování** – uděláme jen takové substituce, které potřebujeme pro odvozování
  - **zobecněné (liftované) pravidlo Modus Ponens**

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

kde θ je substituce taková, že $Subst(\theta,p_i) = Subst(\theta,q_i)$

(pro **definitní klauzule** s právě jedním pozitivním literálem - **pravidla**)

  - potřebujeme najít substituci, která způsobí, že dva logické výrazy budou identické (po substituci)
    - King(John) $\wedge$ Greedy(y) King(x) $\wedge$ Greedy(x)
    - substituce {x/John, y/John }

Unifikace

- **Jak najít substituci θ takovou, že dva logické výrazy p a q po její aplikaci vypadají identicky?**
  - Unify(p,q) = θ , kde $Subst(\theta,p) = Subst(\theta,q)$

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}

  - **Co když ale existuje více takových substitucí?**

    Knows(John,x) Knows(y,z),

    ↳ θ = {y/John, x/z } nebo θ = {y/John, x/John, z/John}
    - První **substituce je obecnější** než druhá substituce (druhý výsledek lze získat dodatečnou aplikací substituce {z/John}).
  - Existuje jednoznačná (až na přejmenování proměnných) substituce, která je obecnější než všechny ostatní – **nejobecnější unifikátor** (mgu – most general unifier).

Hledáme unifikátor

```
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
```

výrazy se procházejí současně použitím rekurze a staví se mgu dokud se struktury „nevyprázdní" resp. nezjistí se, že se liší

složené termy musí mít stejný funkční symbol a unifikovatelné argumenty

seznamy se řeší odděleně, aby nevzniknul cyklus při jejich reprezentaci strukturou .(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
```

kontrola výskytu proměnné *var* v termu *x*
- *x* a *f(x)* nejsou unifikovatelné
- vede ke kvadratické složitosti algoritmu
- existuji i lineární algoritmy
- někdy se neprovádí (Prolog)

Standardizace stranou

- Představme si, že **dotaz** na bázi znalostí je Knows(John, x)
- Odpovědi můžeme získat **nalezením tvrzení** v bázi znalostí, která jsou **unifikovatelná s dotazem**:

  Knows(John, Jane) → {x/Jane}
  Knows(y, Mother(y)) → {x/Mother(John)}
  Knows(x, Elizabeth) → fail

  - ???
  - Knows(x,Elizabeth) v bázi znalostí znamená, že každý zná Elizabeth (předpokládáme univerzální kvantifikátor), tedy i John.
  - Problém je, že oba výrazy obsahují proměnnou x a nelze je tedy unifikovat.
  - ∀x Knows(x,Elizabeth) je ale to samé jako ∀y Knows(y,Elizabeth)
  - Před každým použitím tvrzení z báze znalostí v něm přejmenujeme všechny proměnné na úplně nové proměnné, které se ještě nikde nepoužily – **standardizace stranou**.

Příklad

- Podle práva v USA je zločincem každý Američan, který prodává zbraně nepřátelským státům. Země Nono je nepřítelem USA vlastnícím rakety a všechny tyto rakety jim byly prodány plukovníkem Westem, který je Američanem.
- Dokažte, že West je zločinec.

... je zločincem každý Američan, který prodává zbraně nepřátelským státům:
American(x) ∧ Weapon(y) ∧ Sells(x,y,z) ∧ Hostile(z) ⇒ Criminal(x)

Nono … vlastní rakety, tj. ∃x Owns(Nono,x) ∧ Missile(x):
Owns(Nono,M_1) and Missile(M_1)

… všechny tyto rakety jim byly prodány plukovníkem Westem
Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)

Rakety jsou zbraně.
Missile(x) ⇒ Weapon(x)

Nepřítel USA je „nepřátelský stát".
Enemy(x,America) ⇒ Hostile(x)

West je Američan …
American(West)

Země Nono je nepřítelem USA …
Enemy(Nono,America)

Odvozovací techniky

Všechna tvrzení v příkladě mají tvar definitních klauzulí a nemáme tam žádné funkční symboly.

Pro řešení příkladu můžeme použít:

- **dopředné řetězení**
  - pomocí Modus Ponens odvodíme z daných faktů všechna možná tvrzení
  - typický způsob odvozování v deduktivních databázích (Datalog) a produkčních systémech
- **zpětné řetězení**
  - půjdeme naopak od dotazu Criminal(West) a budeme hledat fakta, která ho potvrzují
  - typický způsob odvozování v logickém programování

Dopředné řetězení

algoritmus

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

repeat until *new* is empty
- *new* ← { }
- **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*

vezmeme pravidlo z databáze, kde přejmenujeme proměnné (standardizace stranou)

odvodíme všechny důsledky, které ještě v KB nejsou

pokud už jsme odvodili tvrzení unifikatelné s dotazem, potom vrátíme příslušnou substituci

získaná tvrzení přidáme do báze znalostí a vše opakujeme dokud přidáváme další tvrzení

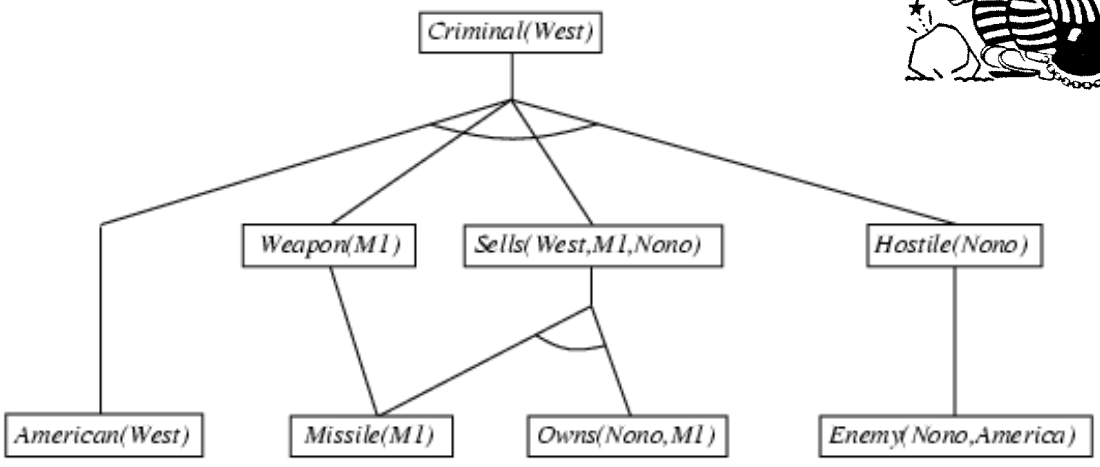
Dopředné řetězení je **korektní** a **úplný** algoritmus pro nalezení odvození.

- Pozor! Pokud odvození neexistuje, nemusí skončit (máme-li v jazyce funkční symboly).

Dopředné řetězení

příklad

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)



Dopředné řetězení

hledání vzorů

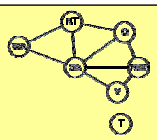
for each θ such that $(p_1 \wedge \ldots \wedge p_n)\theta = (p'_1 \wedge \ldots \wedge p'_n)\theta$
for some $p'_1, \ldots, p'_n$ in KB

- Jak najdeme (rychle) množinu faktů p`$_1$,..., p`$_n$, aby šla unifikovat s tělem pravidla?
  - tzv. **hledání vzorů** (pattern matching)
  - Příklad 1: *Missile(x) ⇒ Weapon(x)*
    - můžeme **indexovat** množinu **faktů podle jména** predikátu, čímž se vyhneme pokusům typu *Unify(Missile(x),Enemy(Nono, America)*
  - Příklad 2: *Missile(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)*
    1. můžeme hledat objekty, které vlastní Nono, a jsou to rakety ...
    2. nebo můžeme hledat rakety a zjišťovat, zda je vlastní Nono

    Co je lepší?
    - Začni tam, kde je méně možností (pokud existují dvě rakety, ale Nono vlastní hodně věcí, je lepší alternativa 2) – **připomeňme pravidlo prvního neúspěchu** z omezujících podmínek

Hledání vzorů je NP-úplný problém.

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)



Dopředné řetězení

inkrementálně

- Příklad: *Missile(x) ⇒ Weapon(x)*
  - při první iteraci dopředného řetězení odvodí, že všechny známé rakety jsou zbraně
  - při druhé (a každé další) iteraci odvodí to samé takže bázi znalostí už nemění
- Kdy má cenu pravidlo znova zkoušet?
  - pokud se v KB objeví nový fakt z těla pravidla
- **Inkrementální dopředné řetězení**
  - pravidlo se spustí v iteraci t pouze pokud byl v iteraci t-1 odvozen nový fakt unifikovatelný s faktem v těle pravidla
  - když do KB přidáme nový fakt, podíváme se na všechna pravidla, která obsahují s ním unifikovatelný výraz ve svém těle
  - **Rete algoritmus**
    - předzpracuje pravidla do podoby sítě závislostí, kde rychleji vyhledá, jaké pravidlo se má zkusit po přidání nového faktu

Dopředné řetězení

magická množina

- Dopředné řetězení odvozuje všechna fakta, i když nejsou relevantní pro zjištění odpovědi.
  - jedním z řešení je **zpětné řetězení**
  - jiná možnost je upravit pravidla tak, aby umožňovaly pouze relevantní navázaní proměnných z tzv. **magické množiny**

Příklad: dotaz na Criminal(West)

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

$$Magic(West)$$

- Magickou množinu lze získat zpětným průchodem přes pravidla.

Produkční systémy

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 skákavat 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.

Sandia Labs Jess 7.0

- založené na rete algoritmu
- XCON (R1)
  - konfigurace počítačů DEC
- OPS-5
  - jazyk založený na dopředném řetězení
- CLIPS
  - nástroj pro tvorbu expertním systémů vyvinutý v NASA
- Jess, JBoss Rules,...
  - řízení obchodních pravidel

Zpětné řetězení
algoritmus

```
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
```

vezmeme první z cílů a aplikujeme na něj dosud nalezenou substituci

najdeme pravidlo, jehož hlava je unifikovatelná s cílem

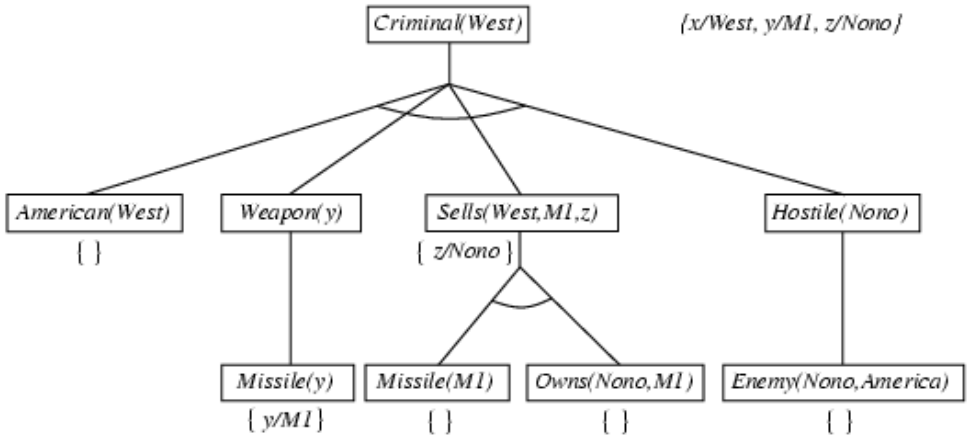
tělo pravidla přidáme mezi cíle, upravíme substituci a rekurzivně pokračujeme v redukci cílů dokud nejsou prázdné

skládání substitucí
Subst(Compose(θ, θ'), p) = Subs(θ', Subst(θ,p))

Algoritmus FOL-BC-Ask metodou **prohledávání do hloubky** hledá **všechna řešení** (všechny substituce) daného dotazu.
Pro běh stačí **lineární prostor** (v délce důkazu).
Na rozdíl od dopředného řetězení **není úplné**, kvůli opakovaným stavům (vytvoří se stejný cíl).

Zpětné řetězení
příklad

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)



Logické programování

- Zpětné řetězení v podstatě odpovídá **logickému programování** (Prolog).

hlava pravidla

tělo pravidla

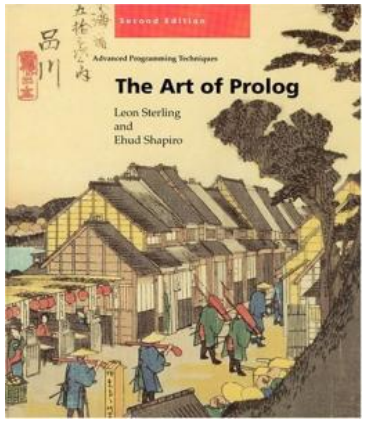
```
criminal(X) :-
    american(X), weapon(Y), sells(X,Y,Z), hostile(Z).
owns(nono,m1).
missile(m1).
sells(west,X,nono) :-
    missile(X), owns(nono,X).
weapon(X) :-
    missile(X).
hostile(X) :-
    enemy(X,america).
american(west).
enemy(nono,america).

?- 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.
```


Logické programování

vlastnosti

- **pevně daný výpočtový mechanismus**
  - podcíle se řeší zleva doprava
  - pravidla se berou v pořadí shora dolu
- vrací **jediné řešení**, další řešení na žádost
  - možnost zacyklení (`brother(X,Y) :- brother(Y,X)`)
- obsahuje **aritmetiku**
  - `X is 1+2.`
  - (aritmeticky) vyhodnotí výraz nalevo a unifikuje s výrazem napravo
- **rovnost** slouží pro explicitní přístup k unifikaci
  - `1+Y = 3.`
  - možnost použít omezující podmínky (CLP – Constraint Logic Programming)
- **negace jako neúspěch** (negation as failure)
  - `alive(X) :- not dead(X).`
    - „Každý je živý, pokud nelze dokázat, že je mrtvý"
  - $\neg Dead(x) \Rightarrow Alive(x)$ není definitní klauzule!
    - $Alive(x) \vee Dead(x)$
    - „Každý je živý nebo mrtvý"



Rezoluční metoda

konjunktivní normální forma

Odvozovací metoda pro formule v **konjunktivní normálně formě**.

- $\forall x\ [\forall y\ Animal(y) \Rightarrow Loves(x,y)] \Rightarrow [\exists y\ Loves(y,x)]$
- **odstraníme implikace**
  $\forall x\ [\neg\forall y\ \neg Animal(y) \vee Loves(x,y)] \vee [\exists y\ Loves(y,x)]$
- **negaci posuneme dovnitř** ($\neg\forall x\ p \equiv \exists x\ \neg p$, $\neg\ \exists x\ p \equiv \forall x\ \neg p$)
  $\forall x\ [\exists y\ \neg(\neg Animal(y) \vee Loves(x,y))] \vee [\exists y\ Loves(y,x)]$
  $\forall x\ [\exists y\ \neg\neg Animal(y) \wedge \neg Loves(x,y)] \vee [\exists y\ Loves(y,x)]$
  $\forall x\ [\exists y\ Animal(y) \wedge \neg Loves(x,y)] \vee [\exists y\ Loves(y,x)]$
- **standardizace proměnných**
  $\forall x\ [\exists y\ Animal(y) \wedge \neg Loves(x,y)] \vee [\exists z\ Loves(z,x)]$
- **Skolemizace** (Skolemovy funkce)
  $\forall x\ [Animal(F(x)) \wedge \neg Loves(x,F(x))] \vee [Loves(G(x),x)]$
- **odstranění univerzálních kvantifikátorů**
  $[Animal(F(x)) \wedge \neg Loves(x,F(x))] \vee [Loves(G(x),x)]$
- **distribuce $\vee$ a $\wedge$**
  $[Animal(F(x)) \vee Loves(G(x),x)] \wedge [\neg Loves(x,F(x)) \vee Loves(G(x),x)]$

Rezoluční metoda

odvozovací pravidlo

- Liftovaná verze resolučního pravidla pro výrokovou logiku:

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

  kde Unify(ℓ_i, $\neg m_j$) = θ.
- U klauzulí se předpokládá standardizace stranou, aby nesdílely žádné proměnné.
- Pro úplnost je potřeba buď:
  - rozšířit binární rezoluci na více literálů
  - použít **factoring** pro odstranění redundantních literálů (ty, které jsou unifikovatelné)

Příklad:

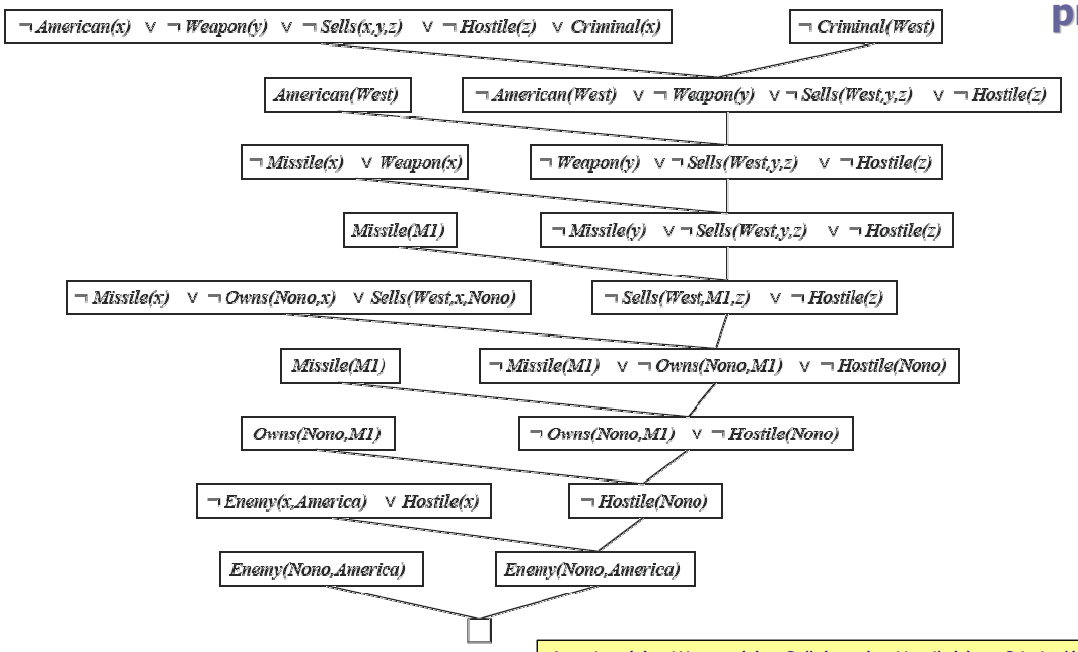
$$\frac{[Animal(F(x)) \vee Loves(G(x),x)], \qquad [\neg Loves(u,v) \vee \neg Kills(u,v)]}{[Animal(F(x)) \vee \neg Kills(G(x),x)]}$$

kde θ = {u/G(x), v/x}

- Dotaz α nad bází znalostí KB řešíme aplikací rezolučních kroků na formuli CNF(KB $\wedge \neg\alpha$).
  - Pokud vede k prázdné formuli, je KB $\wedge \neg\alpha$ nesplnitelné a tedy KB $\models \alpha$.
- Jedná se o **korektní** a **úplnou** odvozovací metodu pro predikátovou logiku

Rezoluční metoda

příklad



Resoluční metoda aplikovaná na definitní klauzule je vlastně aplikace **zpětného řetězení**, které určuje jaké rezoluční pravidlo se má použít.

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)

Rezoluční metoda

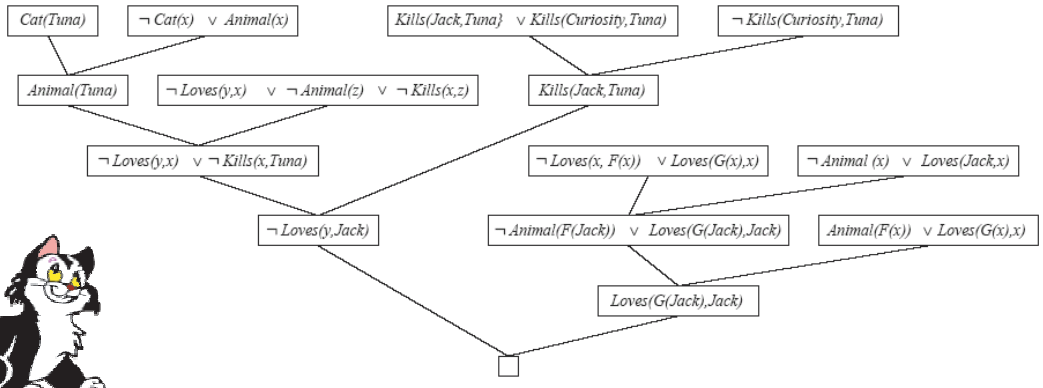
složitější příklad

- Každého, kdo má rad všechna zvířata, má někdo rád. Každého, kdo zabíjí zvířata, nemá rád nikdo. Jack má rád všechny zvířata. Buď Jack nebo Curiosity zabili kočku jménem Tuna. Každá kočka je zvíře.
- Zabila Curiosity Tunu?

∀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)

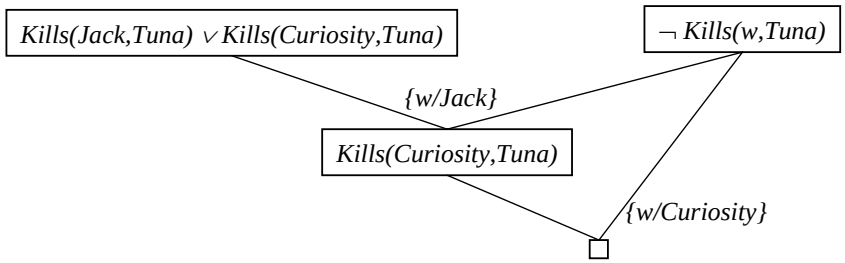
Animal(F(x)) ∨ Loves(G(x),x)
¬ Loves(x,F(x)) ∨ Loves(G(x),x)
¬ Animal(y) ∨ ¬Kills(x,y) ∨ ¬ Loves(z,x)
¬ Animal(x) ∨ Loves(Jack,x)
Kills(Jack,Tuna) ∨ Kills(Curiosity,Tuna)
Cat(Tuna)
¬ Cat(x) ∨ Animal(x)
¬ Kills(Curiosity,Tuna)



Rezoluční metoda

nekonstruktivní důkaz

- Co když bude dotaz formulován jako **„Kdo zabil Tunu?"**



- Odpověď tedy je **„Ano, někdo zabil Tunu"**.
- Můžeme do dotazu přidat speciální **literál pro získání odpovědi**.
  - ¬ *Kills(w,Tuna)* ∨ ***Answer(w)***
  - předchozí nekonstruktivní důkaz by skončil u *Answer(Curiosity)* ∨ *Answer(Jack)*
  - je tedy vynucen důkaz se strukturou podobnou pro dotaz ¬ *Kills(Curiosity,Tuna)*

Rezoluční metoda

strategie

Jak **efektivně** hledat rezoluční důkazy?

- **jednotková rezoluce**
  - snahou je získat prázdnou klauzuli, proto je dobré, pokud se nově odvozené klauzule zkracují
  - preferujeme rezoluční krok, kde je jedna z klauzulí jednotková (obsahuje jediný literál)
  - obecně se nelze omezit pouze na jednotkové rezoluce, ale pro Hornovské klauzule je to úplná metoda (vlastně dopředné řetězení)
- **množina podpor**
  - vybrané klauzule, z nichž alespoň jedna se vždy účastní rezoluce (výsledek rezoluce se přidává do množiny podpor)
  - počáteční množina podpor typicky obsahuje negovaný dotaz
- **vstupní rezoluce**
  - každého rezolučního kroku se účastní alespoň jedna klauzule ze vstupu – počáteční KB nebo dotaz
  - není to úplná metoda
- **subsumce**
  - eliminuje klauzule, které jsou zahrnuty jinými klauzulemi
  - máme-li v bázi znalostí P(x), nemá cenu přidávat P(A) ani P(A) ∨ Q(B)