

# RÉSOLUTION & PROLOG

## Raisonnement automatique en LPPO

LRC - 2025/2026

Gauvain Bourgne

# Résumé des épisodes précédents

## Logique classique

- Logique propositionnelle
- Logique des prédicats du premier ordre (LPPO)

## Fragment : Ontologies

- Logiques de Description ( $\mathcal{ALC}$ ,  $\mathcal{FL}^-$ ,  $\mathcal{SHOIN}$ )

## Extensions ?

- Dans la suite (cours 7 à 10) : Modalités (logiques modales)

## Méthode des tableaux

- Satisfiabilité pour formule propositionnelle, LPPO,  $\mathcal{ALC}$
- Sera aussi utilisé pour logiques modales (satisfiabilité et vérification de formules)

Automatisation : Lotrec, FacTT

## Systèmes d'inférences

- Logique classique : Hilbert (prop), Système de déduction naturelle (prop ou LPPO)

pas très automatisable

# Un autre système de preuve : la Résolution

## Résolution

### Une méthode

- simple
  - Pas d'axiomes
  - Une seule règle d'inférence généralisant le modus ponens
- générale (applicable à LPPO)
- adaptée à la déduction automatique (méthodes correctes et complètes)
- mais qui nécessite une normalisation des formules en une théorie clausale (forme normale conjonctive skolémisée)

Mécanisme à la base de la programmation logique (via interprétation de clauses en règles)

# Plan

- 1 La règle de résolution
  - En logique propositionnelle
  - En LPPO
- 2 Unification en LPPO
- 3 Raisonnement par résolution
  - Mise sous forme clausale
  - Preuves par résolution
  - SLD-résolution
- 4 Programmation logique
  - Programmes logiques
  - Prolog

# La règle de résolution

# Généraliser le Modus Ponens

## Règle d'inférence

Modus ponens classique

$$\frac{A \rightarrow B \quad A}{B}$$

# Généraliser le Modus Ponens

## Règle d'inférence

Si on n'est pas sûr de la prémisse

$$\frac{A \rightarrow B \quad A \vee C}{???}$$

# Généraliser le Modus Ponens

## Règle d'inférence

Réécriture  $D = \neg C$

$$\frac{A \rightarrow B \quad D \rightarrow A}{D \rightarrow B}$$

# Généraliser le Modus Ponens

## Règle d'inférence

Sous forme disjonctive

$$\frac{\neg A \vee B \quad \neg D \vee A}{\neg D \vee B}$$

# Généraliser le Modus Ponens

## Règle d'inférence

Sous forme disjonctive (avec  $C = \neg D$ )

$$\frac{\neg A \vee B \quad A \vee C}{B \vee C}$$

# Généraliser le Modus Ponens

## Règle d'inférence

Avec  $B = L_1 \vee \dots \vee L_n$  et  $C = L'_1 \vee \dots \vee L'_k$  où chaque  $L_i$  ou  $L'_i$  est un littéral ( $p$  ou  $\neg p$ )

$$\frac{\neg A \vee L_1 \vee \dots \vee L_n \quad A \vee L'_1 \vee \dots \vee L'_k}{L_1 \vee \dots \vee L_n \vee L'_1 \vee \dots \vee L'_k}$$

# Généraliser le Modus Ponens

## Règle d'inférence

$$\frac{C_1 : \neg A \vee L_1 \vee \dots \vee L_n \quad C_2 : A \vee L'_1 \vee \dots \vee L'_k}{R : L_1 \vee \dots \vee L_n \vee L'_1 \vee \dots \vee L'_k}$$

C'est la règle de résolution en logique propositionnelle.

- $C_1$ ,  $C_2$  sont appelées les prémisses de cette résolution
- $A$  est le littéral sur lequel on fait la résolution
- $R$  est appelé la résolvante de cette résolution. On note  $R = \text{Res}(C_1, C_2, A)$

# Généraliser le Modus Ponens

## Règle d'inférence

Simplification de  $R$  si  $L_1 = L'_1 = C_1$

$$\frac{C_1 : \neg A \vee C_1 \vee \dots \vee L_n \quad C_2 : A \vee C_1 \vee \dots \vee L'_k}{R : C_1 \vee \dots \vee L_n \vee \dots \vee L'_k}$$

C'est la règle de résolution en logique propositionnelle.

- $C_1, C_2$  sont appelées les prémisses de cette résolution
- $A$  est le littéral sur lequel on fait la résolution
- $R$  est appelé la résolvante de cette résolution. On note  $R = \text{Res}(C_1, C_2, A)$

# Résolution en logique propositionnelle

## La règle de résolution

Soit  $p$  une variable propositionnelle et  $C_1 : l_1 \vee \dots \vee l_n$ ,  $C_2 : l'_1 \vee \dots \vee l'_k$  deux clauses (disjonctions de littéraux). Résolution de  $p \vee C_1$  et  $\neg p \vee C_2$  sur  $p$  :

$$\frac{p \vee C_1 \quad \neg p \vee C_2}{C_1 \vee C_2}$$

écriture linéaire :  $\text{Res}(p \vee C_1, \neg p \vee C_2, p) = C_1 \vee C_2$

## Exemples

$a \vee b$  et  $\neg a \vee b$

$\neg a \vee b$  et  $a \vee \neg b$

$\neg a \vee b \vee c$  et  $\neg b$

# Résolution en logique propositionnelle

## La règle de résolution

Soit  $p$  une variable propositionnelle et  $C_1 : l_1 \vee \dots \vee l_n$ ,  $C_2 : l'_1 \vee \dots \vee l'_k$  deux clauses (disjonctions de littéraux). Résolution de  $p \vee C_1$  et  $\neg p \vee C_2$  sur  $p$  :

$$\frac{p \vee C_1 \quad \neg p \vee C_2}{C_1 \vee C_2}$$

écriture linéaire :  $\text{Res}(p \vee C_1, \neg p \vee C_2, p) = C_1 \vee C_2$

## Exemples

$a \vee b$  et  $\neg a \vee b$   
 $b \vee b = b$

$\neg a \vee b$  et  $a \vee \neg b$   
 sur  $a : b \vee \neg b = \top$

$\neg a \vee b \vee c$  et  $\neg b$   
 $\neg a \vee c$

# Cas de la LPPO

## Syllogisme

Tout les hommes sont mortels

Socrate est un homme

⇒ Socrate est mortel

# Cas de la LPPO

## Syllogisme

Tout les hommes sont mortels

$R : \forall X (H(X) \rightarrow M(X))$

Socrate est un homme

$F : H(s)$

$\Rightarrow$  Socrate est mortel  
 $C : M(s)$

# Cas de la LPPO

## Syllogisme

Tout les hommes sont mortels

$R : \forall X (H(X) \rightarrow M(X))$

Socrate est un homme

$F : H(s)$

$\Rightarrow$  Socrate est mortel  
 $C : M(s)$

## Preuve LPPO

On mobilise deux règles d'inférences :

- Instantiation de  $R$  avec  $\sigma = \{X \mapsto s\}$   
donne  $\sigma(R) = H(s) \rightarrow M(s)$
- Modus ponens simple entre  $\sigma(R)$  et  $F$   
donne  $C : M(s)$

# Cas de la LPPO

## Résolution en LPPO

Littéral : une formule atomique ( $P(t_1, \dots, t_k)$  où  $P$  prédicat d'arité  $k$ ) ou sa négation

Clauses : disjonction universellement quantifiées de littéraux (peut s'écrire comme un ensemble de littéraux)

Ici :  $C_R : \neg H(X) \vee M(X)$  et  $C_F : H(s)$ .

On combine les deux étapes : trouver  $\sigma$  tel que  $\sigma(C_R)$  et  $\sigma(C_F)$  ait deux littéraux complémentaires, puis appliquer la résolution comme en propositionnel sur  $\sigma(C_R)$  et  $\sigma(C_F)$ .

$$\frac{\neg H(X) \vee M(X) \quad H(s)}{M(s)} \{X \mapsto s\}$$

# Résolution en logique propositionnelle

## La règle de résolution

Soit  $L_{pos} = P(t_1, \dots, t_k)$  et  $L_{neg} = \neg P(t'_1, \dots, t'_k)$  deux littéraux sur le même prédicats  $P$  d'arité  $k$  tels que  $L_{pos}$  et  $\neg L_{neg}$  ( $P(t'_1, \dots, t'_k)$ ) sont unifiables ( $\exists \sigma$  telle que  $\sigma(L_{pos}) \equiv \sigma(\neg L_{neg})$ ).

Résolution de  $L_1 \vee C_1$  et  $L_2 \vee C_2$  sur  $\sigma$  :

$$\frac{L_{pos} \vee C_1 \quad L_{neg} \vee C_2}{\sigma(C_1) \vee \sigma(C_2)} \sigma$$

écriture linéaire :  $Res(L_{pos} \vee C_1, L_{neg} \vee C_2, \sigma) = \sigma(C_1) \vee \sigma(C_2)$

## Exemples

(1) :  $\neg P(X) \vee R(X)$  et  $P(f(a)) \vee P(b)$

(2) :  $\neg P(f(X), f(a)) \vee R(X, b)$  et  $P(f(f(X)), Y) \vee R(f(a), Y)$

# Résolution en logique propositionnelle

## La règle de résolution

Soit  $L_{pos} = P(t_1, \dots, t_k)$  et  $L_{neg} = \neg P(t'_1, \dots, t'_k)$  deux littéraux sur le même prédicats  $P$  d'arité  $k$  tels que  $L_{pos}$  et  $\neg L_{neg}$  ( $P(t'_1, \dots, t'_k)$ ) sont unifiables ( $\exists \sigma$  telle que  $\sigma(L_{pos}) \equiv \sigma(\neg L_{neg})$ ).

Résolution de  $L_1 \vee C_1$  et  $L_2 \vee C_2$  sur  $\sigma$  :

$$\frac{L_{pos} \vee C_1 \quad L_{neg} \vee C_2}{\sigma(C_1) \vee \sigma(C_2)} \sigma$$

écriture linéaire :  $Res(L_{pos} \vee C_1, L_{neg} \vee C_2, \sigma) = \sigma(C_1) \vee \sigma(C_2)$

## Exemples

(1) :  $\neg P(X) \vee R(X)$  et  $P(f(a)) \vee P(b)$

Res :  $R(f(a)) \vee P(b)$  ( $X \mapsto f(a)$ ) et  $R(b) \vee P(f(a))$  ( $X \mapsto b$ )

(2) :  $\neg P(f(X), f(a)) \vee R(X, b)$  et  $P(f(f(X)), Y) \vee R(f(a), Y)$

# Résolution en logique propositionnelle

## La règle de résolution

Soit  $L_{pos} = P(t_1, \dots, t_k)$  et  $L_{neg} = \neg P(t'_1, \dots, t'_k)$  deux littéraux sur le même prédicats  $P$  d'arité  $k$  tels que  $L_{pos}$  et  $\neg L_{neg}$  ( $P(t'_1, \dots, t'_k)$ ) sont unifiables ( $\exists \sigma$  telle que  $\sigma(L_{pos}) \equiv \sigma(\neg L_{neg})$ ).  
Résolution de  $L_1 \vee C_1$  et  $L_2 \vee C_2$  sur  $\sigma$  :

$$\frac{L_{pos} \vee C_1 \quad L_{neg} \vee C_2}{\sigma(C_1) \vee \sigma(C_2)} \sigma$$

écriture linéaire :  $Res(L_{pos} \vee C_1, L_{neg} \vee C_2, \sigma) = \sigma(C_1) \vee \sigma(C_2)$

## Exemples

(1) :  $\neg P(X) \vee R(X)$  et  $P(f(a)) \vee P(b)$

(2) :  $\neg P(f(X), f(a)) \vee R(X, b)$  et  $P(f(f(X)), Y) \vee R(f(a), Y)$

Res :  $R(f(X_2), b) \vee R(f(a), f(a))$  ( $X_1 \mapsto f(X_2), Y \mapsto f(a)$ )

# Unification en LPPO

# Exemples

Exemple 1 :  $P(f(X), g(Y, Y))$  et  $P(Z, g(b, b))$

Exemple 2 :  $P(f(X), Y)$  et  $P(f(a), g(b, b))$

Exemple 3 :  $P(f(X), g(Y, Y))$  et  $P(a, g(a, b))$

Exemple 4 :  $P(f(X), Y)$  et  $P(X, g(b, b))$

# Algorithme de [Martelli et al 1982]

## Unification de $P(t_1, \dots, t_k)$ et $P(t'_1, \dots, t'_k)$

- 1 *Initialisation.*  $G \leftarrow \{t_1 \doteq t'_1, \dots, t_k \doteq t'_k\}$  et  $\sigma = \emptyset$
- 2 Tant que  $G$  non vide, choisir  $C = (\alpha \doteq \beta)$  dans  $G$  et :
  - 1 (swap) Si  $\alpha$  est une fonction<sup>1</sup> et  $\beta$  une variable, les inverser
  - 2 (del) Si  $\alpha$  et  $\beta$  sont égaux, retirer  $C$  de  $G$
  - 3 Si  $\alpha = f(t_1, \dots, t_i)$  est une fonction
    - (conflict) Si  $\beta = g(s_1, \dots, s_j)$  est une fonction différente ( $g \neq f$  ou  $i \neq j$ ), renvoyer  $\perp$  (Echec)
    - (decomp) Sinon ( $\beta = f(s_1, \dots, s_i)$ ), retirer  $C$  de  $G$  et y ajouter  $\{t_1 \doteq s_1, \dots, t_i \doteq s_i\}$
  - 4 Si  $\alpha = X$  est une variable :
    - (check) Si  $X$  apparaît dans  $\beta$  : renvoyer  $\perp$  (Echec)
    - (eliminate) Sinon, retirer  $C$  de  $G$ , remplacer  $X$  par  $\beta$  dans toutes les expressions de  $G$  et  $\sigma$ , puis ajouter  $\{X \mapsto \beta\}$  à  $\sigma$ .
- 3 Renvoyer  $\sigma$

---

1. Les constantes sont des fonctions d'arité 0

# Exemple complexe

$\phi_1 : Enc(X, e(m(Y, e(Z))), m(e(X), e(a)))$  et

$\phi_2 : Enc(m(e(b), a), e(m(U, e(Z))), m(e(m(V, W)), Z))$

(soit  $X \leq e^{Y.e^Z} \leq e^X.e^a$  et  $e^b.a \leq e^{U.e^Z} \leq e^{V.W}.Z$ )

Application de l'algo : arborescent ou alignement (voir tableau)

# Unification pour la résolution

## Renommage

Les variables d'une clause étant universellement quantifiées, on peut renommer les variables avant de chercher à unifier des littéraux de 2 clauses complémentaires.

## Exemple

$P(f(X), Y)$  et  $P(X, g(b, b))$  ne sont pas unifiables .

Réso.  $C_1 = P(f(X), Y) \vee Q(X, Y)$  et  $C_2 = \neg P(X, g(b, b)) \vee R(X)$  ?

Méthode A : on renomme  $X$  en  $Z$  dans  $C_2$ .  $\sigma = \{Z \mapsto f(X), Y \mapsto g(b, b)\}$

$$\frac{P(f(X), Y) \vee Q(X, Y) \quad \neg P(Z, g(b, b)) \vee R(Z)}{Q(X, g(b, b)) \vee R(f(X))} \sigma$$

Méthode B : On n'explicite pas le renommage, mais on scinde  $\sigma$  en deux selon les variables substituées :  $\sigma_1 = \{X \mapsto f(X)\}$  et  $\sigma_2 = \{Y \mapsto g(b, b)\}$

$$\{Y \mapsto g(b, b)\} \frac{P(f(X), Y) \vee Q(X, Y) \quad \neg P(X, g(b, b)) \vee R(X)}{Q(X, b) \vee R(f(X))} \{X \mapsto f(X)\}$$

# Raisonnement par résolution

# Mise sous forme clausale

## Méthode

La résolution se fait sur une théorie clausale. Mais toute formule peut-être mise sous cette forme.

Etant donné une formule close  $\mathcal{F}$

- 1 Mettre la formule sous forme normale négative
- 2 Mettre la formule sous forme prénexe
- 3 Eliminer les  $\exists$  par skolémisation
- 4 Mettre sous forme normale conjonctive (CNF) et séparer en clauses.

# Mise sous forme clausale

## Forme normale négative

Etant donné une formule close  $\mathcal{F}$

- 1 Réécrire toutes les équivalences

$$\varphi \leftrightarrow \psi \mapsto (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$$

- 2 Réécrire toutes les implications

$$\varphi \rightarrow \psi \mapsto \neg\varphi \vee \psi$$

- 3 Propager les négations jusqu'aux atomes

$$\neg(\forall X \varphi(X)) \mapsto \exists X(\neg\varphi(X))$$

$$\neg(\exists X \varphi(X)) \mapsto \forall X(\neg\varphi(X))$$

$$\neg(\varphi \vee \psi) \mapsto \neg\varphi \wedge \neg\psi$$

$$\neg(\varphi \wedge \psi) \mapsto \neg\varphi \vee \neg\psi$$

Forme normale négative : toutes les négations sont devant une formule atomique

# Mise sous forme clausale

## Forme prénexe

Etant donné une formule close  $\mathcal{F}$  sous forme normale négative

- 1 Renommer les variables qui apparaissent plusieurs fois
- 2 Ramener tous les quantificateur au début de la formule en les étendant à toutes la formule (mais en conservant leur ordre d'apparition !)

Forme prénexe : tous les quantificateurs sont en tête de formule et s'appliquent à toute la formule

# Mise sous forme clausale

## Skolémisation

Etant donné une formule close  $\mathcal{F}$  sous forme prénexe

- Remplacer de gauche à droite chaque variable existentielle ( $\exists X$ ) par une fonction de skolem (nouveau symbole de fonction) prenant pour argument toutes les variable universellement quantifiées précédant le  $\exists X$

Exemples :

$$\exists X(P(X) \vee \neg Q(a, X)) \mapsto$$

$$\forall X_1 \exists X_2 \forall X_3 \forall X_4 \exists X_5 \exists X_6 \forall X_7 \varphi(X_1, X_2, X_3, X_4, X_5, X_6, X_7) \mapsto$$

Après la skolémisation, il n'y a plus de quantificateurs existentiels

## Skolémisation

Etant donné une formule close  $\mathcal{F}$  sous forme prénexe

- Remplacer de gauche à droite chaque variable existentielle ( $\exists X$ ) par une fonction de skolem (nouveau symbole de fonction) prenant pour argument toutes les variable universellement quantifiées précédant le  $\exists X$

Exemples :

$$\exists X(P(X) \vee \neg Q(a, X)) \mapsto P(g) \vee \neg Q(a, g)$$

$$\forall X_1 \exists X_2 \forall X_3 \forall X_4 \exists X_5 \exists X_6 \forall X_7 \varphi(X_1, X_2, X_3, X_4, X_5, X_6, X_7) \mapsto$$

$$\forall X_1 \forall X_3 \forall X_4 \forall X_7 \varphi(X_1, f_1(X_1), X_3, X_4, f_2(X_1, X_3, X_4), f_3(X_1, X_3, X_4), X_7)$$

Après la skolémisation, il n'y a plus de quantificateurs existentiels

# Mise sous forme clausale

## Forme normale conjonctive

Etant donné une formule close  $\mathcal{F}$  sans quantificateurs existentiels

- Propager toutes les disjonctions jusqu'aux littéraux :

$$\varphi \vee (L_1 \wedge \dots L_n) \mapsto (L_1 \vee \varphi) \wedge \dots (L_n \vee \varphi)$$

Forme normale conjonctive : conjonctions de disjonctions

$\mathcal{F}' = \forall X_1 \dots \forall X_k (C_1 \wedge \dots C_n)$  où chaque  $C_i = L_1^i \vee \dots L_{p_i}^i$

## Forme clausale

Etant donné  $\mathcal{F} = \forall X_1 \dots \forall X_k (C_1 \wedge \dots C_n)$ , on retire les quantificateurs universels (implicites dans les clauses) et on rassemble les  $C_i$  dans  $\Sigma$

$$\Sigma = \{C_1, \dots, C_n\}$$

Rappel :  $\forall X(\varphi(X) \wedge \psi(X)) \equiv \forall X\varphi(X) \wedge \forall X\psi(X)$

# Mise sous forme clausale

Exemple :

$$\forall X((\exists Y P(X, Y) \wedge Q(Y)) \rightarrow (\exists Z(P(X, Z) \wedge R(Z, f(X))))$$

# Mise sous forme clausale

Exemple :

$$\forall X((\exists Y P(X, Y) \wedge Q(Y)) \rightarrow (\exists Z(P(X, Z) \wedge R(Z, f(X))))$$

$$\Sigma = \{C_1 : \neg P(X, Y) \vee \neg Q(Y) \vee P(X, f_1(X)), \\ C_2 : \neg P(X, Y) \vee \neg Q(Y) \vee R(f_1(X), f(X))\}$$

# Preuve par résolution

## Définition

Etant donnés un ensemble de clauses  $\Sigma$  (clauses d'entrées) et une clause  $C$  (la conclusion), une *preuve par résolution de  $C$  à partir de  $\Sigma$*  est un ensemble de clauses  $R_1, \dots, R_n$  tel que

- Pour chaque  $i$ ,  $R_i = \text{Res}(G_i, D_i, \sigma_i)$  où  $G_i$  et  $D_i$  sont pris dans  $\Sigma \cup \{R_1, \dots, R_{i-1}\}$
- $R_n = C$

On a alors

$$\Sigma \vdash_{\text{res}} C$$

Cette procédure est correcte est complète ( $\Sigma \models C \Leftrightarrow \Sigma \vdash_{\text{res}} C$ )

# Exemples

$$\Sigma_1 = \{p \vee q, \neg p \vee q, p \vee \neg q, \neg p \vee \neg q\}$$

Prouver  $\Sigma_1 \vdash_{res} \square$

$$\Sigma_2 = \{\neg P(f(X)) \vee Q(X, Y), \neg Q(X, X) \vee P(X), P(f(f(a)))\}$$

Prouver  $\Sigma_2 \vdash_{res} P(a)$

# Stratégies

## Stratégie

Pour construire une preuve, il faut choisir à chaque étape deux clauses  $G_i$  et  $D_i$ .

$G_1 \in \Sigma$  est appelée la top-clause.

Preuve linéaire :  $\forall i > 1, G_i = R_{i-1}$ .

## Preuve par réfutation

Soit une requête conjonctive  $Q = \bigwedge_{i \in \{1, \dots, k\}} L_i$ .

Pour prouver  $\Sigma \models Q$ , on prouve  $\Sigma \cup \{\neg Q\} \vdash_{res} \square$

Typiquement,  $\neg Q = \neg L_1 \vee \neg L_k$  est utilisée comme top-clause.

## SLD-resolution pour la réfutation

SLD : Selective, Linear, Definite.

Pour prouver  $\Sigma \cup \{\neg Q\} \vdash_{res} \square$ .

- $G_1 = \neg Q$
- Pour chaque  $i > 1$ ,  $G_i = R_{i-1}$
- Pour chaque  $i$ , on choisit (de façon déterministe) un littéral de  $R_{i-1}$  et on prend pour  $D_i$  une clause de  $\Sigma$  contenant un littéral complémentaire.

L'existence d'une telle preuve se note  $\Sigma \vdash_{sld} Q$ .

# SLD-résolution

## Programme défini

Clause de Horn : clause ne contenant qu'au plus un littéral positif

- soit  $A_1 \vee \neg L_1 \dots \vee \neg L_k$  qui peut aussi s'écrire  $A_1 \leftarrow L_1, \dots, L_k$
- soit  $\neg L_1 \dots \vee \neg L_k$  qui peut aussi s'écrire  $\perp \leftarrow L_1, \dots, L_k$

Théorie définie : théorie clausale qui ne contient que des clauses de Horn

## Propriété

La SLD résolution est correcte, mais non complète dans le cas général ( $\Sigma \vdash_{sld} C \Rightarrow \Sigma \models C$  mais pas dans le sens inverse).

Quand  $\Sigma \cup \{\neg Q\}$  est une théorie définie, la SLD résolution est complète ( $\Sigma \vdash_{sld} C \Leftarrow \Sigma \models C$ ).

# SLD-résolution

## Arbre de résolution

Pour explorer l'ensemble des preuves possibles, on construit un arbre :

- La racine est la top-clause  $\neg Q$  (qu'on note  $R_0$ ), initialement ouverte
- On itère jusque saturation
  - Pour chaque feuille ouverte  $R_i \neq \square$ , on choisit un littéral  $L \in R_i$  et on détermine l'ensemble des clauses  $D_i^k$  de  $\Sigma$  contenant un littéral unifiable à  $\neg L$
  - Si cet ensemble est vide, on ferme le noeud (fail).
  - Sinon, on construit pour chaque  $D_i^k$  de l'ensemble, un noeud fils  $R_{i+1} = \text{Res}(R_i, D_i^k, \sigma_i^k)$

Chaque chemin  $R_1^k, \dots, R_n^k$  de la racine (exclue) vers une feuille  $\square$  donne une preuve par SLD-résolution de  $\sigma(Q)$  (où  $\sigma$  est la composition de tous les  $\sigma_i^k$  sur le chemin).

Si toutes les feuilles sont fermées, il n'y a pas de preuves de  $Q$  à partir de  $\Sigma$  (renvoie fail).

Si l'on cherche une seule preuve, on peut s'arrêter dès que l'on trouve  $\square$ .

# Exemple

1.  $[P(x,y), \neg Q(x,z), \neg R(z,y)]$

2.  $[P(x,x), \neg S(x)]$

3.  $[Q(x,b)]$

4.  $[Q(b,a)]$

5.  $[Q(x,a), \neg R(a,x)]$

6.  $[R(b,a)]$

7.  $[S(x), \neg T(x,a)]$

8.  $[S(x), \neg T(x,b)]$

9.  $[S(x), \neg T(x,x)]$

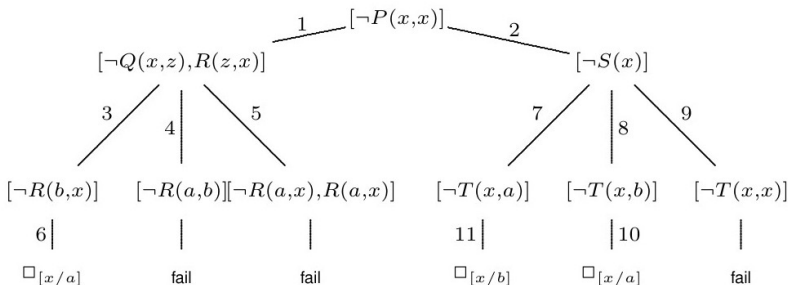
10.  $[T(a,b)]$

11.  $[T(b,a)]$

cíl:  $[\neg P(x,x)]$

# Exemple

- |   |                            |                          |
|---|----------------------------|--------------------------|
| 1. $[P(x,y), \neg Q(x,z), \neg R(z,y)]$ | 5. $[Q(x,a), \neg R(a,x)]$ | 9. $[S(x), \neg T(x,x)]$ |
| 2. $[P(x,x), \neg S(x)]$                | 6. $[R(b,a)]$              | 10. $[T(a,b)]$           |
| 3. $[Q(x,b)]$                           | 7. $[S(x), \neg T(x,a)]$   | 11. $[T(b,a)]$           |
| 4. $[Q(b,a)]$                           | 8. $[S(x), \neg T(x,b)]$   | cél: $[\neg P(x,x)]$     |



# Programmation logique

## Des clauses aux règles

Soit une clause  $C_1 = P_1 \vee \dots \vee P_k \vee \neg P_{k+1} \vee \dots \vee \neg P_n$ . Elle peut se réécrire en une règle  $R_1 = (P_{k+1} \wedge \dots \wedge P_n) \rightarrow (P_1 \vee \dots \vee P_k)$

On appelle programme logique un ensemble de règle

Par convention, on met la règle en tête, on utilise  $:-$  pour représenter  $\leftarrow$ , les  $;$  en tête correspondent à des  $\vee$  et les  $,$  à des  $\wedge$  :  
 $P_{k+1}; \dots; P_n \text{ } :- \text{ } P_1, \dots, P_k.$

# Programme logique

## Programme défini

Clause de Horn : clause ne contenant qu'au plus un littéral positif

$A_0 \quad \mapsto \quad A_0. \quad \text{(fait)}$

$A_0 \vee \neg B_1 \dots \vee \neg B_k \quad \mapsto \quad A_0 :- B_1, \dots, B_k. \quad \text{(règle)}$

$\neg B_1 \dots \vee \neg B_k \quad \mapsto \quad ?- B_1, \dots, B_k. \quad \text{(requête)}$

Programme défini : programme qui ne contient que des clauses de Horn.

## Résolution de règles

Avec la notation en règles des clauses, la règle de résolution s'écrit :

$$\frac{?- Q_1, \dots, Q_s \quad Q_1 :- P_1, \dots, P_k}{?- P_1, \dots, P_k, Q_2, \dots, Q_s} \quad X = \dots$$

## Principes

Langage de programmation logique basé sur le SLD-résolution. Etant donné un programme (défini)  $\mathcal{P}$  chargé en mémoire (ensemble ordonné de règles), l'utilisateur fait une requête conjonctive  $Q$ .

Stratégie SLD :

- Choix du littéral ? de gauche à droite (L'ordre d'écriture des atomes dans le corps des règles importe)
- Choix de l'ordre d'utilisation des règles applicable ? de haut en bas (L'ordre d'écriture des règles importe)

Réponse : true (si  $\mathcal{P} \models Q$ ) ou une substitution  $\sigma$  sur les variables de  $Q$  telle que de  $\mathcal{P} \models \sigma(Q)$ .

Possibilité de demander plusieurs réponses avec ";" (donne réponse suivante dans le parcours en profondeur de l'arbre SLD, ou false si plus de réponse)

# Prolog : Syntaxe

## Éléments (1/2)

- Termes (apparaissent comme arguments d'un prédicat ou d'un opérateur prolog infixe – par ex. =, ==, is... )
  - Constantes : chaînes de caractères commençant par une minuscule (non suivie de parenthèses)  
Exemples : alice, edward, james\_bond\_007, a, x, x12
  - Nombres : interprétés comme constantes, mais prolog contient des primitives arithmétiques  
Exemples : 0, 12, -1...
  - Variables : chaînes de caractères commençant par une majuscule ou par \_  
Exemples : X, \_G341, Stranger, The\_Older, A, X12
  - Fonctions : chaînes de caractères commençant par une minuscule (suivie de parenthèses avec les arguments)  
Exemples : f(a,b), s(s(0)), pireEnnemi(X), ...
  - Listes : expression encadrée par [ ] (voir plus tard)

# Prolog : Syntaxe

## Éléments (2/2)

- Prédicats : commencent aussi par minuscules (la différence avec les fonctions se fait par la position : les prédicats sont dans une règle, les fonctions dans un prédicat)

Exemples : `parents(X,Y,Z)`, `addition(X,Y,Res)`, ...

- Règles : chaque ligne du programme. Commence par un prédicat et finit par un point.

- Faits : juste un prédicat suivi d'un point.

Exemples : `parents(edward, victoria, albert)`,  
`addition(s(0),s(0),s(s(0)))`., `addition(0,X,X)` .

- Règle : tête de la règle (conclusion), suivi de `:-`, d'un corps de règle et d'un point.

Exemples : `soeur(X,Y) :- femme(X)`,  
`parents(X,Mere,Pere)`, `parents(Y,Mere,Pere)` .,

# Prolog : Syntaxe

## Version de prolog

Dans l'UE, on utilise SWI-prolog.

Téléchargement : <https://www.swi-prolog.org/>

## Programmes

Mode *consult*.

Contient les faits et les règles qui définissent le problème.

Il est demandé de mettre ensemble les règles ayant le même prédicat en tête (facilite le débogage).

Edité à part et enregistrer dans un fichier ".pl".

# Prolog : Syntaxe

## Requêtes

Mode *query*.

C'est l'invite de commande quand on lance prolog

Une requête est un prédicat, ou une séquence de prédicats (interprétés conjonctivement). Si la requête contient des variables, prolog essaie de trouver une substitution permettant de la résoudre (si nécessaire).

Les commandes natives de prolog sont exprimées comme des prédicats :

- `[filename]` . permet de charger un programme.
- `halt` . permet de quitter prolog.

# Exemple

## Programme famille.pl

```
femme(alice).  
femme(victoria).  
homme(albert).  
homme(edward).  
parents(edward, victoria, albert).  
parents(alice, victoria, albert).  
soeur(X,Y) :- femme(X),  
               parents(X,Mere,Pere),  
               parents(Y,Mere,Pere).
```

## Mode query

```
?-[famille.pl].
```

```
true.
```

```
?-femme(alice).
```

# Exemple

## Programme famille.pl

```
femme(alice).  
femme(victoria).  
homme(albert).  
homme(edward).  
parents(edward, victoria, albert).  
parents(alice, victoria, albert).  
soeur(X,Y) :- femme(X),  
               parents(X,Mere,Pere),  
               parents(Y,Mere,Pere).
```

## Mode query

```
?-[famille.pl].
```

```
true.
```

```
?-femme(alice).
```

```
true.
```

```
?-femme(albert).
```

# Exemple

## Programme famille.pl

```
femme(alice).
```

```
femme(victoria).
```

```
homme(albert).
```

```
homme(edward).
```

```
parents(edward, victoria, albert).
```

```
parents(alice, victoria, albert).
```

```
soeur(X,Y) :- femme(X),  
              parents(X,Mere,Pere),  
              parents(Y,Mere,Pere).
```

## Mode query

```
?-[famille.pl].  
true.
```

```
?-femme(alice).  
true.
```

```
?-femme(albert).  
false.
```

```
?-homme(X).
```

# Exemple

## Programme famille.pl

```
femme(alice).  
femme(victoria).  
homme(albert).  
homme(edward).  
parents(edward, victoria, albert).  
parents(alice, victoria, albert).  
soeur(X,Y) :- femme(X),  
               parents(X,Mere,Pere),  
               parents(Y,Mere,Pere).
```

## Mode query

```
?-[famille.pl].
```

```
true.
```

```
?-femme(alice).
```

```
true.
```

```
?-femme(albert).
```

```
false.
```

```
?-homme(X).
```

```
X=albert
```

# Exemple

## Programme famille.pl

```
femme(alice).  
femme(victoria).  
homme(albert).  
homme(edward).  
parents(edward, victoria, albert).  
parents(alice, victoria, albert).  
soeur(X,Y) :- femme(X),  
               parents(X,Mere,Pere),  
               parents(Y,Mere,Pere).
```

## Mode query

```
?-[famille.pl].  
true.  
  
?-femme(alice).  
true.  
  
?-femme(albert).  
false.  
  
?-homme(X).  
X=albert ;  
X=edward.  
  
?-femme(X), parents(X,U,I).
```

# Exemple

## Programme famille.pl

```
femme(alice).  
femme(victoria).  
homme(albert).  
homme(edward).  
parents(edward, victoria, albert).  
parents(alice, victoria, albert).  
soeur(X,Y) :- femme(X),  
               parents(X,Mere,Pere),  
               parents(Y,Mere,Pere).
```

## Mode query

```
?-[famille.pl].  
true.  
  
?-femme(alice).  
true.  
  
?-femme(albert).  
false.  
  
?-homme(X).  
X=albert ;  
X=edward.  
  
?-femme(X), parents(X,U,I).  
X=alice, U=victoria, I=albert
```

# Exemple

## Programme famille.pl

```
femme(alice).
femme(victoria).
homme(albert).
homme(edward).
parents(edward, victoria, albert).
parents(alice, victoria, albert).
soeur(X,Y) :- femme(X),
               parents(X,Mere,Pere),
               parents(Y,Mere,Pere).
```

## Mode query

```
?-[famille.pl].
true.

?-femme(alice).
true.

?-femme(albert).
false.

?-homme(X).
X=albert ;
X=edward.

?-femme(X), parents(X,U,I).
X=alice, U=victoria, I=albert ;
false.

?-soeur(U, V).
```

# Exemple

## Programme famille.pl

```
femme(alice).
femme(victoria).
homme(albert).
homme(edward).
parents(edward, victoria, albert).
parents(alice, victoria, albert).
soeur(X,Y) :- femme(X),
               parents(X,Mere,Pere),
               parents(Y,Mere,Pere).
```

## Mode query

```
?-[famille.pl].
true.
```

```
?-femme(alice).
true.
```

```
?-femme(albert).
false.
```

```
?-homme(X).
X=albert ;
X=edward.

?-femme(X), parents(X,U,I).
X=alice, U=victoria, I=albert ;
false.
```

```
?-soeur(U, V).
U=alice, V=edward
```

# Exemple

## Programme famille.pl

```
femme(alice).
femme(victoria).
homme(albert).
homme(edward).
parents(edward, victoria, albert).
parents(alice, victoria, albert).
soeur(X,Y) :- femme(X),
               parents(X,Mere,Pere),
               parents(Y,Mere,Pere).
```

## Mode query

```
?-[famille.pl].
true.

?-femme(alice).
true.

?-femme(albert).
false.

?-homme(X).
X=albert ;
X=edward.

?-femme(X), parents(X,U,I).
X=alice, U=victoria, I=albert ;
false.

?-soeur(U, V).
U=alice, V=edward ;
U=alice, V=alice
```

# Exemple

## Programme famille.pl

```
femme(alice).
femme(victoria).
homme(albert).
homme(edward).
parents(edward, victoria, albert).
parents(alice, victoria, albert).
soeur(X,Y) :- femme(X),
               parents(X,Mere,Pere),
               parents(Y,Mere,Pere).
```

## Mode query

```
?-[famille.pl].
true.

?-femme(alice).
true.

?-femme(albert).
false.

?-homme(X).
X=albert ;
X=edward.

?-femme(X), parents(X,U,I).
X=alice, U=victoria, I=albert ;
false.

?-soeur(U, V).
U=alice, V=edward ;
U=alice, V=alice ;
false.
```

# Inversion de fonction

## Variables dans la requête

Prolog essaie de résoudre la requête en trouvant des règles dont la tête est unifiable avec la requête.

Une définition peut répondre à plusieurs questions :

`?-soeur(X, edward).`

*Qui sont les soeurs d'Edward ?*

`X=alice ; false.`

`?-soeur(alice, X).`

*Qui a Alice pour soeur ?*

`X=edward ; X= alice ; false.`

`?-addition(s(0), s(s(0)),X).`

*Combien font 1+2 ?*

`X=s(s(s(0))).`

`?-addition(X, Y, s(s(0))).`

*Quels sont les (x,y) tels que  $x+y=2$  ?*

`X=0,Y=s(s(0)); X=s(0),Y=s(0);X=s(s(0)),Y=0.`

# Quelques autres notions en prolog

## Unifiable vs égal

- $exp_1 = exp_2$  vérifié si  $exp_1$  est unifiable avec  $exp_2$   
`?-a=a. true., ?- X=a. X=a., ?-X=Y. X=Y., ?-f(X,Y)=f(g(Y,b),a).  
X=g(a,b),Y=a., ?-f(X,X)=f(g(Y,b),a). false.`
- $exp_1 == exp_2$  vérifié si  $exp_1$  est égal à  $exp_2$   
`?-a=a. true., ?- X=a. false., ?-X=Y. false., ?-f(X,Y)=f(X,Y).  
true., ?-f(X,Y)=f(X,X). false.`

# Quelques autres notions en prolog

## Listes

### Constructeurs de listes :

- Liste vide : `[]`
- Constructeur de liste : `[Tete|Queue]` où `Tete` doit être un élément de la liste et `Queue` une liste.
- Liste à plusieurs éléments : `[X]` (liste à 1 élément), `[X,Y]` (liste à deux éléments)
- Récupérer les `n` premier éléments : `[X1,X2,...,Xn|Reste]` (liste commençant par `X1`, ... `Xn` et se poursuivant par les éléments de la liste `Reste`.)

## Arithmétique

- $t$  is  $exp$  permet d'évaluer arithmétiquement l'expression  $exp$  et d'en unifier le résultat à  $t$  (ie affecter si  $t$  variable, comparer si  $t$  constante). Les variables de  $exp$  doivent être instantiées avant de résoudre le  $is$ .

Exemple :  $?-K=3, N$  is  $K+1$ .  $K=3$ ,  $N=4$ .

## Négation

- Il existe en prolog une négation dite 'par l'échec' notée par  $not(\dots)$ .
- Pour résoudre  $?-not(P), \dots$ , prolog tente de résoudre (nouvel arbre SLD) la requête  $?-P$ 
  - Si cette résolution échoue, alors  $?-not(P)$  est true.
  - Sinon,  $?-not(P)$  échoue (fail).