École Centrale de Lyon
Cours optionnel 3° Année

Concepts et Paradigmes de Programmation

S. Saidi
M. I . S.

1993-1994

CHAPITRE I

Fondements du Paradigme Logique

Ecole Centrale de Lyon
Département Mathématiques,
Informatiques et Systèmes
1993-94
S. SAIDI

Table des matières

I. Introduction 4
I.1- Qu'est-ce que la programmation logique ? 4
I.2- A quoi ressemble un programme logique ? 4
I.3- Quelles différences avec les autres paradigmes ? 5
I.4- Intérêts de la programmation logique 6
I.5- Deux approches pour comprendre la programmation logique 7
II. Le langage de la logique du premier ordre 8
III. Eléments de la théorie de modèles 9
III.1- Quelques définitions 11
IV. Langage des clauses 13
IV.1- Classification des clauses 13
IV.2- Réarrangement des littéraux 14
IV.3- La forme clausale de la logique 16
V. Eléments de la théorie de preuve 18
V.1- Utilisation de l'implication dans la résolution de problèmes 20
V.2- Utilisation de la preuve dans la résolution de problèmes 21
V.3- Validité des programmes 23
VI. Programmation avec la logique 24
VI.1- Le domaine de Herbrand 24
VI.2- Systèmes d'inférence 26
VI.2.1- Preuve par Confirmation / Contradiction 26
VI.2.2- Preuve par contradiction : procédure de Herbrand 28
VI.2.2.1- Saturation / Résolution 29
VI.2.3- Résolution : cas de la logique propositionnelle 29
VI.2.4- Résolution : cas de la logique des prédicats 30
VI.2.4.1- Application et composition de substitutions 30
VI.2.4.2- Extension de la résolution aux prédicats du 1er ordre 32
VI.3- Programmation avec la résolution 34
VI.3.1- Extraction des réponses 36

VII. La SLD-Résolution 37
VII.1- Unification 40
VII.2- Stratégie de recherche 42
VII.3- Règles de choix 43
VII.4- Interprétation procédurale 44
VII.4.1- Contrôle avec CUT 45
VIII. Sémantique des programmes définis 47
VIII.1- Interprétation de Herbrand 47
VIII.1.1- Définition formelle 48
VIII.2- Les modèles d'un programme 49
VIII.3- Modèles minimaux et conséquence logique 51
VIII.3.1- Résultats importants 52
VIII.4- Modèle minimal et la dérivation 52
VIII.5- Construction du modèle minimal 55
VIII.6- Interprétation point-fixe du modèle minimal 56
VIII.6.1- Définitions formelles 56
IX. Sémantiques dénotationnelles des programmes définis 59
X. Traitement de la Négation 60
X.1- Hypothèse du monde fermé 60
X.2- Complétion de programmes 62
X.2.1- La complétion en propositionnelle 63
X.2.2- La complétion en prédicats du premier ordre 66
XI. L'échec fini 69
XI.1- Extension de la SLD avec l'échec fini 69
XI.2- Ensemble d'échecs fini de la SLD résolution 72
XI.3- Caractérisation mathématique de l'échec fini 75
XI.4- Sémantique de l'échec fini par complétion 80

I. Introduction

I.1- Qu'est-ce que la programmation logique ?

- Un formalisme de calcul qui combine deux principes fondamentaux :
  - L'utilisation de la logique pour exprimer des connaissances
  - L'utilisation de l'inférence pour manipuler ces connaissances

- En résolution de problèmes, le premier principe concerne la représentation de connaissance sous la forme

  Conclusions si Conditions

et le second concerne les liens entre les conditions et les conclusions.

- Le but est d'inférer certaines conclusions à partir des hypothèses (conditions) d'une manière "raisonnable" (du point de vu calcul).

- Concrètement, on utilise le noyau **Forme Clausale** + **Résolution** où
  - La forme clausale est un sous ensemble de la logique des prédicats du premier ordre pour exprimer les connaissances et
  - La résolution est un système d'inférence particulier pour leur manipulation.

- Pour obtenir une implantation "raisonnable" telle que Prolog, on ajoute à ce noyau une stratégie de contrôle.

I.2- A quoi ressemble un programme logique ?

```
aime(SanAntonio, Tout_le_monde)   si
                                  achete(Tout_le_monde, Son_livre).
achete(Tout_le_monde, Son_livre)  si
                                  sensible(Tout_le_monde).
sensible(Vous).
```

- A partir de ce programme, on peut déduire (inférer) que SanAntonio vous aime.

- Le but est de comprendre pourquoi cela et quel est son intérêt.

I.3- Quelles différences avec les autres paradigmes ?

En programmation logique (programmation relationnelle) :

- On décrit le "Quoi". Le "Comment" est figé par un certain algorithme fixe (cet algorithme implante le principe de résolution).

- Une variable est une inconnue algébrique dont on cherche la valeur.
  Une variable est <u>instanciée</u> lorsqu'une variable reçoit une valeur.

- On ne peut pas modifier la valeur des variables instanciées.

- Il n'y a pas d'affectation (au sens mise à jour des variables ou des paramètres par des opérations telles que X:=X+1).

- La résolution calcule une instance de la <u>relation</u> décrite par le programme.

- Exemples de relations :

Relation	***Signification***
pere(jean, paul).	*Jean est le père de paul*
fils(X, Y) <- pere(Y, X).	*X est le fils de Y si Y est le père de X*
origine(X,Y) <- X=0 & Y =0.	*Le couple (X,Y) est l'origine si X=Y=0*

dans_cercle(X,Y, R) <- $X^2 + Y^2 < R^2$.	*Le point (X,Y) est dans le cercle de rayon R si* $X^2 + Y^2 < R^2$
fermat(A,B,C,N) <- enum(A) , enum(B) , enum(C) , $A^N + B^N = C^N$. enum(X) <- X□{1,2,... }.	*Trouver les entiers A,B,C,N tels que* $A^N + B^N = C^N$
entier(0). entier(succ(N)) <- entier(N).	*0 est un entier* *Si N est un entier alors son successeur l'est également*

I.4- Intérêts de la programmation logique

- Tout calcul est concerné par la <u>représentation</u> et la <u>dérivation</u> de connaissances.
- La programmation logique est la base même de “la programmation à base de connaissances”.
- La pouvoir d’expression de la logique permet de formuler les connaissances sur le domaine du problème d’une façon directe et <u>indépendante de toute implantation.</u>
  - ③ Programmes résultants sont plus compacts, plus flexibles, plus lisibles et plus simples à maintenir

- Les conclusions calculées sont les <u>conséquences logiques exactes</u> de la formulation.

- Elle permet également de partir des conclusions et de retrouver les hypothèses.

- Elle procure un cadre uniforme pour le génie-logiciels : spécification, construction, exécution et manipulation de programmes, ...
- On peut établir les liens (mathématiques) entre
  - les hypothèses et les conclusions
  - les spécifications et les programmes

- les programmes et les autres programmes

Les aspects de la programmation à base de connaissance

I.5- Deux approches pour comprendre la programmation logique

Théorie de modèles :

- Explique les relations entre les phrases logiques avec le monde réel (interprétations) pour leur donner des valeurs de vérité.
- En étudiant la théorie de modèles, on étudie les notions telles que : vrai, faux, interprétation, satisfaction, modèle, implication, conséquences sémantiques...

Théorie de preuve :

- Explique les relations entre les phrases logiques en terme de leur dérivabilité en utilisant des règles exploitant la forme syntaxique des phrases.
- En étudiant la théorie de preuve, on étudie les notions telles que : axiome, inférence, règle d'inférence, théorème, preuve, consistance, conséquence syntaxique..

II. Le langage de la logique du premier ordre

- Ensemble de phrases construites à partir de :

- *Constantes* a,b,c,1,2,5, toto
- *Symboles de fonctions* f,g, salaire, point
- *Variables* X, Y, Z
- *Symboles de prédicats* aime, aide, achete, sensible

- Un *terme* est
  - Une constante
  - Une variable
  - Un symbole fonctionnel appliqué à un tuple de n termes (n >0), n : *arité*

Exemples de termes:

a	X	f(b,f(b,Y,Z),Y)

- Une *formule atomique* : un symbole de prédicat appliqué à un tuple de n termes.

Exemples de formules atomiques :

aime(X, beau(tableau))
donne(jean, livre, pierre)

- Les phrases sont construites à partir des formules atomiques et
  - Des quantificateurs (∃, ∀)
  - Des connecteurs ~ , &, | , <- , <=>

- Une *formule bien formée* :
  - Une formule atomique
  - Pour W, W_1 et W_2 des formules bien formées, les constructions suivantes sont des formules bien formées :

(W)	~W	W1 & W2	W1 | W2
W1 <- W2	W1 <=> W2	∃X W	∀X W

Terminologie :

- Terme/Formule *clos* : Terme/Formule sans variable

- *Littéral* : formule atomique positive ou négative

Exemples:

aime(jean, marie) ~aime(paul, misere).

- *Prédicat* :Formule atomique

- Négation (~W)
  Conjonction (W_1 & W_2)
  Disjonction, (W_1 | W_2)
  Conditionnel (W_1 <- W_2)
  Bi-Conditionnel (W_1 <=> W_2)

- W_1 est *instance* de W_2 par la *substitution* $\{X_1/T_1,...,X_n/T_n\}$ où X_i/T_i est une <u>remplacement</u> (un lien).

Exemple :

avec	W =	aime(jean,X)	<-	aime(X,Z)
on a :	W'= W{X/pere(Y)} =			
		aime(jean, pere(Y))	<-	aime(pere(Y),Z)

- Instance close

- *Programme* : ensemble de formules bien formées

III. Eléments de la théorie de modèles

- Théorie de modèles : un moyen d'associer un sens à une phrase logique

- L'idée de base : associer la phrase à une certaines valeur de vérité à propos d'un domaine : **interprétation**.

- Le domaine est n'importe quel ensemble de notre choix.

- Pour obtenir une interprétation pour un domaine D, on associe
  - toute constante à un élément de D
  - tout symbole de fonction à une fonction de D^n -> D
  - tout symbole de prédicat à une relation R dans D^n (R $\prod D^n$)

  On a $D^n = \{<d_1,...,d_n> \mid d_i \square D\}$

Exemple : soit *aime(jean, marie)* une phrase. Soit le domaine considéré

- Afin de construire une interprétation, associons chaque constante de la phrase à un élément du domaine et associons une <u>relation binaire</u> sur le domaine avec le prédicat binaire *aime* . Les tuples de cette relation sont :

- Sous une telle <u>interprétation</u>, la phrase *aime(jean, marie)* est lue comme une proposition affirmant que le tuple <n, o > appartient au domaine de la relation.

- Si la proposition est "vraie", on dit que la phrase prend la <u>valeur vraie sous cette interprétation</u>.
  Avec la même relation sur le domaine et une nouvelle interprétation qui associe n à marie et o à jean, la phrase prend la valeur "fausse" car le tuple <o, n > n'appartient pas au domaine de la relation.

- Ainsi, une phrase peut être évaluée à vrai ou à faux selon l'interprétation que l'on en fait.

- Quand on écrit un programme logique, on a en général une interprétation en tête (interprétation attendue). Dire que nos phrases logiques représentent

un problème donné à résoudre, c'est dire que "sous cette interprétation attendue", elles auront la valeur vraie.

III.1- Quelques définitions

- On étend la notion d'interprétation aux phrases avec variables :

  - $\forall X\ W$ est vrai si pour tout élément du domaine associé aux occurrences de X, W est vrai.

  -$\exists X\ W$ est vrai s'il existe un élément du domaine associé aux occurrences de X pour lequel W est vrai.

- Extensions aux connecteurs :

  (W) est vrai si W est vrai.
  ~W est vrai si W est faux.
  W1 & W2 est vrai si W1 et W2 sont vrais.
  W1 | W2 est vrai si W1 ou bien W2 est vrai.
  W_1 *<- W2* est faux ssi W_1=faux et W_2=vrai.
  W_1 *<=>* W_2 est faux ssi W_1=W_2=vrai ou bien W_1=W_2=faux.

- Une interprétation qui donne la valeur vraie à une phrase <u>satisfait</u> la phrase.

- Une telle interprétation est un **modèle** pour la phrase.

Exemple :

pour la phrase *aime(jean,marie)*, l'interprétation choisie est un modèle.

- Une interprétation est un modèle pour un ensemble de phrases si elle est un modèle pour chacune des phrases de cet ensemble.

- Une phrase qui a au moins un modèle est **satisfiable**.

- Une phrase qui n'a aucun modèle est <u>non-satisfiable</u>.

Exemple :

aime(jean,marie)	est satisfiable.
(∃X,Y) (aime(X,Y) & ~aime(X,Y))	est non-satisfiable.

- Extension de la notion de satisfiabilité à un ensemble de phrases.
- Une phrase pour laquelle toute interprétation est un modèle est **valide**.

Exemple :

(∀X,Y) (aime(X,Y) | ~aime(X,Y))	est valide.
aime(jean, marie)	n'est pas valide

car, comme nous l'avons vu ci-dessus, on peut trouver une interprétation qui n'est pas un modèle pour la phrase *aime(jean, marie)*.

- S_1 implique logiquement S_2 (noté $\mathbf{S_1 \models S_2}$) ssi tout modèle pour S_1 est un modèle pour S_2.

  S_2 est **conséquence logique** (sémantique) de S_1.
  "|=" est réflexive et transitive.

Exemple :

(∀X) (aime(jean, X) |= aime(jean, marie) .

- On étend la notion de la conséquence logique à un ensemble de phrases Γ:

Γ |= s si *s* est satisfait par tout modèle de Γ.

- $S_1 \models S_2$ et $S_2 \models S_1$ impliquent $S_1 \int S_2$ (S_1 est équivalent à S_2)

Exemple d'équivalence :

(∀X) ~aime(X, misere) ∫ ~ (∃X) aime(X, misere)

- Deux phrases logiquement équivalentes ont le même sens : elles partagent les mêmes modèles.

IV. Langage des clauses

- Une clause est une des formes normales d'expression des phrases de la logique du premier ordre.
- c'est une forme régulière de phrases.
- Le format général : **<quantificateurs> <matrice>**

Pour dire :

"jean aime tout le monde" : (∀X) aime(jean, X)

"jean aime quelqu'un" : (∃X) aime(jean, X)

- Exemples (les quantificateurs universels omis):

" personne n'aime personne" : ~aime(X,Y)

" chacun s'aime" : aime(X,X)

" tout est bien qui fini bien" : bien(X) | ~finit_bien(X)

" la fin justifie les moyens" :

justifie(Y, X) | ~(fin(X) & moyen(X,Y))

∫ justifie(Y, X) | ~ fin(X) | ~moyen(X,Y)

La relation "X est sous-ensemble de Y" :

(∀X)(∀Y) [s(X,Y) <=> (∀U) (U□Y <- U□X)]

IV.1- Classification des clauses

- Classification par rapport à leur littéraux négatifs et positifs

- Une **clause définie** : contient <u>exactement un</u> littéral positif et (n ≥0) littéraux négatifs.

Exemple de clauses définies :

```
content(jean) | ~mange(jean,fois_gras)
content(paul)
```

- Une **clause de Horn** : une clause définie avec <u>au plus un</u> littéral positif.

Exemple de clauses de Horn :

```
~mange(jean, fois_gras)
```

```
content(paul)
```

- Une clauses unité positive = une assertion = un fait : une clause définie sans littéral négatif.

Exemple :

```
content(jean).
```

- Clauses unité négative = but : une clause définie sans littéral positif.

Exemple :

```
~content(pierre)
```

- Une clauses vide : une clause définie négative sans aucun littéral (notée ⇨).
- Une clause conditionnelle : A <- B (règle)
- Une clause bi-conditionnelle : A <=> B
- Un programme défini : un ensemble de clauses définies.
- Un programme non-définie : un programme qui n'est pas défini

- La classe des programmes définis constitue la classe la plus fondamentale. Les autres programmes posent certains problèmes théoriques plus complexes et leur implantation est plus complexes.

- Pour obtenir certaines formes de clauses voulues, on peut procéder à des réarrangement des littéraux à l'intérieur des clauses.

IV.2- Réarrangement des littéraux

- La forme générale d'une clause traitée par la résolution est la forme conditionnelle :
  disjonction de littéraux <- conjonction des littéraux

- Les clauses dont on dispose ne sont pas toujours sous la forme voulue et il faut procéder à des réarrangements.

On peut par exemple réécrire

A | B | ~C | ~D | ~E
③ A | B | ~(C & D & E)
③ A | B **<-** C & D & E

- On peut déplacer les littéraux de part et d'autres de "<-" :

A | B | ~C | ~D | ~E
③ A | B | ~(C & D & E)
③ A | ~(~B & C & D & E)
③ A <- ~B & C & D & E
③ B <- ~A & C & D & E

Dans ces différentes formes, on peut démontrer A, démontrer B ou .... selon le problème que l'on a à résoudre.

- On a **W ∫ W <- vrai**dans toute interprétation
  On peut donc réécrire la phrase ci-dessus
  ③ A | B | ~C | ~D | ~E<- vrai

- On a **~W ∫ faux <- W** dans toute interprétation
  On peut donc réécrire la phrase ci-dessus
  ③ faux <- ~A & ~B & C & D & E

- Une clause vide : **faux <- vrai** (le seul cas où la forme conditionnelle est fausse)

- Les réarrangements répondent par fois aux besoins spécifiques de formulation d'un problème :

Exemple : pour "A | B", on peut écrire
A | B ∫A <- ~B i.e. pour prouver A, il faut prouver que B est fausse
A | B ∫B <- ~Ai.e. pour prouver B, il faut prouver que A est fausse

Ces deux formalisations sont, d'un point de vu opérationnel, chacune l'inverse de l'autre mais elles sont logiquement équivalentes (identiques à A | B). On en choisira une selon le problème à résoudre, c'est à dire, lorsqu'il nous importe de prouver A ou de prouver B..

- Le processus de déplacement d'un littéral de part et d'autre de "<-" est comme l'application d'une règle d'inférence appelée la **transposition de littéraux**.
- La règle informelle de transposition dans une clause :

$$X_1 \mid X_2 \mid \ldots \mid X_n \quad \mathbf{<-} \quad Y_1 \,\&\, Y_2 \,\&\ldots\, \& \, Y_n$$

$$\int \quad X_1 \mid X_2 \mid \ldots \mid X_n \mid \mathbf{\sim Y_1} \quad \mathbf{<-} \quad Y_2 \,\&\ldots\, \& \, Y_n$$

$$\int \quad X_2 \mid \ldots \mid X_n \quad \mathbf{<-} \quad \mathbf{\sim X_1} \,\&\, Y_1 \,\&\, Y_2 \ldots \& \, Y_n$$

IV.3- La forme clausale de la logique

- Il y plusieurs raisons pour préférer la forme clausale au langage de la logique des prédicats du premier ordre :
  - La forme régulière des clauses facilite leur stockage en mémoire
  - Réduit le nombre de règles d'inférence.
  - Il suffit d'une seule règle pour "A <- B" par opposition à plusieurs règles nécessaires pour toutes les combinaisons de connecteurs.
  - Facilité l'association d'un sens à une phrase via une interprétation <u>procédurale</u>.

- Ils existent plusieurs méthodes de conversion.

Une classe de règles de conversion :

1- Réécrire la bi-conditionnelle en conditionnelle :

$$W_1 \Leftrightarrow W_2 \quad \Longrightarrow \quad (W_1 \leftarrow W_2) \,\&\, (W_2 \leftarrow W_1)$$

2- Réécrire la conditionnelle en disjonction :

$$W_1 \leftarrow W_2 \quad \Longrightarrow \quad W_1 \mid \sim W_2$$

3- Distribuer au maximum la négation en réécrivant

$$\sim(\exists X)\, W \quad \Longrightarrow \quad (\forall X) \sim W$$

$$\sim(\forall X)\, W \quad \Longrightarrow \quad (\exists X) \sim W$$

$$\sim(W1 \mid W2) \quad \Longrightarrow \quad \sim W1 \,\&\, \sim W2$$

$$\sim(W_1 \,\&\, W_2) \quad \Longrightarrow \quad \sim W_1 \mid \sim W_2$$

$$\sim\sim W \quad \Longrightarrow \quad W$$

4- Distribuer au maximum la disjonction en réécrivant (dans l'ordre ci-dessous) :

$W \mid (W_1 \& W_2) \quad ==> \quad (W \mid W_1) \& (W \mid W_2)$

$W1 \mid (\forall X)\ W_2 ==> \quad (\forall X)\ (W_1 \mid W_2)$

$W1 \mid (\exists X)\ W_2 ==> \quad (\exists X)\ (W_1 \mid W_2)$

Dans les deux derniers cas, X ne doit pas figurer dans W_1

5- Distribuer au maximum les ∀ en réécrivant :

$(\forall X)\ (W_1 \& W_2) \quad ==> \quad (\forall X)\ W_1 \& (\forall X)\ W_2$

6- Remplacer toute occurrence de formules close (sans variable libre) par la fonction de Skolem :

$(\forall X_1)..(\forall X_n)\ (\exists Y)W(Y) \quad ==> \quad (\forall X_1)..(\forall X_n)W(\mathbf{f}(X_1,...,X_n)$

f est appelé la fonction de Skolem. C'est une nouvelle symbole fonctionnel. Si n=0, une constante "**c**" appelé la constante de Skolem remplacera f() :

$(\exists Y)W(Y) \quad ==> \quad W(\mathbf{c})$

7- Refaire 5 s'il y a lieu. On finit par obtenir une conjonction de disjonctions de clauses. Supprimer les seuls quantificateurs universels qui restent et produire un ensemble de clauses en supprimant les conjonctions.

8- Remplacer les "|~" par "<-". Par exemple ,

A | ~B devient A <- B.

A | ~B | ~C devient (A <- B) <- C i.e. A <- (B & C) .

Exemple : lectures quantifiées équivalentes des conditionnelles :

$(\forall X)\ (A <- B) \int (A <- (\exists X)\ B(X))$

On applique les règles de conversion à la partie droite :

$(A <- (\exists X)\ B(X)) \int (A \mid \sim(\exists X)\ B(X)) \int (A \mid (\forall X)\ \sim B(X)) \int$

$(\forall X)\ (A \mid \sim B(X)) \int (\forall X)\ (A <- B(X)) \int$ la partie gauche

- On remarquera que la "Skolémisation" (étape 6) ne préserve pas l'équivalence entre la forme clausale et la phrase initiale.

Par exemple, l'étape 6 convertit (∃X) p(X) en p(a). Supposons que le domaine d'interprétation est {0,1} et 'p' est la relation {X : X>0}. Cette interprétation est un modèle pour la phrase initiale mais pas pour p(a).

Concrètement, si une proposition est vraie pour un élément du domaine, il est claire que ceci n'implique pas que cette proposition se vérifié pour n'importe quel élément du domaine. Notons en revanche que l'inverse est vraie.

V. Eléments de la théorie de preuve

- La théorie de modèles considère l'assignation d'un <u>sens</u> aux phrases. Elle donne des informations sur les sémantiques du formalisme.
- La théorie de preuve considère la génération de phrases à partir des autres. Elle procure <u>un cadre opérationnel</u> pour le formalisme.

- Considérons la relation d'implication logique de la théorie de modèles et le langage L de la logique du premier ordre.

La relation **|=** décrit l'ensemble **:**

|= : {<S, s> tel que S est une sous-ensemble de L,
s est un membre de L et
tout modèle pour S est un modèle pour s}

- La théorie de preuve définit des relations sur L d'une manière tout à fait différente : au lieu de considérer <u>la valeur de vérité</u> d'une phrase dans le contexte d'une certaine <u>interprétation</u>, on considère <u>la dérivabilité</u> de la phrase dans le contexte d'un ensemble de <u>règles de dérivation</u> de phrases.

- On peut présenter ces règles de la façon suivante :
  " *d'un ensemble de phrases de telle sorte,*
  dériver une phrase de telle sorte"

Par exemple :

" de l'ensemble de chaînes {aX, abX}, X : chaîne de caractères,
dériver la chaîne ababX"

On peut donc prendre n'importe quel ensemble initial S de phrases de L et, par l'application de différentes règles d'une ensemble R, dériver une phrase nouvelle s de S et/ou d'autres phrases dérivées.

- L'ensemble d'un tel exercice décrit la relation de dérivabilité (notée par |-) caractérisée par R :

  |- : *{<S, s> tel que S est une sous-ensemble de L,*
  s est un membre de L et
  s est dérivé de S en utilisant R}

- En logique, les phrases initiales sont les <u>axiomes</u> et les phrases dérivées sont les <u>théorèmes</u> (conséquences syntaxiques) et les règles de dérivation R sont les règles <u>d'inférences</u>.

- Une des plus connues (et fréquemment utilisées) de ces règles est **modus Ponens** :

{ (A <- B), B} |- A

- Ensemble, les axiomes et les règles R constituent un **système d'inférence**. Une <u>preuve</u> dans un tel système est la séquence $<s_1,...,s_n>$ où s_i est soit un axiome, soit une phrase dérivable par R d'un sous-ensemble des membres précédents de la séquence. La séquence $<s_1,...,s_n>$ est la **preuve** du théorème s_n (preuve par <u>dérivation</u> ou par <u>déduction</u>).

Exemple : soit un système d'inférence avec :

l'alphabet = {a,b,c}, axiome = a

R = {Xaba <- Xa, c <- bab}

La preuve de *acaba* est apportée par la séquence :

< <u>a</u> => <u>a</u>ba => a<u>ba</u>ba => ac<u>a</u> => acaba >

- Les axiomes et l'ensemble des théorèmes dérivables à l'aide de R constituent une <u>théorie</u>.
- Une théorie est <u>consistante</u> si et seulement s'il n'y a aucune phrases s telle que (s & ~s) soient dans la même théorie; sinon la théorie est <u>inconsistante</u>.

Exemple : avec modus ponens, on peut montrer que la théorie résultante des théorèmes suivantes est inconsistante :

~A <- B
B <- C
A <- C
C <- D
D.

car <D, C, A> est une preuve de A et <D, C, B, ~A> est une preuve de ~A.

- Notons que tous ces concepts sont <u>syntaxiques</u> (basés seulement sur la syntaxe des phrases) et il n'y a aucune notion d'interprétation (sens).

V.1- Utilisation de l'implication dans la résolution de problèmes

- L'approche de la résolution de problèmes en logique :
  - On a en entrée un ensemble S de phrases qui exprime nos connaissances sur le domaine du problème;
  - On demande au solveur de produire en sortie une phrase s exprimant une conclusion sur le domaine du problème.

- Le solveur doit produire les conclusions de telle sorte que :

  Si S exprime correctement les connaissances
  Alors s exprime correctement les conclusions

Par "correctement exprimé", on indique <u>une</u> "interprétation attendue" **I** reliant les objets et les relations du domaine aux termes et aux prédicats inclus dans les phrases.

- On interprète ensuite les conclusions **s**. En termes de la théorie de modèles, le minimum que nous demandons est :

  Pour <u>toute</u> interprétation <u>attendue</u> I,
  Si I est un modèle pour S
  Alors I est un modèle pour s.

- Mais, nous ne pouvons pas exprimer une "interprétation attendue". Cette interprétation est celle que nous avons en tête et concerne le monde réel que la machine ne sait reconnaître ni traiter : elle ne traite que des objets syntaxiques.

Nous devons donc obtenir (S |= s):

Pour <u>toute</u> interprétation <u>possible</u> I,
Si I est un modèle pour S
Alors I est un modèle pour s.

C'est à dire (S |= s)

- Nous verrons qu'en pratique, il n'est pas nécessaire de considérer <u>toutes</u> les interprétations mais simplement une seule définie sur les objets syntaxiques; les autres interprétation étant <u>isomorphes</u> à celle-ci (deux interprétations sont isomorphes si elles ont la même structure).

V.2- Utilisation de la preuve dans la résolution de problèmes

- Le mécanisme de résolution en programmation logique est fondamentalement un mécanisme inférentiel de preuve de théorèmes.
- Lorsque nous utilisons la résolution, nous sommes intéressés à caractériser le calcul par la manipulation <u>orientée inférence</u> des connaissances. C'est à dire, les interpréteurs utilisés sont des démonstrateurs de théorèmes. La théorie de preuve nous précise les capacités et les limites de ces démonstrateurs.
- Lors d'utilisation d'un système d'inférence pour obtenir une phrase (théorème) s, réponse à notre problème à partir d'un ensemble S d'axiomes, nous voulons que s soit <u>logiquement</u> impliquée par S afin d'être sûr de la justesse de la réponse par rapport à une <u>interprétation attendue</u> que l'on ne peut pas caractériser.

- Exemples d'inférences :

- {A,B} |- A <- B est <u>correcte</u> c'est à dire {A,B} |= A <- B.
  Il suffit de regarder la table de vérité de "<-"

- {(A | B) & (~A | C)} |- B | C est <u>correct</u> car
  {(A | B) & (~A | C) |- {(A <- ~B) & (C <- A) |- (~B <- C) |- (B | C)

- {(A <- C) & (B <- C)} |- A <- B est <u>incorrecte</u> c'est à dire:
  {(A <- C) & (B <- C) |≠ A <- B.

L'incorrection se démontre en développant les phrases (e.g. A=C=Faux, B=Vrai). On peut également la constater par l'exemple suivant : *{maison <- argent) & (voiture <- argent) |≠ (maison <- voiture).*

- L'obtention de réponses justes est une affaire de règle d'inférence. Pour obtenir une réponse "juste", la règle d'inférence doit être "juste" (correcte).

- Pour une règle juste, la relation de dérivabilité déterminée par une règle d'inférence est un sous-ensemble de la relation d'implication logique :
  ∀S,s, S|=s si S |-s

- Les systèmes d'inférence qui utilisent des règles justes produisent des "inférences justes" ou "valides".
- Exemples de règles justes :
  - **modus ponens** {B, A <- B} |- A
  - **modus tolens** {~A, A <- B} | ~B

- Un exemple de règle non-juste mais pourtant répandue par le dicton "il n'y a pas de fumée sans feu" :
  - modus **populi** {A, A <- B} |- B

Par exemple : *{sol mouillé, sol mouillé <- il pleut} |- il pleut*
On peut montrer qur *modus populi* est incorrecte, e.g. avec A=Vrai, B=Faux).

- On veut également que la règle d'inférence soit assez puissante pour dériver s de S lorsque S implique s. Sinon, le problème peut avoir des solutions que le solveur ne trouvera pas. La règle doit donc permettre :
  ∀S,s S |- s si S |= s
  On dira qu'un telle règle est **complète**.

- En résumé, pour un système d'inférence juste et complet, la relation de dérivabilité implication coïncident (|- ∫ |=):
  ∀S,s S |- sssi S |= s

- Est-ce que de tels systèmes existent ? La réponse dépend du langage considéré:

- pour la logique propositionnelle : **Oui**
- pour la logique des prédicats du premier ordre : **Oui**
- pour la logique d'ordre deux : **Non**

- Notons qu'un système d'inférence n'est pas en soi un algorithme. Il décrit les règles d'inférences mais ne précise pas de stratégie pour leur application.
- Disposer de ces systèmes est important mais pas suffisant : Il faut également disposer d'un algorithme capable de générer toutes les dérivations pour résoudre les problèmes.
- Le couplage d'un système d'inférence et d'une stratégie donne une procédure de preuve. La préservation de la complétude et la justesse de cette procédure dépend de la stratégie employée.

V.3- Validité des programmes

- Lorsque l'on écrit un programme logique, il faut veiller à écrire des clauses valides : des clauses vraies dans toute interprétation. Un exemple de clause valide est A | ~A ∫ A <- A.

- Soit S l'ensemble des clauses d'un programme et s la question. On veut vérifier si S |= s.

- Le lien entre la validité et l'implication se formalise de la manière suivante :

  Pour tout ensemble de phrases $S = \{s_1,..., s_n\}$ et toute phrase s,

  $S \models s$ *si-et-seulement-si* $S - \{s_i\} \models (s \leftarrow s_i)$ $\forall i$

  C'est à dire, si I est un modèle pour S et pour s (par l'implication), I est un modèle pour tout s_i, pour (s <- si), ...

- On peut itérer le processus et obtenir la forme suivante :
  S |= s si-et-seulement-si Ø |= s <- s_1 & s_2 & ... & s_n

- Toute interprétation est un modèle pour Ø et les phrases impliquées par Ø sont justement les phrases valides. On peut donc dire :
  S |= s si-et-seulement-si s <- s_1 & s_2 & ... & s_n est valide

- C'est ce que nous exprimons en écrivant des conditionnelles :
  si l'on veut *(q & r) |= p*, on écrit par exemple *p <- q & r*, sous entendu : *p <- q & r* (ou *p <- q*, ...) est valide.

- Notons de plus qu'une phrase est valide si sa négation est non-satisfiable (inconsistante) :
  Ø |= A ∫ Ø |= A | faux ∫ ~A |= faux.

- Ces observations permettent d'expliquer les connections entre l'organisation d'un programme logique et ses fondements logiques.
  De plus, il y a des résultats importants sur la décidabilité de la validité montrant les limites des solveurs orientés implication. Nous utiliserons ces résultats lors de l'exposé des procédures de résolution.

VI. Programmation avec la logique

- Quand on écrit des programmes logiques, on a "en tête" un domaine du monde réel que l'on veut modéliser.

- Puisque les machines ne savent pas opérer sur les domaines réels, nous représentons le monde réel à l'aide des <u>symboles</u> que la machine peut comprendre.

- En programmation logique, ce domaine est appelé l'**univers de Herbrand**, noté $\mathbf{U_H}$, déterminé par le langage L dans lequel on exprime le programme (ce n'est pas le langage de programmation).

VI.1- Le domaine de Herbrand

- U_H consiste en tous les termes clos que l'on peut construire en utilisant les constantes et les symboles fonctionnels du langage L. Ces termes sont les données de base avec lesquelles les programmes logiques travaillent pour construire des preuves (dérivations).

Exemple :

Soit l'alphabet de L constitué de la constante '0' et des symboles fonctionnels 'p' et 's'. A l'aide de ces éléments, on peut écrire :

int(0)
int(s(X)) <- int(X)

A l'aide de ce programme, on peut (essayer de) prouver

int(s(s(0))) ou int(p(p(0)))

Ici,
U_H = {0, p(0), s(0), s(s(0)), p(p(0)), s(p(0)), p(s(0))...)}

- Pour cet exemple, on peut imaginer que le monde réel que l'on veut représenter est l'ensemble des entiers :
  Z = {..., -2, -1, 0, +1, +2,...}.

On associe donc :
- 0 à 0
- 'p' à la fonction prédécesseur (i.e. p(0) est interprété par -1)
- 's' à la fonction successeur (i.e. s(0) est interprété par +1)
- la relation 'int' à la relation Z

- Lorsque l'on analyse le comportement d'un programme logique, on n'a (presque) pas besoin de connaître le monde réel envisagé par le programmeur : nous raisonnons syntaxiquement sur l'univers de Herbrand. Le programme logique n'est alors qu'un "générateur de symboles de Herbrand".

- L'U_H est construit en considérant les constantes et les symboles fonctionnels précisés dans L ainsi que dans les questions. Si le langage L n'est explicitement précisé, on se contente des clauses du programme (et des questions).

Ainsi, pour le programme précédent, la question *int(a)*, enrichit l'univers de Herbrand de la constante 'a'.

- La **base de Herbrand** associée à un programme est notée B(P). C'est l'ensemble de tous les atomes clos dont le symbole de prédicat apparaît dans le programme P et dont les arguments appartiennent à l'U_H. On tient compte de l'arité des symboles de prédicats.

Pour l'exemple ci-dessus, la base (infinie) de Herbrand est construite en combinant 'int' avec tous les éléments de U_H :

{int(0), int(s(0)), int(p(s(0))),...}

Les conséquences du programmes sont des atomes dans B(P).

- L'U_H est infinie dès qu'au moins un symbole fonctionnel apparaît dans le programme (ou dans l'alphabet de L). Si cet alphabet est vide, on se donne une constante arbitraire (par exemple 'a'). La base de Herbrand étant construite à partir de U_H, celle-ci sera infinie si U_H l'est.

VI.2- Systèmes d'inférence

- Deux méthodes de preuve courantes : Par confirmation et par Contradiction.

VI.2.1- Preuve par Confirmation / Contradiction

- Pour la propositionnel (sans variable), la résolution est simple. L'U_H étant fini, il suffit d'appliquer par itération la règle modus ponens pour construire une **preuve par confirmation**.

- Par exemple, avec le programme P suivant et U_H ={jean,marie,logique) :

```
aime(jean, X) <-   aime(X, logique)
aime(marie, logique)
```

On obtient G(P), le programme P clos, en remplaçant les variables pour tout élément de U_H :

```
aime(jean, jean)        <- aime(jean, logique)
aime(jean, marie)       <- aime(marie, logique)
aime(jean, logique)     <- aime(logique, logique)
aime(marie, logique)
```

La preuve par confirmation démontre les conséquences suivantes :

aime(marie, logique) et aime(jean, marie)

- Une autre manière de faire est d'apporter une **preuve par contradiction** en appliquant la règle modus tolens.

- Apporter la preuve par contradiction de la proposition A dans un programme P, c'est de prouver :

$$P \approx \{\sim A\} \models \Rightarrow \quad faux$$

- Nous avons vu que

$$S \models s \quad \text{si-et-seulement-si} \quad S\text{-}\{s_i\} \models (s \leftarrow s_i) \quad \forall i$$

Prenons le cas où S = P ≈ {~A} et s = ⇨ = faux. Nous avons :

P ≈ {~A} |= ⇨ si-et-seulement-si P |= (faux <- ~A)

i.e. P ≈ {~A} |= ⇨ si-et-seulement-si P |= A
i.e. P ≈ {~A} |- ⇨ si-et-seulement-si P |- A
Inversement, pour prouver A dans un programme P par un système d'inférence correct et complet, on doit avoir :

	P |= A		
∫	P |= A | faux	car A ∫ A | faux	∀A
∫	P ≈ {~A} |= ⇨	⇨ = faux	
∫	P ≈ {~A} |- ⇨		

Donc, au lieu de dériver A de P, on peut dériver une contradiction de P & ~A.

- Lorsque P est défini et clos et A atomique et clos, modus tolens et la transposition des littéraux suffisent à construire un nouveau système d'inférence. La règle modus tolens est {~A, (A <- B)} |- ~B. En particulier, pour B=vrai, on a : {~A, A} |- ⇨.

- Pour l'exemple ci-dessus, on démontre *aime(jean, marie)* à l'aide de *modus tolens* :
  {~aime(jean, marie) & (aime(jean, marie) <- aime(marie, logique))}
  |- ~aime(marie, logique)

  {~aime(marie, logique) & aime(marie, logique)} |- ⇨

- Cette dérivation traite uniquement la contradiction d'une proposition particulière. Ce qui est souvent le cas en programmation logique où l'on est plutôt intéressé par la démonstration d'une conséquence que par celle de <u>toutes</u> les conséquences d'un programme.

- Cette dérivation peut être vue comme une preuve par la séquence :

< ~aime(jean, marie), ~aime(marie, logique), ⇨>

- Une preuve terminée par la contradiction "⇨" est appelée une **réfutation**. Nous avons utilisé les clauses de P pour *réfuter* la proposition *~aime(jean, marie)* plutôt que d'affirmer la proposition *aime(jean, marie)* comme cela a été fait avec modus ponens.

- La réfutation est le mécanisme de base de la majorité des langage de programmation logique.

VI.2.2- Preuve par contradiction : procédure de Herbrand

- Une des plus importantes théorème en logique est le théorème de Herbrand :

  Une contradiction ⇨ peut être dérivée d'un ensemble de clauses $\mathcal{P} \approx \{\sim\mathcal{A}\}$
  si-et-seulement-si
  elle peut être dérivée d'un sous-ensemble <u>fini</u> d'instance close $\mathcal{G}(\mathcal{P} \approx \{\sim\mathcal{A}\})$

  Ce qui a conduit aux premières procédures de démonstration de théorème utilisant le principe :

  Remplacer les clauses par leur instanciation closes
  Chercher une contradiction

- Le point crucial dans le théorème de Herbrand est le mot **fini** car elle assure l'existence d'un algorithme effectif par lequel on démontre une contradiction et, indirectement, on démontre que A est conséquence de P.

- Ce théorème permet de répondre à une nécessité fondamentale pour montrer que *P implique A*:

  Chaque modèle de $\mathcal{P}$ est un modèle pour $\mathcal{A}$.

  Si on prouve que P ≈ {~A} n'a pas de modèle, on aura démontré que tout modèle de P est un modèle pour A.
  Notons qu'un programme peut avoir une infinité de modèles.

- Le théorème de Herbrand a été utilisé et reformulé de multiples manières. Par exemple, en terme de <u>satisfaction</u>, on a :

  Un ensemble de clauses $\mathcal{P} \approx \{\sim\mathcal{A}\}$ est non-satisfiable
  si-et-seulement-si
  un sous-ensemble <u>fini</u> de $\mathcal{G}(\mathcal{P} \approx \{\sim\mathcal{A}\})$ est non-satisfiable.

  Ce théorème est utilisé pour montrer que la *résolution* est complète.

- Une autre variante de ce théorème est le théorème de *compacité* que nous utiliserons en programmation avec contraintes :

  Un ensemble de clauses est satisfiable

si-et-seulement-si
tout ses sous-ensembles finis sont satisfiables.

VI.2.2.1- Saturation / Résolution

- Un algorithme naïf basé sur le théorème de Herbrand devrait constituer une séquence $<g_1,g_2,..,g_n>$ à partir de G(P ≈ {~A}); puis pour k=1..n, chercher une contradiction sur la séquence finie $<g_1,g_2,..,g_k>$. D'après le théorème de Herbrand, il existe un k tel que après k étapes, une contradiction est trouvée.

- Ce type d'algorithme est appelé une procédure de **saturation**. On peut remarquer que ce type d'algorithme est particulièrement inefficace pour des besoins de programmation. Une meilleure procédure basée sur le théorème de Herbrand est la *résolution*.

VI.2.3- Résolution : cas de la logique propositionnelle

- Une étape de la résolution procède comme par la règle d'inférence modus tolens en considérant deux clauses quelconques (appelées parents) de la forme :

  𝔸 | d_autres_littéraux et
  ~𝔸 | encore_d_autres_littéraux

et infère un **résolvant** :

d_autres_littéraux | encore_d_autres_littéraux

Exemple :

de mere | <u>pere</u> | ~parentet male | <u>~pere</u>
on infère mere | male | ~parent

- On constate que la forme clausale des phrases rend la résolution plus claire et met en évidence la transitivité de "<-" que nous utilisons :

  { B <- <u>A</u> , <u>A</u> <- C } |- B <- C

Par exemple, de : *𝔸 <- ~ d_autres_littéraux*
et *encore_d_autres_littéraux <- 𝔸*
on infère *encore_d_autres_littéraux <- ~d_autres_littéraux*

- Pour l'exemple, de male <- pere
  pere <- parent & ~mere
  on infère male <- parent & ~mere
- Avec la forme clausale, il y a un seul résolvant possible car les parents forment une seule paire complémentaire.

- Les propriétés importantes de la *résolution* sont les suivantes :
  - La résolution est **juste**. Chaque résolvant est une implication logique des clauses parents.
  - Le résolvant est la clause vide ⇨ si les parents sont des clauses unités. Dans ce cas, un des parents est de la forme A et l'autre de la forme ~A.
  - Pour tout ensemble de clauses de Horn, la clause vide est dérivée par la résolution si cet ensemble est non-satisfiable. On dit alors que la résolution est **réfutation-complète.**
  - On peut montrer que le résolvant de deux clauses de Horn est une clause de Horn.

- La résolution en programmation logique est exploitée sous la forme suivant :
  - Les connaissances sont présentées sous forme de clauses définies
  - La question A est formulée sous la forme ~A
  - On a P |= A si-et-seulement-si P ≈ {~A} est non-satisfiable.
  C'est à dire : P ≈ {~A} |= ⇨ par la résolution grâce à la justesse et la complétude-réfutationnelle.

VI.2.4- Résolution : cas de la logique des prédicats

- On introduit des variables.

VI.2.4.1- Application et composition de substitutions

- Une **substitution** est un ensemble fini de **remplacements** d'une variable U par le couple U/t où t est un terme.
- Une substitution est de la forme $\theta=\{U_1/t_1,..., U_n/t_n\}$
- L'application de θ à une formule W remplace simultanément chaque variable U_i de W par t_i. Le résultat est appelé une **instance** de W noté Wθ.

Exemple :

- W = aime(X,Y), θ={X/jean, Y/logique) ❹ Wθ= aime(jean, logique)
- θ'={X/Y, Y/logique) ❹ Wθ'= aime(Y, logique)

si on applique θ' à Wθ' ❹ Wθ'= aime(logique, logique)

- Dans une substitution $\theta=\{U_1/t_1,..., U_n/t_n\}$, les variables U_i sont <u>distinctes</u>. Ce que l'on appelle la propriété de la <u>fonctionnalité</u> (l'application de θ est une fonction).
- Une substitution doit être **idempotente**. C'est à dire $W\theta = (W\theta)\theta$. Ce qui exige qu'une variable U_i ne doit pas figurer dans un terme t_j. Par cette condition, la substitution θ' ci-dessus n'est pas légale.

- L'opération importante sur les substitutions est la **composition**. Composer deux substitution θ et σ notée θ°σ produit une substitution δ telle que $(W\theta)\sigma = W\delta$.
- Pour $\theta_1 = \{U_1/t_1,..., U_m/t_m\}$ et $\theta_2 = \{V_1/s_1,..., V_n/s_n\}$, la composition est $\theta_1 \circ \theta_2 = \{U_1/t_1\theta_2,..., U_m/t_m\theta_2\} \approx \{V_j/s_j$ tel que $V_j \square \{U_1,..., U_m\}\}$

- <u>Explication de la composition :</u>

  L'application de θ_1 à une formule W remplace les U_i par t_i et produit $W\theta_1$. θ_2 doit remplacer les V_j figurant dans t_i que θ_1 avait introduites dans W. Elle doit également remplacer les V_j de W que θ_1 <u>n'a pas</u> remplacées.

- Les compositions ne sont pas toujours légales car elle ne sont pas toutes idempotentes. Cependant, la composition possède les propriétés suivantes :
  - $\theta_1 \circ \theta_2$ est fonctionnelle
  - $W(\theta_1 \circ \theta_2) = W(\theta_1)\theta_2$
  - $(\theta_1 \circ \theta_2) \circ \theta_3 = \theta_1 \circ (\theta_2 \circ \theta_3)$ Associativité

  Notons que la composition n'est pas commutative.

Exemple :

$\theta_1 = \{X/f(a,Y), W/f(U,Z)\}$

$\theta_2 = \{X/f(a,a), Y/b, V/c\}$

$W = p(V, X, Y, W)$

On a :

$\theta_3 = \theta_1 \circ \theta_2= \{X/f(a,b), W/f(U,Z), Y/b, V/c\}$

$W\theta_3 = p(c, f(a,b), b, f(U,Z))$

$W\theta_1 = p(V, f(a,Y), Y, f(U,Z))$
$(W\theta_1)\theta_2 = W\theta_3 = p(c, f(a,b), b, f(U,Z))$

On peut facilement vérifier que $(W\theta_1)\theta_2 \neq (W\theta_2)\theta_1$

VI.2.4.2- Extension de la résolution aux prédicats du 1er ordre

- On introduit la substitution pour étendre la résolution propositionnelle à la résolution des prédicats. Dans ce cas, la substitution est considérée comme une règle d'inférence appelée l'<u>instanciation universelle</u> :

  $(\forall X)\ W(X) \vdash (\forall Y_1)...(\forall Y_m)\ W(X)\ \{X/t\}$ Pour tout terme t
  où $Y_1..Y_m$ sont les variables figurant dans t.

- Exemple : soient les clauses parents
  ~aime(jean, Y) | ~femelle(Y) | ~tante(Y, Z)
  aime(X, marie) | ~adore(X, marie)

La résolution construit *θ = {Y/marie, X/jean}* de sorte que les deux littéraux 'aime' soient complémentaires. On a alors les instances des clauses parents

~aime(jean, marie) | ~femelle(marie) | ~tante(marie, Z)
aime(jean, marie) | ~adore(jean, marie)

et on obtient le résolvant :

~femelle(marie) | ~tante(marie, Z) | ~adore(jean, marie)

- Supposons les clauses parents ci-dessus et le domaine
  U_H = {jean, marie, paul}.

La base de Herbrand pour les prédicats aime(X,Y) est l'ensemble fini :
B(P) = { *aime(jean, jean),* ***aime(jean, marie),*** *aime(jean, paul),*
aime(marie, jean), *aime(marie, marie),* aime(marie, paul),
aime(paul, paul), *aime(paul, marie),* aime(paul, jean)}.

Considérons **~aime(jean, Y)** et **aime(X, marie)**.

Pour rendre ces deux littéraux complémentaires, il faut chercher dans B(P) deux atomes :

- L'un ayant *jean* en premier argument (la première ligne dans B(P)) et
- L'autre *marie* en second (la deuxième colonne dans B(P)).

Il suffit alors de prendre l'intersection de ces deux sous-ensembles de B(P). Ce qui est représenté par $\theta = \{Y/marie, X/jean\}$.

- Cette substitution doit être *la plus générale* pour considérer la totalité de l'intersection. La substitution θ rend W_1=aime(jean, Y) et W_2=aime(X, marie) identiques (ou *unifiés*).

  θ est appelée l'**unificateur le plus général** de W_1 et de W_2 : on a $W_1\theta = W_2\theta$. La substitution θ est le plus général si pour tout autre unificateur σ de W_1 et W_2, $W_1\sigma$ est une instance de $W_1\theta$.

- D'une manière générale, on peut construire un démonstrateur de théorèmes qui exploite toutes les manières pour obtenir la clause vide.

Prenons l'exemple du programme :

```
C1 : ~p(X)
C2 : p(Y) | ~q(Y)
C3 : q(a)
```

L'ensemble {C1, C2, C3} est non-satisfiable et l'on peut démontrer une contradiction de différentes façons (par différents **graphes de résolutions**) :

- Cette approche est de toute évidence inefficace. Les démonstrateurs pratiques utilisent une stratégie qui construit plutôt le graphe de gauche. Dans ce cas, les unificateurs construits sont dans l'ordre θ_1={X/Y} puis θ_2={X/a, Y/a}.

- Pour préserver la complétude de la réfutation, la procédure de réfutation construit l'unificateur le plus général de deux littéraux complémentaires.

VI.3- Programmation avec la résolution

- Lorsque nous employons la programmation logique pour la résolution de problèmes, nous travaillons avec

- <u>un programme</u> : ensemble de clauses définies écrites sous la forme **A <- B** où toute variable est <u>quantifiée universellement</u>; ce qui permet d'omettre les quantificateurs;
- <u>une question</u> de la forme **<- q** dont les <u>quantificateurs universels</u> sont omis.

Exemple : le programme de concaténation de listes :

C1 : concat(nil, W, W) C2 : concat(U.X, Y, U.Z) <- concat(X, Y, Z)

La notation *X.Y* dénote une liste dont la tête est X et dont la queue est Y.

La question est habituellement une clause négative de la forme :

<- concat(t_1, t_2, t_3) où t_1, t_2 et t_3 sont des termes.

Le "<-" est une convention d'écriture pour remplacer le connecteur "~".

Si la question contient des variables, elle sera de la forme :

$$\int \begin{array}{l} (\forall X_1) \ldots (\forall X_n) \ \sim concat(t_1, t_2, t_3) \\ \sim(\exists X_1) \ldots (\exists X_n) \ concat(t_1, t_2, t_3) \end{array}$$

- Pour simplifier, notons les formules par leur <u>fermetures universelle</u> et <u>existentielle</u>. C'est à dire, notons

  $(\forall X_1) \ldots (\forall X_n)$ W par $\forall$W

  et $(\exists X_1) \ldots (\exists X_n)$ W par $\exists$W.

On notera donc une question de la forme <- A_1 & ... & A_n
sous la forme générale $\sim\exists(A_1 \ \& \ldots \& \ A_n)$

- La question étant une négation, la réfutation s'applique pour démontrer la contradiction :

$$P \approx \{\sim\exists(A_1 \ \& \ldots \& \ A_n)\} \vdash \rightarrowtail$$

- Pour le programme ci-dessus, on a la question

  Q1 : <- concat(V.L1, L2, V.a.nil)

On obtient le graphe suivant :

- On peut faire correspondre un **arbre de recherche** à ce graphe **:**

On constate que dans θ1, l'unificateur le plus général de Q1 et de C2, les variables U1, X1, Y1 et Z1 ont remplacées U, X, Y et Z. Cette opération appelée **renommage** consiste à appliquer une <u>substitution</u> à la clause C2 de sorte que cette clause n'ait pas de variable commune avec Q1.

Les variables d'une clauses étant *locales* à la clause, le renommage permet de remplacer les variables d'une clause parent par des variables "fraîches " pour lever toute confusion entre les variables ayant le même nom dans le résolvant et la clause parent.

VI.3.1- Extraction des réponses

- Un arbre de recherche a pour racine la question. La dérivation commençant par cette question est appelée un **calcul**. Un arbre de recherche peut contenir des **calculs finis** et des **calculs infinis** (e.g. boucles).
- Un calcul fini peut être un succès ou un échec. Un calcul fini avec succès termine avec la clause vide (⇨) et représente une réfutation. Un calcul fini avec échec termine avec un résolvant différent de la clause vide.
- Un arbre fini contient un nombre fini de calculs; chaque calcul ayant une longueur finie. Un arbre d'échec fini est un arbre fini dont chaque calcul est un échec fini.
- Une question à la racine d'un arbre de recherche réussit si et seulement si l'arbre de recherche contient au moins un calcul avec succès; sinon, la requête échoue. Dans l'exemple précédent, le calcul de Q1 réussit de deux manières.

- Un calcul réussi émet une réponse construite à l'aide de la séquence $<\theta_1, ..., \theta_n>$ d'unificateurs; n étant la longueur du calcul. Soit $\theta = \theta_1 \circ ... \circ \theta_n$.
- Si une question $\sim\exists(A_1 \& ... \& A_n)$ réussit pour un programme P, le fait d'avoir obtenu une réfutation veut dire :

$$P \models \exists((A_1 \& ... \& A_n)\theta).$$

Dans l'exemple $\theta = \theta_1 \circ \theta_2 \circ \theta_3 \circ \theta_4$. On a

$$P \models \exists((concat(V.L1, L2, V.a.nil)\theta).$$

- Pour obtenir une réponse (valeur des variables), nous sommes seulement intéressés par les valeurs des variables figurant dans la question de départ.

Pour l'exemple, on obtient $P \models (\exists V)concat(V.a.nil, nil, V.a.nil)$.

- Lorsque $(A_1 \& ... \& A_n)\theta$ n'est pas clos, il représente la réponse la plus générale qui est la fermeture universelle de la formule; laquelle est impliquée par P. Pour l'exemple ci-dessus, on a la **réponse finale** :

$(\forall V)concat(V.a.nil, nil, V.a.nil).$

Cette réponse représente une théorème qui résume une classe de tuples de la relation *concat*; chacun étant une réponse à la question Q1.

VII. La SLD-Résolution

- Il existent plusieurs façons de dériver la clauses vide par le résolution :
  - Il y plusieurs manières de choisir une clause parent
  - Il y plusieurs manières de choisir un littéral dans le résolvant.

- Le but est de rechercher une stratégie à combiner avec la résolution pour rendre la résolution efficace sans sacrifier la complétude.
- Une des méthodes de résolutions appliquée aux clauses définies est la **SLD-Résolution**.

- Dans **SLD** :

- **L** pour linéaire : la résolution utilise le résolvant le plus récent (le dernier appelé la *clause centrale*) et utilise une clause du programme comme parent (appelée *clause adjacente*).
- **D** pour défini : le programme contient des clauses définies
- **S** pour sélection : une règle particulière appelée la **règle de calcul** choisit un littéral dans le dernier résolvant.

- La SLD-Résolution reste réfutation-complète sous ces restrictions. Elle produit toutes les dérivations quelque soit la règle de sélection. Cette propriété est appelée l'**indépendance de la règle de calcul** et constitue un élément important du formalisme.

- Exemple : soit le programme et la question suivante :

Q1 : <- gp(amenda, Z).	gp : grand parent
C1 : gp(X,Z) <- p(X, Y) & p(Y, Z).	p : parent
C2 : p(amenda, christophe).	
C3 : p(christophe, brigitte).	
C4 : p(christophe, emile).	

- Un premier arbre de recherche s'obtinet si l'on choisit de résoudre tout littéral du résolvant. Cette règle conduit à des calculs redondants :

- Une deuxième règle de calcul peut traiter les littéraux du résolvant de gauche à droite (comme en Prolog). On obtient comme arbre la branche la plus à gauche de l'arbre ci-dessus.

- Finalement, en traitant les littéraux du résolvant de droite à gauche, on obtient un autre arbre qui est le sous-arbre droit de l'arbre ci-dessus.

- Chacun de ces arbres est appelé un arbre SLD. Dans tous les trois cas ci-dessus et, en accord avec le principe de l'indépendance de la règle de calcul, l'ensemble de réponses obtenu est :

  {gp(amenda, brigitte), gp(amenda, emile)}

Le choix de la règle (le choix de littéral dans le résolvant) ne peut affecter que l'efficacité de la procédure de résolution. Remarquons que la second règle (à la Prolog) est plus efficace (dans cet exemple) que les autres.

Connaissant la règle de calcul du système d'inférence, le programmeur peut mieux organiser la partie droite des clauses de son programme.

- Notons que dans les trois résolutions, les clauses C1-C4 ont été choisies dans leur ordre textuel. Nous verrons que ce choix est d'une importance capitale.

- Pour formaliser la résolution :
  ⑩ **Q : <- A_1 &...& A_i &...& A_n = ~A_1 |...| ~A_n** le résolvant

  ¶ **C : A <- B_1 &...& B_m** une clause du programme

On renomme les variables de C. Si A s'unifie avec A_i et produit θ, le nouveau résolvant est :

① Q' : <- (A_1 &...& A_{i-1} & B_1 &...& B_m & A_{i+1} &...& A_n) θ

On a :

- {Q, C} |= Q'
- Si Q' peut être résolue (réfutée) alors Q le peut.
- Si n=1 et m=0, alors Q'=⇨.

- Si l'on devait résoudre le problème ci-dessus par la procédure de Herbrand, on constitue d'abord la base de Herbrand (ensemble des littéraux clos). C'est à dire, toutes les formes closes de la question et des clauses.

On recherche ensuite un ensemble fini parmi tous ces littéraux pour lequel une contradiction est démontrable (par une règle simple telle que le modus tolens).

Pour le problème donné, il existe 2^{71} - 72 sous-ensemble possibles. On peut améliorer les performances en ne considérant que les ensemble contenant au moins une clause négative; ce qui élimine 2^{67} possibilités. Il reste encore un nombre important de possibilités que l'on peut réduire en analysant par exemple les arguments des clauses.

C'est ce type d'analyse qui est faite par la résolution par l'unification qui joue le rôle de filtre. On constate alors combien l'algorithme de résolution de J. Alan Robinson est important pour la démonstration automatique de théorèmes.

VII.1- Unification

L'unification de deux termes t1 et t2 consiste à trouver un unificateur le plus général θ tel que $t_1\theta = t_2\theta$. Parmi les algorithmes d'unification utilisés en programmation logique, celui connu sous le nom de l'algorithme de Robinson est le plus employé.

L'algorithme suivant produit un couple <θ, vrai> si l'unification réussit ou bien produit <_, faux> dans le cas d'échec.

```
Fonction unifier(t1,t2 : terme ; θ : substitution) retourne booléen =
cas
   1)   t1 = t2: retourne(vrai)
   2)   var(t1) :     θ := θ ∘ {t1 / t2} ; retourne(vrai)
   3)   var(t2) :     θ := θ ∘ {t2 / t1} ; retourne(vrai)
   4) const(t1) ou const(t2) : retourne(faux)
   5)   t1 et t2 des termes composés de même fonctor et de
                                                  même arité:
            soient    t1 = f(arg1,....,argn) et
                      t2 = f(arg'1,....,arg'n)
            alors
            retourne(□i=1..n unifier(substitution(argi,θ),
                            substitution(arg'i,θ), θ))
   6) autre: retourne(faux)
Fincas
Fin unifier
```

Exemples:

- t1= f(a,g(X,b))
  t2=f(Y,g(b,X) □ θ={Y/a, X/b}

- t1= p(X, a, f(b,f(Y,n)))
  t2= p(s,Y, f(Z,f(X,n))) □ Echec

- t1= entier(suc(suc(suc(0))))

t2= entier(suc(suc(1))) Echec

- Qu'obtient-on lors de l'unification de X et de f(X) ?

L'algorithme précédent produira la substitution {X/f(X)}; ce qui représente l'arbre infini f(f(f(f(...)...))). Le couple {X/f(X)} n'est pas une substitution légale en accord avec les conditions de bonne forme de construction des substitutions.

Pour éviter ce type de construction, il faut ajouter aux cas (2) et (3) de l'algorithme précédent un test (dit le test d'occurrence) qui vérifie qu'une variable n'est pas liée à un terme qui contient cette variable. Cependant, étant donné la fréquence d'utilisation de l'unification dans les moteurs Prolog, ceux qui effectuent ce test voient leur performance diminuée de près de 50%. A ceci, les Prolog modernes donne plusieurs réponses :

1- Rendre le test d'occurrence sélectif : le programmeur en fait la demande explicite (dynamiquement);

2- Mettre en place des algorithmes différents (cf. PrologII et son système de réécriture de termes);

3- Provoquer un échec à la détection de boucle lors de l'unification (cf. exemple ci-dessous)

4- Permettre la construction d'arbres infinis (cf. PrologIII, VM/Prolog...)

4- Demander au programmeur de veiller à ne pas écrire des programmes nécessitant le test occurrence.

Exemple : t1= f(X,Y,X)
t2= f(g(X), g(Y), Y).

Comme on peut le constater, l'algorithme d'unification peut rentrer dans une boucle lors de l'unification de t1 et de t2.

VII.2- Stratégie de recherche

- Concerne le développement de l'arbre de recherche.
- Les interpréteurs proposent en général une règle de calcul figée d'avance.
- On peut cependant implanter notre propre règle en écrivant un *méta-programme* s'interposant entre l'interpréteur et notre programme (cf. études de parcours de graphes deuxième année).
- La recherche peut être <u>déterministe</u> (avec une seule réponse = fonctionnel) ou <u>non-déterministe</u> (avec plusieurs réponses = relationnel).
- Habituellement, les programmes logiques nécessitent une recherche non-déterministe; ce qui implique l'emploi d'une <u>stratégie</u> de recherche.

- Il existe un certain nombre de stratégies de recherche. La stratégie classique des interpréteurs logique tournant sur un seul processeur est :
  - Séquentielle
  - Descendante
  - En profondeur d'abord
  - Avec retour-arrières

- La stratégie standard développe un arbre de recherche SLD selon le schéma suivant :

Cette stratégie étant exhaustive et excessive, elle ne garanti pas la terminaison. Elle termine si l'arbre est de profondeur finie.

- Lorsque le nombre de clauses du programme est fini, on peut envisager un développement "en largeur" permettant de trouver les réponses; ce qui n'empêche pas un calcul infini (si l'arbre contient des branches infinie, le calcul ne s'arrête pas après avoir donné toutes les réponses). Cependant, la

stratégie "en profondeur d'abord" est plus optimale et utilise le minimum de mémoire toute en préservant la complétude de la SLD-résolution.

VII.3- Règles de choix

- Il y a deux choix possibles dans la SLD-résolution : le choix d'un littéral dans le résolvant (appelé *la règle de calcul*) et le choix d'une clause du programme (appelé *la règle de recherche*)

- Pour un programme et une question, le choix de la règle de calcul détermine l'arbre SLD correspondant. En supposant la procédure de recherche ascendante et en profondeur d'abord, la règle de recherche détermine l'ordre dans lequel les noeuds (les calculs) sont générés dans l'arbre SLD. Elle concerne le choix des clauses dans le programme.

- Les exemples suivants (traitant du problème classique de recherche d'un chemin dans un graphe) montrent les différentes réponses que l'on peut obtenir d'un même programmes en variant les deux points de choix ci-dessus :

1 - Règle de calcul : choix du littéral le plus à gauche
Règle de recherche : choix des clauses dans l'ordre d'entrée
Programme :

```
p(X,Z) <- a(X,Z)
p(X,Z) <- a(X,Y) & p(Y,Z)
a(a,b)
a(b,c)
<- p(a,Z)
```

Réponses : on obtient un calcul fini avec Z=b, Z=c puis deux échecs.

2 - Règle de calcul : choix du littéral le plus à droite
Règle de recherche : choix des clauses dans l'ordre d'entrée
Programme :

```
p(X,Z) <- a(X,Z)
p(X,Z) <- a(X,Y) & p(Y,Z)
a(a,b)
a(b,c)
<- p(a,Z)
```

Réponses : on obtient un calcul infini avec Z=b, échec, Z=c puis une branche infinie.

3 - Règle de calcul : choix du littéral le plus à gauche
Règle de recherche : choix des clauses dans l'ordre d'entrée
Programme (ordre des clauses modifié) :

```
p(X,Z) <- a(X,Y) & p(Y,Z)
p(X,Z) <- a(X,Z)
a(a,b)
a(b,c)
<- p(a,Z)
```

Réponses : on obtient un calcul fini avec deux échecs, puis Z=c, Z=b (réponses inversées).

4 - Règle de calcul : choix du littéral le plus à droite
Règle de recherche : choix des clauses dans l'ordre d'entrée
Programme : celui du cas 3
Réponses : boucle immédiate et infinie sans aucune réponse.

VII.4- Interprétation procédurale

- L'écriture des clauses est un processus de définition de relations.
- Cette écriture peut devenir *programmer* plutôt que de *définir* si l'écriture d'un programme propose une composition des clauses reflétant une façon particulière de résoudre les questions posées. Une manière de conférer une telle propriété à un programme est de lui associer une **interprétation procédurale**.

- Sous une telle interprétation, les programmes acquièrent un sens procédural complétant leur sens logique (leur interprétation) défini par leur définition même.
- Une telle interprétation calculatoire (procédurale) désigne la différence entre la logique et la logique calculatoire.
- L'interprétation procédurale est déterminée par la SLD-résolution contrôlée par des règles de choix.
- Étant donnée ces informations, nous pouvons considérer la question :

<- q_1 & q_2 & ... & q_n

avec les q_i comme des appels de procédures.

- De même pour les clauses.

VII.4.1- Contrôle avec CUT

- Un contrôle supplémentaire du déroulement des programmes est possible avec le **cut** (noté !) qui n'a qu'une interprétation procédurale. Le cut ne provoque pas un calcul de réponses fausse. Son utilisation incorrecte peut pénaliser la complétude.

- Dans A <- <u>appels</u> & ! & autres_appels.

le "!" est effacé comme tout autre littéral. Lorsque "!" est franchi, son effet est d'élaguer :
- tous les essais non encore faits sur *A*
- tous les essais non encore faits sur *appels*.

- Exemple : on veut calculer le plus petit nombre Z parmi X et Y avec le prédicat *minimum(X, Y , Z)*. On dispose des prédéfinis ">" et "=".

On écrit :

minimum(X, Y, Z) <- plus_grand_ou_egal(X, Y) & Z = Y
minimum(X, Y, Z) <- Y > X & Z = X

plus_grand_ou_egal(X, Y) <- X > Y
plus_grand_ou_egal(X, Y) <- X = Y

Avec la question *minimum(3, 2, Z)*, il y a des redondances dans le test de comparaison de X et de Y.

- Nous avons une idée claire des propriétés des relations > et = mais, cette connaissance n'est pas à la disposition de la machine. Nous avons par exemple que ≥ et < sont complémentaires et mutuellement exclusifs. Nous pouvons alors envisager la spécification :
  minimum(X, Y, Z) <-
  <u>si</u> plus_grand_ou_egal(X, Y) <u>alors</u> Z = Y <u>sinon</u> Z=X.

  qui se traduit par le nouveau prédicat :
  minimum(X, Y, Z) <- plus_grand_ou_egal(X, Y) & **!** & Z = Y
  minimum(X, Y, Z) <- Y > X & Z = X
  plus_grand_ou_egal(X, Y) <- X > Y
  plus_grand_ou_egal(X, Y) <- X = Y

- Pour l'appel <- *minimum(3, 2, Z)*, nous aurons :

- Ils existent deux sortes de cut : *rouge* et *vert*.
- Un cut vert peut être supprimé sans modifier le comportement du programme. Sa présence n'apporte qu'une meilleure efficacité. Le "!" introduit ci-dessus est un cut vert.
- Un cut rouge, en revanche, ne peut être supprimé sans modifier le comportement du programme. Par exemple, si l'on modifie le prédicat minimum de la manière suivante :
  minimum(X, Y, Z) <- plus_grand_ou_egal(X, Y) & **!** & Z = Y
  minimum(X, Y, Z) <- Z = X

Le cut présent est un cut rouge. S'il est supprimé, on obtiendrait les réponses suivantes à la question :
<- minimum(3,2,Z) => Z=2, Z=3.

VIII. Sémantique des programmes définis

VIII.1- Interprétation de Herbrand

- Nous travaillons avec l'univers (U_H) et la base de Herbrand (B(P)).

- Pour présenter une <u>interprétation de Herbrand</u>, prenons un exemple :
  Soit le domaine U_H = {jean, marie, logique} et le programme P :

```
aime(jean, X) <- aime(X, logique).
aime(marie, logique).
```

- G(P) = l'ensemble des instances closes de P est obtenu en remplaçant les variables par les éléments de U_H :

```
aime(jean, jean)        <- aime(jean, logique)
aime(jean, marie)       <- aime(marie, logique)
aime(jean, logique)     <- aime(logique, logique)
aime(marie, logique)
```

- Une interprétation de Herbrand est simplement une assignation libre de valeur de vérité "vraie" et "fausse" aux éléments de B(P). Cette interprétation va donc associer une valeur de vérité aux éléments de G(P).

- Pour assigner des valeurs de vérité, il suffit en fait d'assigner ces valeurs à un sous-ensemble I de B_P. Ainsi, si un atome de B(P) a une valeur vraie dans I alors cet élément prend la valeur vraie, sinon il aura la valeur fausse.

Pour l'exemple ci-dessus, si l'on prend I={aime(marie, logique), aime(jean, marie)}, les élément auxquels on donne la valeur "vraie" alors G(P) seront :

```
faux <- faux.
vrai <- vrai.
faux <- faux.
vrai.
```

Ainsi, toutes les clauses de G(P) sont vraies (étant donné la table de vérité de l'implication). Cette interprétation est donc un <u>modèle</u> de Herbrand pour G(P) et pour P.

VIII.1.1- Définition formelle

Soit C un ensemble fini de constantes et F l'ensemble fini des symboles fonctionnels d'arité n >0.

Le domaine de Herbrand (U_H) est l'ensemble potentiellement infini des termes de premier ordre que l'on peut construire avec les ensemble C et F.

Exemple : Si C = {0} et F contient *succ* comme symbole fonctionnel d'arité 1, on a

U_H = {0, succ(0), succ(0), succ(succ(0)),...}.

Pour construire une <u>interprétation de Herbrand</u>, on associe :

- Chaque constante à <u>elle même</u>
- Chaque symbole fonctionnel n-aire f à une fonction F : H^n -> H tel que pour tout t□H^n, F(t) = f(t).
- Chaque symbole de prédicat n-aire p à une relation R_p tel que R_p ∏ {t | p(t) □ B(P)}.

- Une interprétation de Herbrand est par fois être présentée par le triplet S=<D, f_D, T_A>

où :

- D = U_H = domaine de Herbrand
- f_D = associe un élément de D à tout terme fonctionnel n-aire (n ≥0)
- T_A = une fonction d'assignation d'une valeur de vérité aux atomes de B(P).

Le triplet S est une <u>interprétation</u> (appelé également une ***structure***).

Résumé :

- Une interprétation associe une valeur de vérité vraie ou fausse aux atomes de G(P).
- Un modèle est l'ensemble des atomes de G(P) ayant une valeur vraie sous une interprétation rendant les clauses de G(P) valides.

- Un programme P a un <u>modèle</u> si-et-seulement-si G(P) a une interprétation de Herbrand.

C'est à dire, si-et-seulement-si P a une interprétation de Herbrand.

VIII.2- Les modèles d'un programme

- Un modèle est un ensemble particulier d'atomes clos assignés à vrai.
- Un programme peut avoir une infinité de modèles. La question est de savoir si un modèle assigne plus d'atomes à vrai; si un modèle est plus grand qu'un autre...

- La relation qui va nous intéresser est la relation d'inclusion ensembliste $\prod$ entre les modèles. Nous verrons comment calculer le "meilleur" modèle.

- Si l'on prend l'ensemble $M(P) = \{M_1, M_2, ...\}$ de tous les modèles d'un programme, et si l'on choisi la relation $\prod$ comme une relation d'ordre partiel alors, $[M(P), \prod]$ est un treillis complet qui a par conséquent un élément (modèle) <u>minimal unique</u> et un élément (modèle) <u>maximal</u>.

- Les clauses de G(P) doivent être vraies sous une interprétation I.

Pour qu'une clause C : A <- B_1 & ..., B_n soit vraie sous I, il faut :

- soit B_i □ I pour au moins un i : $n \leq i \leq 1$
- soit B_i □ I et A □ I pour tout i : $n \leq i \leq 1$

- Pour l'exemple, une façon simple de procéder est de prendre la base de Herbrand

B(P) = {jj, jl, jm, lj, ll, lm, mj, ml, mm}

et d'assigner vrai à tous ses atomes. Ceci rend toutes les clauses vraies. C'est le <u>modèle maximal</u> pour G(P) et donc pour P.

- Cependant, ce modèle est trop excessif car une grande partie des atomes de B(P) ne figurent pas dans G(P).

- Pour l'exemple précédent, G(P) est (à droite : les abréviations) :

aime(jean, jean)	<- aime(jean, logique)	***jj <- jl***
aime(jean, marie)	<- aime(marie, logique)	***jm <- ml***
aime(jean, logique)	<- aime(logique, logique)	***jl <- ll***
aime(marie, logique)		***ml***

- Quels sont les éléments qui doivent figurer dans tout modèle ?

- D'une manière informelle, *ml* doit figurer dans tout modèle afin que la quatrième clauses soit vraie. Par conséquent, *jm* doit y figurer aussi pour rendre la seconde clause vraie. Les autres atomes peuvent être faux; les clauses une et trois seront ainsi vraies.
- Donc il suffit que seuls *ml* et *jm* soient vrais : le <u>modèle minimal</u> de G(P) (noté MM(P)) et donc de P est l'ensemble {ml, jm}, c'est à dire :
  MM(P) = {aime(marie, logique), aime(jean, marie}.

- Le graphe ci-dessous montre le treillis des modèles pour le programme P ci dessus. Les arcs représentent la fermeture réflexo-transitive de la relation Π entre les modèles :

- Si B(P) possède n éléments, il existe 2^n interprétations de Herbrand possibles dont seulement une partie seront des modèles pour P; les autres étant des *contre-modèles*. Dans l'exemple ci-dessus, il y a 64 modèles dans le treillis et 448 contre modèles. Tous les modèles doivent contenir {jm, ml} : le modèle {jm, ml} constitue <u>l'intersection</u> de tous les modèles.

- Nous verrons que MM(P) correspond à ce qui est calculé par Prolog.

- Nous avons mis l'accent sur les programmes définis. Les programmes non-définis ne produisent pas de treillis complet de modèles et ont multiple modèles minimaux. il est donc difficile de leur associer un sens non ambigu.

VIII.3- Modèles minimaux et conséquence logique

- Si une interprétation I est un modèle pour P, I est un modèle pour chaque clause de P (la clause est vraie dans I).

Pour que la clause A <- B soit vraie, il faut (A | ~B) = vrai; c'est à dire :

- soit B □ I
- soit B □ I et A □ I

- Il y a un lien simple mais important entre un programme défini P, son modèle minimal MM(P) et chacune des ses conséquences logiques (closes) q :

  $\mathcal{P} \models q$ *si-et-seulement-si* $q \in \mathcal{MM}(\mathcal{P})$.

MM(P) est exactement l'ensemble de tous les atomes clos impliqués par P.

- La preuve de cette relation utilise la propriété d'intersection des modèles :

  Si $\mathcal{M}_1,\ldots,\mathcal{M}_n$ sont des modèles de Herbrand pour $\mathcal{P}$
  alors leur intersection est un modèle de Herbrand pour $\mathcal{P}$.

- On peut montrer que si M_1 ... M_n sont des modèles pour P alors, pour $\forall k<n$, $I_k = M_1 \leftrightarrow \ldots \leftrightarrow M_k$ est un modèle pour P. On montre également que l'intersection de tous les modèles est effectivement le modèle minimal.

- Par le théorème de Herbrand, nous savons qu'un ensemble de clauses P est non-satisfiable si-et-seulement-si un sous-ensemble fini de G(P) est non-satisfiable : ce sous-ensemble fini n'a pas de modèle de Herbrand (H-modèle).

Ceci revient à dire que pour le programme P et la question q, il existe un sous-ensemble fini g de G(P) tel que $g \approx \{\sim q\}$ n'a pas de H-modèle. En termes d'implication, cela veut dire qu'un un sous-ensemble fini g de G(P) implique q.

- Ce résultat est important car avec les ensembles potentiellement infinis U_H, B(P) et G(P), tout atome $q \in$ MM(P) est conséquence d'un sous-ensemble fini de G(P). La SLD-résolution étant réfutation-complète, on est assuré que ~q est réfutée avec P après un nombre fini d'étapes de résolution utilisant seulement un sous-ensemble fini de P.

VIII.3.1- Résultats importants

- Un programme défini P peut avoir une infinité de modèles (M(P))

- [M(P), $\prod$] est un treillis complet.

- Le minimal de ce treillis est un modèle minimal (MM(P))

- MM(P) est l'intersection de tous les éléments de M(P)

- Les atomes de MM(P) sont exactement ceux que le programme défini P implique : P |= q SSI q MM(P)

- q MM(P) SSI P≈{~q} |= ⇨ pour une question q.

- q MM(P) SSI un sous-ensemble fini de G(P≈{~q}) n'a pas de H-modèle.

- q MM(P) SSI'l existe un sous-ensemble fini g de G(P) tel que g≈{~q} n'a pas de H-modèle.

- q MM(P) SSI un sous-ensemble fini de G(P) implique q.

- La SLD-résolution produit une réfutation sur une question *q* après un nombre fini d'étapes si q MM(P) (donc dans tous les modèles)

VIII.4- Modèle minimal et la dérivation

- Nous avons exposé les résultats importants concernant l'implication et le modèle minimal :
  $\forall q \in \mathcal{B}_p, \quad q \in \mathcal{MM}(\mathcal{P})$ *si-et-seulement-si* $p \models q$.

- Qu'en est-il avec l'aspect calculatoire de la logique. Comment cela se traduit en terme de dérivation ?

- L'ensemble des atomes q tels que <-*q* réussisse dans P via la SLD-résolution est appelé l'ensemble de succès de P noté SS(P).

- Nous avons les résultats suivants :

$$\forall q \in \mathcal{B}_p,\ q \in \mathcal{MM}(\mathcal{P})\ SSI \quad \begin{array}{l} p \models q \\ \mathcal{P} \approx \{\sim q\} \text{ est non-satisfiable} \\ \mathcal{P} \approx \{\sim q\} \vdash_{\mathcal{SLD}} \Rightarrow \end{array}$$

- Et grâce à la justesse et la complétude de la SLD-résolution

$$\mathcal{SS}(\mathcal{P}) = \mathcal{MM}(\mathcal{P})$$

- Les questions sous la forme $<\text{-}q_1 \ \&\ \ldots\ \&\ q_n$ peuvent contenir des variables. Une telle question est la phrase

$$\forall(\sim(q_1 \ \&\ \ldots\ \&\ q_n) \int \sim\exists(q_1 \ \&\ \ldots\ \&\ q_n)$$

utilisée dans la réfutation.
Dans ce cas, la résolution construit un unificateur le plus général θ et nous avons des réponses génériques de la forme $\forall((q_1 \ \&\ \ldots\ \&\ q_n)\ \theta)$.

Nous avons $G(q_i\ \theta) \prod SS(P)$ pour tout $i \in \{1..n\}$.

- La collection des atomes de

$$G(q_1\ \theta) \approx \ldots \approx G(q_n\ \theta)$$

appartient également à SS(P).

- Si le programme définit les prédicats $p_1, \ldots, p_3$, alors

$$SS(P) = SS(p_1) \approx \ldots \approx SS(p_n)$$

- Une question de la forme $p_i(X_1,\ldots, X_r)$ où r est l'arité de p_i et $X_1,\ldots, X_r$ des variables dénote $SS(p_i)$: on demande toutes les valeurs pour les variables : tous les unificateurs θ_j tels que $p_i(X_1,\ldots, X_r)\theta_j \prod SS(p_i)$.

- On peut par exemple pour les prédicats p1, p2 et p3, partitionner la zone SS(P) de la manière suivante :

Exemple :

pour le programme P simple :

```
chiffre(1).      chiffre(2).
lettre(a).       lettre(b).
alpha(X) <-      chiffre(X).
alpha(X) <-      lettre(X).
```

Nous avons :

$U_H = \{1,2,a,b\}$

B(P) = { chiffre(1), chiffre(2), chiffre(a), chiffre(b),
lettre(1), ... lettre(b),
alpha(1) alpha(b)}

SS(chiffre) = {chiffre(1), chiffre(2)}
SS(lettre) = {lettre(a), lettre(b)}
SS(alpha) = SS(chiffre) ≈ SS(lettre)
SS(P) = MM(P) = SS(alpha)

La différence entre SS(P) et B(P) est la zone d'échec.

VIII.5- Construction du modèle minimal

- Il y a un moyen simple pour construire MM(P) d'une manière incrémentale.

- Cette méthode n'est pas un algorithme car si MM(P) est infini, la construction ne se terminera pas. cependant, à chaque de construction, on obtient une meilleur approximation.

- La méthode commence par choisir une interprétation $I_1 \prod B(P)$ constituée des atomes vrais dans tout modèle de P. Habituellement, I_1 contient les (clauses unités positive (les faits).

Ensuite, on considère les clauses de P de la forme A <- B_1 & ..., B_n.
I_{j+1} est construit à partir de I_j en ajoutant à I_j les atomes A si B_1 & ..., B_n sont tous dans I_j. l'atome A est la conséquence immédiate de I_j :

$$I_{j+1} = I_j \approx \{A \mid (A \text{ <- } B_1 \text{ \& ..., } B_n) \square G(P) \text{ et } B_1 \text{ \& ..., } B_n \square I_j\}.$$

On peut montrer qu'il existe un k tel que $I_{k+1}=I_j$. Les éléments de I_k sont des atomes qui doivent être vrais dans tout modèle du programme.

Exemple : soit U_H = {jean, marie} et le programme

```
gentille(X) <- sympa(X)
sympa(jean).
```

On a G(P) =

```
gentille(jean)      <- sympa(jean)
gentille(marie)     <- sympa(marie)
sympa(jean).
```

et B(P) = {gentille(jean) , sympa(jean), gentille(marie), sympa(marie)}

I_1 = {sympa(jean)}
I_2 = {sympa(jean), gentille(jean)}
$I_3 = I_2$

- Toute question ayant une réfutation figure dans I_2.
- I_2 est le modèle minimal de P.

- En l'absence de toute indication préalable sur MM(P), il est plus "économique" de commencer l'itération à partir de l'interprétation vide □ : I_0={□}. Ce qui a pour effet d'insérer dans I_1 les clause unités positives (les faits).

- Dans la section suivante, nous verrons les définitions formelles et les propriétés de cette itération.

VIII.6- Interprétation point-fixe du modèle minimal

- Le processus décrit ci-dessus génère chaque interprétation I_{k+1} à partir de son prédécesseur I_k. Il peut être vu comme l'application d'un fonction T_p:

$$T_p(I_k) = I_{k+1}$$

Cette fonction est une application totale de l'ensemble des parties de B(P) vers B(P). Elle est propre à chaque programme et définie par :

T_p(I) = {q | (q <- corps) □ G(P) et corps est vrai dans I}.

VIII.6.1- Définitions formelles

- Soit f une fonction définie sur l'ensemble partiellement ordonné [S, ≤]. Une propriété importante de la fonction f définie est d'être <u>monotone</u> et <u>continue</u>.

Par la monotonie, nous préservons la structure et par la continuité, nous préservons la limite de la fonction :

- Monotonie : soit les ensembles s_1, s_2 et s_3 .

f est monotone si $s_1 \prod s_2 \prod s_3$ alors $f(s1) \prod f(s2) \prod f(s3)$

- Continuité : soit un ensemble S. f est continue si

pour tout sous-ensemble orienté s de S, f(lub(s)) = lub({(f(u) | u □ s}).

où *lub(s)* dénote la plus petite borne supérieure de s et s est orienté si et seulement si $\forall u,v \square s, \exists w \square s$ tel que u $\prod$w et u $\prod$w

- Ces propriétés garantissent à f d'avoir un **point fixe**. Pour la fonction f définie sur [S, ≤], un élément u□s est un point fixe de f si f(u)=u.

- On fait correspondre
  - M(P) : l'ensemble des interprétation = √(B(P)) à S,
  - $\prod$: la relation d'ordre partiel sur M(P) à la relation ≤ et
  - La fonction T_P à la fonction f.

[M(P), $\prod$] constitue un treillis complet des interprétations de P.

- T_p étant continue définie sur les treillis, elle est nécessairement monotone.

- On peut montrer que la fonction T_p sur le treillis [M(P), $\prod$] possède un point fixe minimal.
- Pour construire ce point fixe, soit l'ensemble des entiers naturel {0,1,2,..} et ω sa cardinal (son premier ordinal limite). Toute fonction continue f sur un treillis complet [S,≤] dont le plus petit élément est ⊥ possède un point fixe minimal dans S tel que

$$f\neq\omega = lub(f^k (\perp) \mid k \square N\}$$

De plus, si ≤ est l'ordre partiel $\prod$ dont le plus petit élément est □ alors on a

$$f\neq\omega = lub(f^k (\square) \mid k \square N\} = \approx_{k\square N} f^k(\square)$$

- L'expression f≠ω (f haut oméga) est d'une importance particulière pour T_p qui nous intéresse: il correspond exactement au <u>modèle minimal</u> de MM(P).

- Nous disposons ainsi d'une méthode de construction incrémentale de l'ensemble {□, T_p^1(□), T_p^2(□), ... } à la recherche d'une plus petite borne supérieure qui coïncide avec MM(P).

Lorsque MM(P) est fini, nous obtiendrons le plus petit k□N tel que T_p^k(□) = T_p^{k+1}(□) = MM(P). Si MM(p) n'est pas fini (selon le programme P), ce point fixe existe mathématiquement même si nous ne pouvons pas le calculer en un temps fini.

- MM(P) est le plus petit point fixe de T_P car :
  d'après la définition de T_P :

  Pour tout I∏ B(P), ∀q
  q□T_p(I) SSI (∃corps) [(q <- corps)□ B(P) et corps est vrai dans I]

  et d'après la définition de modèle :

  Pour tout I∏ B(P)
  I est un modèle pour G(P) SSI ∀q,
  q□I si (∃corps) [(q <- corps)□ G(P) et corps est vrai dans I]

Ce que l'on peut résumer par :

Pour tout I∏ B(P)
I est un modèle pour G(P)

si-et-seulement-si	*∀q, q□I si q□T_p(I)*
si-et-seulement-si	*T_p(I)∏I*
si-et-seulement-si	*I est un point fixe de T_p*

- Nous avons maintenant plusieurs caractérisations du modèle minimal :

  MM(P) = SS(P) = plus-petit-point-fixe(T_p) = $T_p^{\neq\omega}$

Exemples :
définir les points fixes des programmes suivants dont U_H={0, s(0), s(s(0)),...}:

1) p(s(X)) <- p(X)
□–>□ : point fixe. Il y a aucun fait dans le programme
Le modèle minimal est □;

2) p(0)

p(s(X)) <- p(X)
□ –> {p(0)} –> {p(0), p(s(0)} –>... –> {p(t) | t □U_H} : point fixe.
Le point fixe est inféré par la limite d'un ensemble infini d'atomes.
Le modèle minimal est {p(t) | t □U_H}

3) p(X)
p(s(X)) <- p(X)

□ –> {p(X) | X □U_H} –> {p(X), p(s(X)) | X □U_H}
–>...–> {p(t) | t □U_H} : point fixe.
On constate que si X□U_H alors s(X)□U_H et vice versa.
En fait, le point fixe est obtenu après un pas d'itération car après avoir obtenu □–> {p(X) | X □U_H}, l'itération suivante n'apporte rien de plus car {p(s(X)) | X □U_H} ∏ {p(X) | X □U_H}.
Le point fixe est inféré par la limite d'un ensemble infini d'atomes.
Le modèle minimal est {p(t) | t □U_H}.

IX. Sémantiques dénotationnelles des programmes définis

- L'objectif de la sémantique est d'associer une relation r à chaque symbole de prédicat p dans le programme défini P. Cette relation représente le sens de p : c'est ce que p **dénote** dans le programme P.

- On doit définir r de façon non redondante : un r minimal et <u>unique</u> de telle sorte que le sens de r soit <u>sans ambiguïté</u>.

- Si P est défini, ces propriétés s'obtiennent par le modèle minimal MM(p) qui attribue à tout symbole de prédicat n-aire de P la plus petite relation n-aire dans U^n_H : les deux déterminent un modèle.

- Pour l'exemple de programme précédent
  p(X)
  p(s(X)) <- p(X)

Nous avons obtenu le modèle minimal MM(P) {p(t) | t □U_H} représentant ce que p dénote dans ce programme : une relation unaire dont l'argument est dans U_H. Ici, le prédicat p dénote l'univers de Herbrand.

- Les différentes manières que nous avons étudiées pour caractériser le modèle minimal correspond aux différentes notions de sémantiques :

- Une sémantique à base de **conséquence** (théorie de modèles) exprime la dénotation de tout p dans P en terme de <u>conséquence logique</u> par la relation :

$$\{t \mid \mathcal{P} \models p(t)\}$$

- Une sémantique **opérationnelle** dont le <u>comportement</u> nous permet de déterminer quel dénotation doit être donnée aux divers symboles de prédicats. En choisissant la SLD-résolution, la dénotation de tout p dans P est exprimée en terme de dérivabilité par la relation :

$$\{t \mid \mathcal{P} \approx \{<-p(t)\} \vdash_{SLD} \Rightarrow \} = \{t \mid p(t) \in SS(\mathcal{P})\}$$

- Une sémantique **point fixe** exprime la dénotation de tout p de P par la plus petite relation définie qui est fermée sous l'application d'un fonction continue T_P :

$$\{t \mid p(t) \in \mathcal{PPPF}(\mathcal{T}_p)\} = \{t \mid p(t) \in \mathcal{T}_p \neq \omega\}$$

- Dans toutes ces sémantiques, la dénotation de p dans P se trouve être simplement :

$$\{t \mid p(t) \in \mathcal{MM}(\mathcal{P})\}$$

Ainsi, le modèle minimal joue un rôle central dans les propriétés sémantiques de tout programme défini.

- Pour les programmes non définis, on peut avoir plusieurs modèles minimaux. Selon l'exécution de ces programmes, il n'est pas évident de définir clairement les sémantiques de ces derniers.

X. Traitement de la Négation

X.1- Hypothèse du monde fermé

- Etant donné un ensemble de clauses P, <u>l'hypothèse du monde fermé</u> (CWA) est l'affirmation que P contient toutes les connaissances dont on a besoin

pour déterminer les relations spécifiées. C'est une des caractéristiques de la logique binaire.

- Par cette hypothèse, si P |= q alors q est vrai sinon, q est faux.
- On peut opposer à cette hypothèse celle du monde ouverte où une question non démontrable n'est pas forcément fausse.

- CWA donne lieu au mode de raisonnement appelé inférence par défaut . Ce mode de raisonnement est utilisé dans la transformation de programmes, l'échec fini et le raisonnement par information négative.

- Pour un programme P et sous CWA, on infère l'atome ~q de tout atome q□B(P) qui n'est pas impliqué par P. C'est une forme particulière du raisonnement par défaut :

  inférer ~q à défaut d'impliquer q par P.

- L'objectif est de tirer des conclusions justes à propos des informations négatives. Par exemple, de la base de données :

  homme(jean).
  homme(pierre).

On déduira ~homme(jacques).

- Les conditions nécessaires pour l'inférence par défaut sont :

- On se restreint à inférer l'atome ~q lorsque P |≠ q, q□B(P). Sans quoi, on peut impliquer n'importe quoi par un programme, y compris les atomes ne faisant pas l'objet de la question traitée;
- P doit être un programme défini;
- P doit être consistant; il ne doit pas impliquer q et ~q;
- q doit être une formule atomique. Si P n'implique ni *r* ni *~r*, alors à défaut, leur opposés seront vraies. On va donc considérer *~r* et *~~r*; c'est à dire *~r* et *r* seront vrais; ce qui est inconsistant.

- Soit le programme :

  A <- B

La base de Herbrand B(P) = {A, B}.

Ce programme n'implique ni A ni B. On peut donc inférer ~A et ~B.

- Formellement :

$$CWA(P) = P \approx \{\sim q \mid q \in B(P), \textbf{not } P \models q\}.$$

Pour l'exemple, CWA(P) = { A <- B, ~A, ~B}.

- Pour apporter la <u>justesse</u> du raisonnement, il faut formaliser la déduction des informations négatives en termes de <u>conséquence logique</u>.

  Nous avons déjà, sous l'hypothèse du monde ouvert (OWA) :

$$\forall q \in B(P),\ P \models q \text{ si } P \models_{OWA} q$$

où OWA fait référence à la conséquence logique (habituelle, sans l'inférence par défaut) de P.

Pour l'inférence des informations négatives :

$$\forall q \in B(P),\ CWA(P) \models \sim q \text{ si } P \models_{CWA} \sim q$$

- Etant donnée que nous ne pouvons pas directement distinguer un atome particulier négatif, nous devons d'abord inférer ces atomes pour un programme défini P. Ce processus est appelé la **complétion** de P et fait l'objet de la section suivante.

- Rappelons que P doit être un programme défini. Sans le respect de cette condition, les programmes complétés peuvent devenir inconsistants.
  Par exemple, pour le programme : A | B
  CWA(P) = {A | B, ~A, ~B}. Ce qui est inconsistant.

- Un résultat important : si P est défini alors on a
  CWA(P) = P si MM(P) = B(P)
  car si MM(P) = B(P) alors {~q | q ∈ B(P), q ∉ MM(P) } = ∅
  donc CWA(P) = P ≈ ∅ = P.

X.2- Complétion de programmes

- La **complétion** de P est un ensemble particulier de phrases noté **Comp(P)**. La complétion fournit un sens (orienté conséquence) au raisonnement par défaut.

- Comp(P) est une traduction pratique de CWA(P) et peut être obtenu directement pour un programme bien structuré. Comp(P) ne nécessite pas la <u>déclaration</u> explicite des propositions négatives (comme c'est le cas dans CWA(P)).

X.2.1- La complétion en propositionnelle

Le processus est le suivant :
Soit P un ensemble de clauses de la forme *q <- corps*.
1- Pour tout q <u>mentionné</u> dans P mais <u>non défini</u> dans P, construire ~q
2- pour tout q défini dans P de la forme :

$$q \leftarrow corps_1$$
$$\ldots$$
$$q \leftarrow corps_n$$

Construire la phrase

$$q \Leftrightarrow (corps_1 \mid \ldots \mid corps_n)$$

Remarquons que le corps des assertions (faits) est vrai.

- Les phrases obtenues constituent la <u>définition complétée</u> de la relation q. Ainsi, Comp(P) contient la définition complétée de toutes les relations mentionnées dans P.

Exemples :

1- soit P le programme :

$$A \leftarrow B$$

Comp(P) =

$$\sim B$$
$$A \Leftrightarrow B$$

Le programme P n'implique ni A ni B. Comp(P) permettra de déduire ~B (directement) et ~A implicitement (CWA(P) aurait explicité ~A).

2- Avec le programme (non défini) P

A <- ~B

Comp(P) =

~B

A <=> ~B

B est mentionné mais non défini. Par la règle-1 de la construction, nous considérons B (et non pas ~B) et ajoutons ~B.

Ici, CWA(P) aurait déclaré ~A et ~B; c'est à dire CWA(P) = {~A, ~B, A <- ~B}, ce qui est inconsistant.

3- Avec le programme P (définissant la phrase valide A | ~A)

A <- A

Comp(P) =

A <=> A

On a considéré A mentionné et défini. Là encore, CWA(P) aurait inclue ~A explicitement.

- Comp(P) est habituellement plus <u>constructive</u> que CWA(P).

4- Avec le programme P <u>mal-structuré</u>

A <- ~A

Comp(P) =

A <=> ~A

Notons que dans ce cas particulier :

- Comp(P) = {A <=> ~A}∫{(A <- ~A) & (~A <- A)}
  ∫{(A | A) & (~A | ~A)}
  ∫ {A & ~A} ∫ ⇨.

Comp(P) est inconsistante; on en déduit n'importe quoi et son contraire.

- CWA(P) = {(A <- ~A)}. Elle n'ajoute aucune information négative et P demeure consistant.

- Nous utiliserons Comp(P) dans la section traitant de l'interprétation de l'échec fini et la négation.

- Pour mieux comprendre Comp(P), nous pouvons la considérer comme le résultat de simplification de *P ≈ seulement-si(P)* où *seulement-si(P)* inclue :

- toute définition complétée de la forme ~q
- la phrase (q seulement-si W) pour toute définition complétée de la forme (q si-et-seulement-si W).

Exemple :

A <- B
B <- C
B <- D
C

où A est vérifié si B l'est, B est vérifié si C ou D le sont, C est toujours vrai et D ne l'est pas. C'est exactement ce qui est exprimé par seulement-si(P).

seulement-si(P) =
~D
A seulement-si B
B seulement-si C | D
C seulement-si vrai (i.e., vrai)
Comp(P) = P ≈ seulement-si(P) =
~D
A <=> B
B <=> C | D
C. (i.e., C <=> vrai)

Dans laquelle la définition complétée de (q si-et-seulement-si W) est :
- prendre la définition de q de P
- prendre (q seulement-si W) de seulement-si(P)
- combiner les deux et simplifier.

- Comp(P) capte mieux l'intention du programmeur : quand celui-ci écrit (A <- B), il suppose réellement que (A si-et-seulement-si B) car, s'il veut que A soit vraie par d'autres moyens, il l'exprime par des définitions

supplémentaires. Son programme suffit pour calculer les réponses attendues (positives).

Par exemple, si le programmeur écrit le programme P_1:

A <- ~B

Comp(P_1) = {A <=> ~B, ~B}

Et s'il écrit P_2 :

B <- ~A

Comp(P_2) = {B <=> ~A, ~A}

Les deux programmes décrivent (A | B) et pourtant, les complétions ne sont pas équivalentes (les programmes le sont). Ces différences reflètent l'intention du programmeur : dans P_1, on anticipe les questions de la forme <-A; dans le second, les questions de la forme <-B.

X.2.2- La complétion en prédicats du premier ordre

- On procède d'abord par une <u>généralisation</u> :

Supposons qu'on ait la seule clause q(a). Le sens de cette clause est :

q est vrai pour a et seulement pour a.

On la remplace par sa généralisation :

q(a) devient q(X) <- X=a

- D'une façon générale, on généralise les têtes des clauses :

q(X, X, a, f(U)) <- corps

devient

q(X, Y, Z, W) <- Y=X & Z=a & W=f(U) & corps

- On procède ensuite comme le cas propositionnel :

1- Pour toute clause généralisée, quantifier par ∃ toutes ses variables locales figurant dans corps;

2- Spécifier '=' pour le programme et le domaine concerné

- Le point (1) est important. Considérons l'exemple :

parent(X) <- enfant(Y, X)

Si l'on se contente d'écrire *seulement-si* de la complétion :

parent(X) *seulement-si* enfant(Y, X)

Ceci revient à écrire

enfant(Y, X) <- parent(X)

Ce qui est contraire à l'intention exprimée dans la clauses initiale.

La clause initiale est en fait

parent(X) <- (∃Y) enfant(Y, X)

Et l'on ajoute

parent(X) *seulement-si* (∃Y) enfant(Y, X)

La définition complétée est alors :

parent(X) <=> (∃Y) enfant(Y, X)

Exemple :

supposons que la relation *parfait(X)* existe dans B(P) (par exemple via une question posée) mais pas dans l'entête des clauses du programme P :

```
gentille(X) <- aide(X,Y)
aide(paul, X)
```

On obtient :

```
gentille(X)     <=>(∃Y) aide(X,Y)
aide(X,Z)       <=>X = paul
~parfait(X)
```

Comp(P) indique que les personnes gentilles sont exactement ceux qui aident quelqu'un; la seule personne qui aide tout le monde est paul et personne n'est parfaite.

- La complétion sera achevée si l'on considère l'univers de Herbrand et l'égalité '=' au sens de la théorie équationnelle. Appelons *Eq(H, P)* la théorie équationnelle sur l'U_H et un ensemble de clause P. Cette théorie correspond à l'unification de Robinson où l'on s'intéresse à la fois aux cas de succès et d'échecs. La complétion d'un programme P est donnée par :

  $Comp(P) = P \approx seulement\text{-}si(P) \approx Eq(U_H, P)$

Pour simplifier, nous omettons d'expliciter cette théorie.

- Si pour le programme ci-dessus, U_H={jean, paul}, nous pourrons obtenir des conséquences négatives telles que :
  ~parfait(paul)
  ~parfait(jean)
  ~gentille(jean) ....

- La complétion produit en fait un <u>nouveau</u> programme.

Exemples de programmes P et leur complétion (sans la théorie équationnelle) :

1- num(0)
num(s(X)) <- num(X)

Comp(P) =
num(Y) <=> Y=0 | (∃X) (Y= s(X) & num(X))

2- doublon(U.X) <- U □ X
doublon(V.X) <- doublon(X)

Comp(P) =
doublon(Y) <=> (∃U) (∃X) (Y=U.X & U □ X)
| (∃V) (∃X) (Y=V.X & doublon(X))

Ce que l'on peut écrire sous la forme plus compacte :
doublon(Y) <=> (∃U) (∃X) (Y = U.X & (U □ X | doublon(X)))

3- sait(jean, Y) <- ordinateur(X) & en_rapport(X,Y)
sait(paul, Y) <- dit(jean, Y, paul)

Comp(P) =
sait(W, Y) <=> (W= jean & (∃X) ordinateur(X)
& en_rapport(X,Y)))

| (W= paul & dit(jean, Y, paul))

Ainsi, jean sait tout sur les ordinateurs, paul sait les choses que jean lui apprend, aucun d'eux ne sait autre chose et personne d'autre ne sait rien de rien.

XI. L'échec fini

- Le paradigme SLD est adéquat pour tout calcul. Cependant, les clauses de Horn pures (sans négation et d'autres effets de bord non-logiques) ne peuvent pas constituer un véritable langage de programmation praticable.

- Des extensions non-logiques ont dû être apportées par Prolog pour lesquelles l'on trouve des explication théoriques bien fondées. Une des plus importante de ces extensions est **l'échec fini**.

XI.1- Extension de la SLD avec l'échec fini

- L'échec fini permet d'écrire **fail q** (c'est à dire : *q* ne peut être démontré) : une condition pour que q ait un échec fini. Cette forme d'origine a été repris et exprimée par *not q* ou \+*q* dans les Prolog et traduit par la négation par échec. Nous utiliserons la notation *not q* par la suite.
- Le sens opérationnel de **not** est donné par la **règle d'échec fini** :

  not q *a un succès fini SSI* ***q*** *a un échec fini*

L'interprète effectue un appel à *q* et conclut sur une réponse selon ce résultat.

- L'échec fini de *q* est un point crucial : il doit exister un arbre d'échec fini pour *q*; c'est à dire, avec une profondeur finie, avec une largeur finie et tous les calculs qu'il contient doivent se terminer par un échec. Si un seul de ces calculs réussit, alors *not q* échoue.

Exemple :

```
personne(jean)
personne(marie)
personne(paul)
enfant(marie, jean)
enfant(paul, jean)
```

On peut utiliser la SLD pour extraire des réponses positives. Mais pour savoir si une information n'est pas dans la base :

<- personne(X) & not enfant(X, jean).

Ce qui conduit la SLD à résoudre <- *not enfant(jean, jean)*. La démonstration de *enfant(jean, jean)* échoue (fini) et pour *not enfant(jean, jean)*, le premier calcul réussit. Par des retours arrières, les autres calculs échoueront.

- Nous remarquons dans cet exemple que dans *not q*, q est <u>clos</u> au moment de son exécution.

- Avant de soulever certains problèmes lorsque q n'est pas clos, rappelons les conventions de lecture quantifiée des clauses :

La clause p(X) <- q(Y)
est lue (∀Y)(∀X) (p(X) <- q(Y)) ∫ (∀X) (p(X) <- (∃Y) q(Y))

La clause p(X) <- **not** q(Y)
est lue (∀Y)(∀X) (p(X) <- **not** q(Y)) ∫ (∀X) (p(X) <- (∃Y) **not** q(Y))

- Prenons un cas où *not q* n'est pas clos.

Par exemple :
Q : not enfant(X, jean).
Quel est le sens de la question :
Existe-t-il un X tel que *enfant(X,jean)* échoue (échec fini) ?
La règle d'échec fini soumet la question
Q' : enfant(X, jean).
C'est à dire, Existe-t-il un X tel que *enfant(X, jean)* se vérifie.
Q' réussit et Q échoue et l'exécution se termine avec aucune réponse pour X.

Pourtant, il existe un X tel que enfant(X, jean) échoue dans un calcul fini : c'est X = jean.

Nous somme face à une situation étrange où Q échoue alors qu'une de ses instances, c'est à dire Q{X/jean} réussit.

La règle d'échec fini est ainsi démontrée **incomplète** si elle est appliquée à un appel *not q* non clos. Cette incomplétude est bien sûr propagée si de tels appels figurent dans les corps des clauses.

- Il y a des situations pires :

Ajoutons les clauses suivantes à la base précédente :

parent(Y) <- *not* sans_enfant(Y)
sans_enfant(Y) <- *not* enfant(X, Y)

et posons la question <- *parent(Y)*.

La requête réussit laissant entendre qu'il existe un parent sans lier Y.
Pourtant, si l'on considère CWA, la lecture déclarative des clauses implique que *parent(Y)* est vérifié SSI il existe aucun X tel que *enfant(X,Y)* ne se vérifie pas. Mais puisqu'un tel X existe (X=jean), il faut alors que parent(Y) ne soit pas vérifié pour tous les Y.
Ainsi, Par cette lecture déclarative, la réponse calculée est fausse et la règle d'échec fini est **incorrecte**.

- La règle d'échec fini est incomplète et injuste si *not q* n'est pas clos.
- Pour un système d'inférence, ne pas donner toutes les réponses serait (en partie) plus tolérable que de donner des réponses fausses.
- Un solution à cette situation est de disposer d'une règle de choix de littéraux (règle de calcul) restreinte qui :
  - ne sélectionne jamais le littéral *not q* si *q* n'est pas clos
  - sélectionne en priorité *not q* si *q* est clos.

Une telle règle est appelée la **règle de calcul sûre**.

- Avec cette règle, et la question <- *not enfant(X, jean) & personne(X)*, le littéral choisi est *personne(X)* qui produit pour X la valeur *jean*. Puis, on choisit *not enfant(jean, jean)* qui réussit et calcule X=jean. Les retours arrières successifs échouerons.
- Si l'on considère la question <- *parent(Y)* précédente avec la règle de calcul sûre, le calcul est suspendu sur <- *not sans_enfant(Y)*. Le calcul doit être avorté et si l'on n'a pas de réponse. On évitera en tout cas de produire une réponse incorrecte.

- L'incorporation de la règle d'échec fini avec la SLD-résolution sous l'hypothèse d'un calcul sûr donne l'inférence **SLDNF** (SLD + négation par

échec). Cette règle est **incomplète** car des situation de suspension et d'avortement de calculs peuvent se produire. Cette incomplétude est le prix à payer pour la justesse de la SLDNF.

- Une grande partie des Prolog laissent aux programmeur la responsabilité des conséquences de la négation par échec.

- La motivation principale de ces discutions est de permettre le raisonnement par défaut : un élément important du raisonnement du monde réel permettant une utilisation raisonnable et pratique de la programmation logique dans la résolution des problèmes.
- Une autre utilisation importante de *not* dans les bases de données est la possibilité de contrôle d'intégrité des données de ces bases.

XI.2- Ensemble d'échecs fini de la SLD résolution

- Parallèlement à l'ensemble de succès SS(P), nous allons étudier l'ensemble d'échec fini d'un programme <u>fini</u> P avec la SLD-résolution et une règle de calcul sûr.
- Cet ensemble comprend exactement les atomes clos *q* pour lesquels <- *q* a un échec borné.

- Considérons l'exécution de <-*q*, le programme P et la règle R de calcul SLD. La base de Herbrand B(P) peut être partitionnée en trois parties d'atomes clos q :

- Ceux où <-*q* réussit **SS(P, R)**
- Ceux où <-*q* a un échec fini **FF(P, R)**
- Ceux où <-*q* a un échec infini **IF(P, R)**

- En quoi ce partitionnement dépend de R. Nous avons vu que le succès de la requête <-*q* ne dépend pas de R : si une requête réussit dans un arbre SLD, elle réussit dans tout arbre SLD. On va donc noter l'ensemble de succès par **SS(P)** sans faire référence à R.

- Ce n'est pas le cas pour l'échec. Par exemple

  q <- r & q

peut échouer de façon finie avec la règle SLD classique et de façon infinie dans une règle choisissant d'abord le littéral de droite.

- D'une manière formelle, on peut se libérer de toute règle R en utilisant une règle *équitable* qui assure qu'un littéral est traité après un nombre arbitraire mais fini d'étapes de résolution.

Par exemple, pour q <- r & q
on peut imaginer que q est sélectionnée mais, après un certain nombre d'étapes, r sera sélectionné et produira un échec fini.
On peut donc dire que <-*q* a un échec fini sous une règle R si et seulement s'il a un échec fini sous toutes les règles équitables. De même pour l'échec infini.

- Ce genre de règles ne sont pas implantées dans les systèmes pratiques existants, mais nous somme intéressés ici par leurs propriétés mathématiques. Ce qui nous permet de nous libérer d'une règle R particulière et d'écrire FF(P) comme l'ensemble d'atomes q pour lesquels <-*q* a un arbre SLD équitable. De même pour IF(P).

- Pour l'exemple ci-dessus, on aura FF(P) = {q, r} et IF(P) = □.

Résultats :

pour $\forall q \in \mathcal{B}(\mathcal{P}), q \in \mathcal{FF}(\mathcal{P})$ *SSI <- not q réussit sous SLDNF*

$q \in \mathcal{SS}(\mathcal{P})$ *SSI <- q réussit sous SLDNF*

- Nous étudierons plus loin l'échec fini en termes de la fonction T_p et de la négation classique.

Exemple :

soit H = {a,b,c} et le programme :

```
chemin(X, Z) <- arc(X, Z)
chemin(X, Z) <- arc(X, Y) & chemin(Y, Z)
arc(a,b)
arc(b,c)
```

Identifier SS(P,R), FF(P,R) et IF(P, R) pour chacune des règles R suivantes :

(i) R choisit toujours le littéral le plus à gauche
(ii) R choisit toujours le littéral le plus à droite
(iii) R est équitable avec un choix arbitraire

Réponses :

(i) **SS(P, R)** = { arc(a, b), chemin(a, b),
arc(b, c), chemin(b, c),
chemin(a, c)}

FF(P, R) = { arc(a, a), chemin(a, a),
arc(a, c),
arc(b, a), chemin(b, a),
arc(b, b), chemin(b, b),
arc(c, a), chemin(c, a),
arc(c, b), chemin(c, b),
arc(c, c), chemin(c, c)}

IF(P, R) = $\emptyset$

(ii) **SS(P, R)** = comme dans sans chemin(a, c) omis

FF(P, R) = {	arc(a, a),	**IF(P, R)** = {	chemin(a, a),
	arc(a, c),		
	arc(b, a),		chemin(b, a),
	arc(b, b),		chemin(b, b),
	arc(c, a),		chemin(c, a),
	arc(c, b),		chemin(c, b),
	arc(c, c)}		chemin(c, c)}

(iii) Comme en (i)

XI.3- Caractérisation mathématique de l'échec fini

- Nous avons vu que pour tout programme défini P, il existe un moyen simple de construire l'ensemble de succès SS(P) en utilisant la fonction T_P. On commence par □ dans le treillis des interprétations et l'application répétée de cette fonction génère une chaîne monotone croissante :

$$\square \sqsubseteq T_P(\square) \sqsubseteq T_P^2(\square) \sqsubseteq \ldots$$

dont la limite est $T_{P\neq\omega} = \text{lub}\{T_P^k(\square) \mid k \square N\} = \approx_{k \square N} T_P^k(\square)$

Cette limite est exactement SS(P) et correspond à PPPF(T_P) : le plus petit point fixe de T_P.

- Il y a un moyen similaire pour construire FF(P). Il faut avant cela bien comprendre le processus faisant un échec fini à une requête <-*q* dans le programme P. Nous avons besoin de la notion de profondeur à laquelle cet échec intervient.

- La plus simple façon pour que <-*q* échoue est qu'il n'y ait aucune clause dans P dont la tête s'unifie avec *q*. D'une manière équivalente, cet échec arrive si l'atome *q* n'est pas dans les entêtes de G(P). Dans ce cas, l'échec a lieu à la profondeur **k=1**.

Pour un programme donnée, il peut y avoir plusieurs atomes q□B(P) dans ce même cas. L'ensemble de tous ces atomes est dénoté par **FF(P,1)**.

On peut aisément caractériser FF(P, 1) par T_P.
Rappelons la définition de la fonction T_P :

$T_P(I) = \{q \mid (q \leftarrow corps) \in G(P)$ et corps est vrai dans $I\}$

Que représente $T_P(B(P))$: les atomes qui figurent comme entête dans G(p).

$T_P(B(P)) = \{q \mid (q \leftarrow corps) \in G(P)$ et *corps* est vrai dans $B(P)\}$

Pour avoir un échec au niveau k=1, il faut que *q* ne soit pas dans les entêtes de G(P), i.e. $q \notin T_P(B(P))$.
Donc si $q \in B(P)$ et $q \notin T_P(B(P))$, alors $q \in B(P) - T_P(B(P))$.

$$\mathbf{FF(P, 1) = B(P) - T_P(B(P))}$$

Nous venons d'écarter les atomes non définis dans G(P) mais présents dans B(P). A partir de ce stade, nous allons considérer les atomes q ayant une définition de G(P). Ces atomes échoueront si un littéral de leur corps n'a pas définition dans G(P).

- Si *q* n'est pas dans FF(P, 1), cela veut dire que $q \notin T_P(B(P))$. Nous devons examiner les corps des clauses de G(P) dont l'entête est *q*.

Pour que *q* échoue à k=2, il faut que le corps de la clause $q \leftarrow q_1 \& \ldots \& q_r$ de G(P) échoue. Une façon simple pour cela est que l'un des littéraux du corps ne soit pas défini dans G(P).

Que représente $T_P^2(B(P))$: les atomes définis dans G(P) dont le corps est défini dans G(P). Les atomes qui ne sont pas dans $T_P^2(B(P))$ sont ceux qui ont dans leur corps au moins un atome non défini dans G(P). Ces atomes échouent à **k=2**.
Donc

$$\mathbf{FF(P, 2) = B(P) - T_P^2(B(P))}$$

- Ainsi, à chaque itération, on écarte des atomes dont l'absence va causer le rejet d'autres atomes à l'itération suivante.

- Si l'on généralise ce principe, FF(P) contient uniquement les atomes *q* pour lesquels <-*q* échoue à un pas fini k□N :

$$\mathbf{FF(P) = \approx_{k\square N} FF(P,k) = B(P) - \leftrightarrow_{k\square N} T^k{}_P(B(P))}$$

Puisque P a un nombre fini d'atomes, une requête <-*q* peut générer un nombre fini de calcul (q appelle p qui appelle r qui appelle t...). Si tous ces calculs échouent alors il doit exister une profondeur k à laquelle tous ces calculs terminent.

Remarquons qu'en commençant avec l'élément de sommet B(P) du treillis des interprétations, l'itération de T_P engendre la séquence monotone et décroissante :

$$B(P) \quad T_P(B(P)) \quad T^2{}_P(B(P)) \ ...$$

dont la limite est :

$$\mathbf{glb\{T^k{}_P(B(P)) \mid k\square N\} = \leftrightarrow_{k\square N} T^k{}_P(B(P))}$$

Ce que l'on dénote par $T_p\neg\omega$ (T_P bas oméga). On peut alors écrire

$$\mathbf{FF(P) = B(P) - T_p\neg\omega}$$

glb correspond à la plus grande borne inférieure.

- Cette interprétation mathématique de FF(P) est totalement indépendante du mécanisme d'exécution.

Exemple-1 :

P= G(P) =

A_1 <- B_1 & C1 & D1
B1 <- B1 & F1
C1 <- F1 & **H1**
D1 <- A1 & E1
E1 <- C1
F1

B(P) = {A1,B1,C1,D1,E1,F1,**G1,H1**}

T_P(B(P)) = {A1,B1,C1,D1,E1,F1}G1 et H1 écartés k=1

$T^2_P(B(P))$ = {A1,B1,D1,E1,F1} C1 écarté k=2

$T^3_P(B(P))$ = {B1,D1,F1} A1 et E1 écartés k=3

$T^4_P(B(P))$ = {B1,F1} D1 écarté k=4

$T^5_P(B(P))$ = $T^4_P(B(P))$

FF(P) = B(P) - $T^4_P(B(P))$ = {A1, C1, D1, E1, G1, H1}

Exemple-2 :

```
aime(jean, X) <- aime(X, logique)
aime(marie, logique)
```

B(P) = {jj, jl, jm, lj, ll, lm, mj, ml, mm}

G(P) (à droite : les abréviations) =

aime(jean, jean)	<- aime(jean, logique)	***jj <- jl***
aime(jean, marie)	<- aime(marie, logique)	***jm <- ml***
aime(jean, logique)	<- aime(logique, logique)	***jl <- ll***
aime(marie, logique)		***ml***

$T_P(B(P))$ = {jj, jm, jl, ml}

FF(P,1) = B(P) - $T_P(B(P))$ = {lj, ll, lm, mj, mm}

$T^2_P(B(P))$ = {jj, jm, ml}

FF(P,2) = B(P) - $T^2_P(B(P))$ = {lj, ll, lm, mj, mm, jl}

$T^3_P(B(P))$ = {jm, ml}

FF(P,2) = B(P) - $T^3_P(B(P))$ = {lj, ll, lm, mj, mm, jl, jj}

$T^k_P(B(P)) = T^3_P(B(P))$ = {jm, ml}.

Pour $\forall k>3$, on a : $T^3_P(B(P)) = T_p\neg\omega$

- Comme T_P est monotone et $\square \prod B(P)$ on montre aisément que

$$\mathbf{T_p\neq\omega \prod T_p\neg\omega}$$

- La structure suivante montre les différentes zone de la base de Herbrand en fonction des limites de T_P :

FF(P) est le complément (ensembliste) de $T_p\neg\omega$ par rapport à B(P). Elle peut être calculée également par la différence entre la complétion de P (Comp(P)) et de $T_p\neg\omega$)

La différence entre $T_p\neq\omega$ et $T_p\neg\omega$ correspond à IF(P) représentant les atomes avec un échec <u>infini</u>.

Ce sont des atomes q pour lesquels l'exécution de <-q génère une 'boucle' sans aucun succès. Notons que l'infinité peut être causée par soit une boucle classique (récursivité à gauche) où par un calcul infini sans boucle. Considérons le programme suivant dont U_H = {0, s(0), s(s(0))...}:

```
nombre <- num(X)
zero(0) <- zero(0)
num(X) <- num(s(X))
```

où la question <- *zero(0)* entre immédiatement dans une boucle infinie alors que la question <-*nombre* développe un arbre infini sans boucle mais avec une infinité (de branches) d'échecs finis. L'atome *nombre* ne réussit pas et n'a pas une profondeur de calcul fini : il ne peut appartenir qu'à IF(P). *zero(0)* appartient également à IF(P) et tout autre atome de ce programme sera dans FF(P).

La zone IF(P) est caractérisée par :

$$T_p\neq\omega \prod \mathbf{PGPF(T_P)} \prod T_p\neg\omega$$

où PGPF(T_P) est le plus grand point fixe de T_P.

- Pour les cas ordinaires de la majorité des programmes logiques, on a :

  IF(P) = □ sans échec infini

auquel cas $T_p \neq \omega = T_p \neg\omega$ et
$B(P) = SS(P) \approx FF(P)$

Exemple-3 : avec le programme

p(a).
p(b) <- p(b)
<- p(c)

$U_H = \{a, b, c\}$
$B(P) = \{p(a), p(b), p(c)\}$

$T_p \neq \omega = SS(P) = \{p(a)\}$
$T_p \neg\omega = \{p(a), p(b)\} = SS(P) \approx IF(P)$
$FF(b) = \{p(c)\} = B(P) - T_p \neg\omega$
$IF(P) = \{p(b)\} = T_p \neg\omega - SS(P)$

XI.4- Sémantique de l'échec fini par complétion

- Pour les programmes définis, nous avons
  $\forall q \in B(P), P \models q$ si-et-seulemnt-si $q \in SS(P)$

- Dès que l'on introduit *not*, ce résultat n'est plus valide.

- Une manière de lier l'ensemble de succès SS(P) à la conséquence logique est d'utiliser Comp(P) à la place de P :
  - En interprétant not comme le connecteur ~ (par la négation par échec)
  - En liant le succès ou l'échec au Comp(P)

- Ainsi, l'on obtient la justesse :
  $\forall q \in B(P)$, $Comp(P) \models q$ si **<-q** réussit sous SLDNF
  $Comp(P) \models \sim q$ si **<-fail q** réussit sous SLDNF

- Exemple : pour le programme P

q <- not p
q <- r
r

On a

Comp(P) =

q <=> (~p | r)
~p

De Comp(P), on obtient un succès pour les requêtes

<- q, <- r, <- ~p

- Il y a des problèmes de complétude car la complétion <u>n'est pas consistante</u> et la règle d'échec fini n'est pas correcte et complète pour les littéraux négatifs contenant des variables.

- Sous un certain nombre de conditions strictes (de structure) pour les programmes et les requêtes, on obtiendrait une forme restrictive de complétude :

∀q □ B(P),	**<-q** réussit sous SLDNF si	Comp(P) I= q
	<-not q réussit sous SLDNF	Comp(P) I= ~q

- En conclusion, la négation apporte un certain nombre de problèmes théoriques en particulier quand elle comporte des variables. En contre partie, son utilisation est d'un tel intérêt que l'on peut accepter certaines conditions restrictives.