S. Cerrito

Logique pour l'Informatique

(Licence en Informatique et IUP2, CFA)

(Version revisée en septembre 96)

Table des matières

1 INTRODUCTION

Sujet du cours

Ce cours se propose d'aborder l'étude de la logique. Mais qu'est que c'est la logique ? Grosso modo, on pourrait dire que la logique est une formalisation des lois qui font si qu'un raisonnement est correct, au-delà de son contenu.

Par exemple, considérons les raisonnements suivants.

1. S'il neige, alors il fait froid. Il neige. Donc, il fait froid.

2. Si la lune est faite de camembert, alors les astronautes peuvent manger la lune. La lune est faite de camembert. Donc, les astronautes peuvent manger la lune.

Les premiers deux raisonnements sont généralement reconnus comme étant corrects. En effet, il s'agit de deux exemples de la *même* forme de raisonnement :

Si @@@ alors —.
@@@.
Donc, —.

La correction de ces deux raisonnements ne dépend pas de la vérité des premises (par exemple, la deuxième premise du deuxième raisonnement est évidemment fausse), mais de la signification attribuée à toute expression de la forme "Si @@@ alors —". Généralement (ou, au moins, dans les contexte du discours mathématique), on considère que la seule façon de rendre une expression de la forme "Si @@@ alors —" fausse c'est de rendre "@@@" vrai et "—" faux; donc, si nous *supposons* que "si @@@ alors —" et "—" sont vrais (c'est à dire : si nous supposons que les premises du premier ou bien du deuxième raisonnement ci-dessus sont vraies), alors "—" est forcement vrai (c'est à dire : la conclusion du raisonnement est forcement vraie). Comparez les exemples ci-dessus au raisonnement suivant, qui *n'est pas* correct :

Si la terre est plate, alors il est possible de construire une autoroute allant de New-York à San Francisco. Il est possible de construire une autoroute allant de New-York à San Francisco. Donc la terre est plate.

Un peu d'histoire

La logique, en tant qu'étude des lois formelles du raisonnement, est née en Grèce. En particulier, Aristote (IV^e siècle av. J.C), a développé une théorie des syllogismes (un exemple typique de syllogisme est un raisonnement ayant la forme : Tous les A sont des B et tous les B sont des C; donc tous les A sont des C). Le but d'Aristote était d'étudier la forme des raisonnements scientifiques, par exemple de la physique. Ensuite, la logique a connu un certain développement pendant le Moyen-Age, mais pas de progrès substantiels jusqu'au début du vingtième siècle. C'est à cette période qu'elle naît comme vraie "science".

Parmi les "pères fondateurs" de la logique on peut citer Frege, Russell, Hilbert, Gödel, Gentzen, Herbrand... Le contexte est celui du fort intérêt que l'on sent, au début de ce siècle, pour les problèmes dits des “fondements des mathématiques”. On éprouve la nécessité de fonder les théories mathématiques sur une base “sûre”, car la vérité de certaines parties des mathématiques, notamment la théorie des ensembles, semble être minée par des paradoxes.

Un exemple fameux de paradoxe en théorie des ensembles est le paradoxe de Russel. Supposons que pour bien définir un ensemble il suffise de définir une propriété qui caractérise ses éléments. Considérons alors l'ensemble E défini par :
$E = \{x \mid x \text{ est un ensemble tel que } x \notin x\}$.
On obtient alors $E \in E$ ssi $E \notin E$. Donc E est bien défini et pourtant il est contradictoire !

Bien que le moteur initial du développement de la logique au siècle actuel ait été la problématique des fondements des mathématiques, ensuite son étude s'est de plus en plus affranchie de ce but initial. Toutefois, pendant un bon moment les mathématiques sont restées le champ principal d'application de la logique.

Mais dans la deuxième partie du vingtième siècle on a établi des connexions de plus en plus étroites entre la logique et la toute nouvelle science de l'informatique. Par exemple, des liens étroits entre la logique et le λ-calcul fournissent la base théorique de la programmation dite “fonctionnelle” (LISP, ML, etc.). D'autre part, l'étude de la recherche automatique des preuves logiques amène à la naissance de la programmation “logique” (PROLOG, par exemple). La logique comme moyen de représentation des connaissances est largement utilisée en Intelligence Artificielle. Nombreuses sont les applications de la logique aux Bases de Données; par exemple, un langage logique est à la base de plusieurs langages de requête commerciaux. Plusieurs problèmes de typage sont étudiés à l'aide d'outils logiques, etc.

Le point de vue du cours

Dans ce cours, on abordera l'étude de la logique en connexion avec la programmation logique. Plus spécifiquement, on étudiera la logique comme base théorique du langage PROLOG.

Un programme PROLOG définit des relations, plutôt qu'indiquer un algorithme de calcul. Par exemple, supposons que nous voulions écrire un programme calculant la somme de deux entiers positifs ou nuls. On peut représenter les entiers par z, s(z), s(s(z)), ... (la lettre minuscule “z” représentera 0 et “s” représentera la fonction “successeur”) et écrire le programme :

```
plus(z,X,X).
plus(s(Y), X, s(V)) :- plus(Y,X,V).
```

On lira la première ligne comme : pour tout entier X, 0+X=X et la deuxième comme : pour tous entiers X, Y, V, si X+Y=V, alors s(Y)+X=s(V). Pour calculer, par exemple, la somme de 2 et 3, on pose la “requête” (ou “but”) : plus(s(s(z)), s(s(s(z))),W). PROLOG “cherche alors une preuve” de cette dernière expression à partir des déclarations du programme, ainsi qu'une valeur appropriée pour la variable W. Il fournit la réponse : W=s(s(s(s(s(z))))).

On peut voir les instructions de PROLOG, dites “clauses”, comme les premises d'un raisonnement et une requête PROLOG comme un énoncé-conclusion. Un calcul de PROLOG est alors la recherche d'une preuve du fait que la conclusion suit logiquement à partir des prémisses, en même temps que des valeurs appropriées pour les variables de la requête.

La logique étant ainsi la base de PROLOG, dans ce cours on présentera d'abord (et surtout !) la logique "en soi" pour enfin revenir à son utilisation dans le cas de PROLOG.

Quelques notions de base de PROLOG

Comment représente-t-on les données en PROLOG ? La plupart des langages de programmation permettent de construire des structures de données à partir de certains types concrets, par exemple entiers, flottants, listes, pointeurs, etc. Ceci n'est pas le cas pour PROLOG pur, où la seule structure de donnée est celle de *terme*.

Soit X un ensemble dénombrable de variables et F un ensemble de symboles de fonction, chacun muni de son arité, c'est-à-dire de son nombre d'arguments. L'ensemble des termes sur X et F est défini récursivement :

- Si $x \in X$, x est un terme,
- si $t_1, \cdots, t_n$, $n \geq 0$, sont des termes et $f \in F$ est d'arité n, alors $f(t_1, \cdots, t_n)$ est un terme.

On observera que toute constante c est un terme car c est un symbole de fonction à 0 arguments et on identifie les notations c et $c()$.

Dans la syntaxe PROLOG, tout mot écrit en minuscule peut être utilisé comme symbole de fonction et tout mot commençant par une majuscule peut être utilisé comme variable. Ainsi, s(z), s(s(z)) ... sont des termes, que l'on a choisi ci-dessus pour représenter les entiers; de même, X et $s(V)$ sont des termes.

Un programme PROLOG ne définit que des relations entre des termes. Pour ce faire, on dispose d'abord des *atomes*.

Soit T l'ensemble dénombrable des termes sur X et F et soit R un ensemble de symboles de relation, chacun muni de son arité. L'ensemble des atomes sur X et R est défini par :
si $t_1, \cdots, t_n$ sont des termes en T et $r \in R$ est d'arité n, alors $r(t_1, \cdots, t_n)$ est un atome.

Comme pour les symboles de fonction, tout mot écrit en minuscule peut être utilisé comme symbole de relation. Ainsi, plus(s(z),s(s(z)), s(s(s(z)))) est un atome, signifiant que 1+2=3.

On remarquera que, du point de vue de la syntaxe PROLOG, termes et atomes correspondent au même type d'expression. Toutefois, ces deux classes correspondent à deux notions logiques bien différentes. Intuitivement, un terme représente une donnée : par exemple, on a choisi s(z) comme représentant l'entier 1. Un atome représente non pas une donnée, mais un énoncé sur des données, qui peut être vrai ou faux pour les données en question. Par exemple, l'atome plus(s(z),s(s(z)), s(s(s(z)))) est vrai.

Un programme PROLOG n'est rien d'autre qu'une série de définitions de relations, à l'aide des *clauses PROLOG*.

L'ensemble des clauses PROLOG consiste en ce qu'on appellera ici "clauses de programme" et des "clauses requête".

Une clause de programme est une expression de la forme :

$$A : -B_1, \cdots, B_n.$$

où $A, B_1, \cdots, B_n$ sont des atomes et n un entier positif ou nul. L'atome A est appelé *tête* de la clause, et l'expression $B_1, \cdots, B_n$ *corps* de la clause. La lecture intuitive d'une clause de programme est la suivante : pour toutes valeurs des variables, si $B_1, \cdots, B_n$ sont tous vrais, alors A est vrai. Si $n = 0$, alors la clause est appelée *fait*; dans ce cas, on notera $A.$ à la place de $A : -$.

Une *définition* d'une relation r est une suite de clauses telles que r est le symbole de relation des atomes qui apparaissent dans les têtes. Par exemple, le programme vu ci-dessus constitue une définition de la relation "plus".

Un programme est une suite de définitions de relations.

Une clause requête est une expression de la forme :

$$: -B_1, \cdots, B_n.$$

où $B_1, \cdots, B_n$ sont des atomes et n un entier strictement positif. Sa lecture est la suivante : est-il vrai qu'il existe des valeurs des variables telles que $B_1, \cdots, B_n$ sont tous vrais ? Si oui, quelles sont ces valeurs ?

Un exemple d'évaluation PROLOG

Voyons, pour le moment intuitivement, comment PROLOG évalue une requête, par exemple plus(s(z), s(z),W) sur le programme "plus" ci-dessus.

Au départ, la requête courante est plus(s(z), s(z),W); indiquons par R_0 cette requête. Considérons la deuxième clause du programme. Vu que cette clause du programme se "lit" : "pour tous entiers X, Y, V, si X+Y=V, alors s(Y)+X=s(V)", pour trouver une valeur de W telle que s(z)+s(z)=W on peut se ramener à chercher une valeur de la variable V telle que on puisse établir z+s(z)=V. Si val est la valeur de V trouvée, pour obtenir la valeur de W cherchée il suffira de poser W=s(val).

Donc, la requête R_1 : plus(z,s(z),V), est maintenant la nouvelle requête courante.

Observez que pour déterminer qui est la nouvelle requête courante à partir de la requête de départ, il suffit de travailler sur la *syntaxe* de la clause du programme considérée et de R_0, de la façon suivante. Si on *substitue* la variable Y de la clause du programme avec le terme z, la variable X de la clause du programme avec le terme s(z) et la variable W de R_0 avec le terme s(V), alors la tête la clause du programme et R_0 deviennent la *même* expression, notamment "plus(s(z),s(z),s(V)". Avec cette même substitution de termes aux variables, le corps de la clause devient l'expression "plus(z,s(z),V)" : c'es bien cette expression que l'on a pris comme nouvelle requête courante.

Remarquez qu'aucune substitution de termes aux variables aurait permit d'identifier R_0 avec la première clause du programme.

Pour évaluer R_1, considérons la première clause du programme, qui est un fait (disant que, pour tout entier x, 0+x=x). La substitution de sa variable X par le terme s(z) e de la variable V de R_1 par s(z) permet d'identifier R_1 et le fait considéré (on obtient l'expression plus(z,s(z),s(z))). On peut donc prendre s(z) comme valeur de la variable V de R_1 .

Cette valeur de V nous fournit enfin la valeur de W de la requête de départ, càd s(s(z)).

Observons qu'on pourrait poser aussi la requête plus(s(z), X, s(s(z))) pour chercher un entier n tel que 1+n=2; (donc on pourrait interroger le programme "plus" pour calculer une différence). On remarquera que cela n'a pas de sens de considérer les entiers à additionner comme les entrées du programme et le résultat de la somme comme la sortie : les trois entiers sont sur le même plan, et sont seulement les arguments d'une relation.

Ainsi, un programme PROLOG n'indique pas un algorithme de calcul, mais seulement des définitions de relations entre des données. C'est le "moteur d'inférence" de PROLOG qui se charge de comment évaluer une requête. Pour le moment, on a regardé très intuitivement et informellement comment se passe l'évaluation. En réalité, la structure du moteur d'inférence de PROLOG est basée sur des notions de logique (résolution, unification), que l'on étudiera dans le cours.

2 Logique Propositionnelle

2.1 Syntaxe de la Logique Propositionnelle

BUT : Intuitivement, les expressions manipulées par la logique propositionnelle ont le rôle de formaliser des énoncés du langage courant, pouvant être vrais ou faux. Mais ici on s'occupe exclusivement de spécifier quelles expressions sont considérées, indépendamment de leur signification et de leur vérité.

Définition 1 Un alphabet $\mathcal{A}$ d'un langage propositionnel $\mathcal{L}_P$ consiste en la donné des symboles suivants :

- Un ensemble dénombrable de variables propositionnelles $p, q, r, \ldots$.
- Les deux constantes propositionnelles $\mathcal{V}$ et $\mathcal{F}$.
- L'ensemble de connecteurs logiques contenant le connecteur à un argument $\neg$ et les connecteurs binaires $\wedge, \vee, \rightarrow, \leftrightarrow$.
- Les parenthèses $(,)$.

Définition 2 Les expressions bien formées ou "formules" d'un langage propositionnel $\mathcal{L}_P$ sont les mots sur l'alphabet de $\mathcal{L}_P$ définis récursivement comme suit.

- Toute variable propositionnelle est une formule. De même, les constantes propositionnelles $\mathcal{V}$ et $\mathcal{F}$ sont des formules. Les formules ici définies sont dites formules *atomiques*;
- si F_1 et F_2 sont des formules, alors $(\neg F_1), (F_1 \wedge F_2), (F_1 \vee F_2), (F_1 \rightarrow F_2), (F1 \leftrightarrow F_2)$ sont aussi des formules.

Exemple 3 $((\neg(p \wedge q) \rightarrow (p \vee (q \wedge r)))$ est une formule; $pq\vee$ ne l'est pas.

Quand il n'y aura pas de possibilité de confusion, on évitera d'écrire certaines parenthèses. En particulier, on évitera d'écrire les parenthèses les plus extérieures et on suivra la convention suivante :

- le connecteur $\neg$ s'applique à la plus petite formule le suivant immédiatement; ainsi, on écrira $\neg p \wedge q$ à la place de $(\neg p) \wedge q$;

- $\wedge$ et $\vee$ "lient plus fort" que $\rightarrow, \leftrightarrow$; ainsi, on écrira $p \wedge q \rightarrow r \vee p$ à la place de $(p \wedge q) \rightarrow (r \vee p)$.

On peut aussi définir récursivement la notion de "sous-formule" comme suit.

Définition 4

- Si p est une formule atomique, l'ensemble des sous-formules de p est $\{p\}$;

- l'ensemble des sous-formules de $\neg F_1$ est l'union de $\{\neg F_1\}$ et des sous-formules de F_1; l'ensemble des sous-formules de F_1 *conn* F_2 où *conn* est un connecteur binaire est l'union de $\{F_1 \; conn \; F_2\}$, des sous-formules de F_1 et de celles de F_2.

2.2 Sémantique de la Logique Propositionnelle

BUT : Donner une signification aux connecteurs logiques, établissant ainsi la valeur de vérité (Vrai ou Faux) d'une formule complexe à partir de celles des formules atomiques qui la composent. Pour cela, il faut *interpréter* les variables propositionnelles, en leur attribuant une valeur de vérité.

Définition 5 Soit BOOL l'ensemble dont les éléments sont V ("Vrai") et F ("Faux"). Une fonction booléenne à n arguments (dite aussi "fonction de vérité) est une fonction $\text{BOOL}^n \Rightarrow \text{BOOL}$.

On associe au connecteur logique $\neg$ une fonction booléenne $f_\neg$ à un argument et aux autres connecteurs les fonctions booléennes binaires $f_\wedge$, $f_\vee$, $f_\rightarrow$, $f_\leftrightarrow$.

Définition 6 Les fonctions booléennes $f_\neg$, $f_\wedge$, $f_\vee$, $f_\rightarrow$, $f_\leftrightarrow$. sont définies par les tables suivantes; (l'argument de la fonction $f_\neg$ est indiqué dans la première colonne).

		$f_\neg$	$f_\vee$	$f_\wedge$	$f_\rightarrow$	$f_\leftrightarrow$
V	V	F	V	V	V	V
V	F	F	V	F	F	F
F	V	V	V	F	V	F
F	F	V	F	F	V	V

Définition 7 Une interprétation $\mathcal{I}$ pour un ensemble de variables propositionnelles P est une fonction ayant P comme domaine et BOOL=$\{V, F\}$ comme co-domaine.

Définition 8 La valeur de vérité d'une formule F par rapport à une interprétation $\mathcal{I}$, notée $[F]_\mathcal{I}$, est définie récursivement comme suit.

- Si F est une variable propositionnelle, $[F]_\mathcal{I} = \mathcal{I}(F)$; de plus, $[\mathcal{V}]_\mathcal{I} = V$ et $[\mathcal{F}]_\mathcal{I} = F$;

- $[\neg F_1]_\mathcal{I} = f_\neg([F_1]_\mathcal{I})$
  $[F_1 \; conn \; F_2]_\mathcal{I} = f_{conn}([F_1]_\mathcal{I}, [F_2]_\mathcal{I})$
  où *conn* est un connecteur binaire et f_{conn} est la fonction booléenne associée.

Définition 9 On dit qu'une interprétation $\mathcal{I}$ est un modèle d'une formule F si $[F]_{\mathcal{I}} = V$; dans ce cas, on note $\mathcal{I} \models F$. On dit que $\mathcal{I}$ est un modèle d'un ensemble de formules E si elle est modèle de tout élément de E.

EXEMPLES

Définition 10 Une formule F (resp. un ensemble de formules E) est satisfiable s'il existe au moins une interprétation $\mathcal{I}$ qui est modèle de F (resp. E); sinon, F (resp. E) est dite (resp. dit) insatisfiable.

EXEMPLES

Définition 11 Une formule F est valide si, pour n'importe quelle interprétation $\mathcal{I}$, $\mathcal{I}$ est modèle de F. On notera $\models F$ pour indiquer que F est valide.

EXEMPLES
On remarquera qu'il existe un moyen très simple pout déterminer si une formule F est valide (resp. satisfiable). On construit une "table de vérité" pour F, où chaque ligne représente une interprétation possible. F est valide (resp. satisfiable) si et seulement si, pour toute ligne de la table (resp. il existe une ligne telle que) la valeur de vérité de la formule F est V.
EXEMPLE

Définition 12 Soit $E = \{F_1, \cdots, F_n\}$, $n \geq 0$, un ensemble de formules, soit G une formule. On dit que G est conséquence logique de E, et on note $F_1, \cdots, F_n \models G$, si tout modèle de E est aussi un modèle de G.

Observer que dans le cas où $n = 0$, dire que G est conséquence logique de l'ensemble vide de formules revient à dire que G est valide.

On notera les faits suivants.

1. $F_1, \cdots, F_n \models G$ si et seulement si la formule $(F_1 \wedge \cdots \wedge F_n) \rightarrow G$ est valide.
2. Soit G n'importe quelle formule. Si E est un ensemble insatisfiable de formules, alors, de façon triviale, G est conséquence logique de E.
3. Soit G n'importe quelle formule valide. Alors, de façon triviale, G est conséquence logique de n'importe quel ensemble de formules E.
4. Soit E un ensemble de formules, G une formule. $E \models G$ si et seulement si $E \cup \{\neg G\}$ est insatisfiable. En particulier, G est valide si et seulement si $\neg G$ est insatisfiable.

La notion de conséquence logique est un outil permettant de vérifier la correction de certains raisonnements formulés en langage naturel. Par exemple, considérons le raisonnement suivant :

Je vous paierai pour votre installation de la TV seulement si elle marche. Or, votre installation ne marche pas. Donc je ne vous paierai pas

Ce raisonnement peut être formalisé de la façon suivante. On se donne le "vocabulaire" suivant :
p modélise l'énoncé "Je vous paierai pour votre installation de la TV";
m modélise l'énoncé "votre installation marche".
L'énoncé complexe "Je vous paierai pour votre installation de la TV seulement si elle marche" est alors modélisé par la formule $p \to m$ (et non pas par $p \leftrightarrow m$, attention!). Le raisonnement a alors les prémisses $p \to m$ et $\neg m$ et la conclusion $\neg p$. Pour vérifier sa correction il suffit de vérifier si la conclusion est une conséquence logique des prémisses, c'est-à-dire si $p \to m, \neg m \models \neg p$. Ceci est le cas. Donc on peut conclure que le raisonnement est correct. Par contre, observons l'raisonnement suivant.

Je vous paierai pour votre installation de TV seulement si elle marche. Je ne vous paierai pas. Donc votre installation ne marche pas.

Ici il s'agit de vérifier si $p \to m, \neg p \models \neg m$. Puisque ceci est faux, le raisonnement n'est pas correct. (Remarquer qu'aucune premise de ce deuxième raisonnement n'assurait un paiement dans le cas où l'installation aurait marché !!).)

Dans le cas où non seulement $F_1 \models F_2$ mais on a aussi la réciproque, c'est-à-dire que $F_2 \models F_1$, on dira que les deux formules F_1 et F_2 sont *logiquement équivalentes*. On notera ce fait par $F_1 \equiv F_2$. On remarquera que $F_1 \equiv F_2$ si et seulement si la formule composée $F_1 \leftrightarrow F_2$ est valide. (Faites bien attention à la différence entre le symbole $\leftrightarrow$, qui fait partie de l'alphabet du langage propositionnel, et la notation $\equiv$ qui n'en fait pas partie mais qui nous sert à exprimer une propriété de certains couples de formules du langage propositionnel. La même remarque s'applique à $\to$ et $\models$).

Ci-dessous, on donne des équivalences logiques qui nous seront utiles dans la suite. (Le sens exact du mot "définissabilité" sera précisé plus loin).

- $F \leftrightarrow G \equiv F \to G \wedge G \to F$ (définissabilité de $\leftrightarrow$ par $\wedge, \to$)
- $F \to G \equiv \neg F \vee G$ (définissabilité de $\to$ par $\neg, \vee$)
- $\neg(F \wedge G) \equiv \neg F \vee \neg G$ et $\neg(F \vee G) \equiv \neg F \wedge \neg G$ (lois de De Morgan)
- $\neg\neg F \equiv F$ (loi de double négation)
- $F \vee (G \wedge H) \equiv (F \vee G) \wedge (F \vee H)$ et $F \wedge (G \vee H) \equiv (F \wedge G) \vee (F \wedge H)$ (lois de distributivité)

Il est facile de prouver le résultat suivant.

Résultat 13 **(Principe de substitution)** Soient F une formule, A une sous-formule de F et B une formule telle que $A \equiv B$. Si on remplace une occurrence de A dans F par B, on obtient une formule F' telle que $F \equiv F'$.

Ainsi, par exemple, $\neg(\neg p \vee q) \wedge r \equiv p \wedge \neg q \wedge r$.

Définissabilité des fonctions booléennes

Jusqu'ici on a considéré exclusivement 4 fonctions booléennes binaires : $f_\vee, f_\wedge, f_\rightarrow$ et $f_\leftrightarrow$. En réalité, il existe d'autres fonctions de vérité binaires. (**Combien ? Exercice**). Et il existe aussi des fonctions booléennes à plus de deux arguments, par exemple celle définie par la table suivante.
EXEMPLE.
Mais toute fonction booléenne peut être "définie" à partir d'un très petit nombre de fonctions, au sens suivant.

Définition 14 Soit F une formule ayant $p_1, \cdots, p_n$, $n \geq 0$, comme variables propositionnelles. Soit $f(F)$ la fonction booléenne à n arguments telle que, pour toute interprétation $\mathcal{I}$, $f(F)(v_1, \cdots, v_n) = [F]_\mathcal{I}$ si $\mathcal{I}(p_i) = v_i$. On dit alors que la fonction $f(F)$ est réalisée par la formule F.

Par exemple, la fonction $f_\neg$ de la définition 6, est réalisée par les formules $\neg p, \neg\neg\neg p, \neg p \wedge (p \vee \neg p)$ etc.

Un ensemble de connecteurs C est dit *fonctionnellement complet* si toute fonction booléenne est réalisée par une formule utilisant seulement des connecteurs de C.

Résultat 15 L'ensemble de connecteurs $\{\neg, \vee, \wedge\}$ est fonctionnellement complet.

Preuve
Soit f une fonction booléenne à n arguments, $n \geq 0$. Si $n = 0$, alors f est ou bien la fonction à 0 arguments dont la valeur est toujours V ou bien la fonction à 0 arguments dont la valeur est toujours F. Dans le premier cas, la formule atomique $\mathcal{V}$ réalise f, dans le deuxième la formule atomique $\mathcal{F}$ réalise f.

Maintenant, soit $n > 0$ et soient $p_1, \cdots, p_n$ des variables propositionnelle. Soient $a_1 \cdots a_k$ les n-uplets de BOOL^n tels que f vaut V. Si $a_j = \langle v_1, \cdots, v_n \rangle$ soit F_j la formule $L_1 \wedge \cdots \wedge L_n$ où, pour tout i tel que $1 \leq i \leq n$, $L_i = p_i$ si $v_i = F$ et $L_i = \neg p_i$ si $v_i = F$. Il est clair que la formule $F_1 \vee \cdots \vee F_k$ réalise f.
Dans le cas particulier où la fonction f vaut toujours F, il suffit de prendre comme formule réalisant f n'importe quelle formule insatisfiable ayant comme variables propositionnelles $p_1, \cdots, p_n$. Par exemple, la formule $p_1 \wedge \neg p_1 \wedge p_2 \cdots \wedge p_n$.

EXEMPLE DE CONSTRUCTION DE LA FORMULE A PARTIR D'UNE TABLE DEFINISSANT UNE FONCTION BOOLEENNE

De plus, en utilisant les lois de De Morgan et l'équivalence logique $A \rightarrow B \equiv \neg A \vee B$, on obtient que les ensembles de connecteurs $\{\neg, \vee\}$, $\{\neg, \wedge\}$, $\{\neg, \rightarrow\}$ sont aussi fonctionnellement complets.

Il s'en suit qu'il aurait été suffisant prendre comme seuls connecteurs "primitifs" de notre langage $\mathcal{L}_P$ de la logique propositionnelle $\neg$ et $\vee$, par exemple.

En effet, dans le premier système de preuve que l'on va considérer, $\leftrightarrow$ n'est pas primitif.

2.3 Systèmes de Preuve en Logique Propositionnelle

BUT : se donner des moyens purement **syntaxiques** permettant de montrer qu'une formule est valide.

En général, un *système de preuve* pour la logique (ici, propositionnelle) est un ensemble fini de règles mécaniques, ne tenant compte que de la syntaxe des expressions manipulées, permettant d'obtenir une nouvelle expression à partir de certaines autres. Ces règles sont dites *règles d'inférence*; elles ont des expressions de départ, les *prémisses*, et une expression d'arrivée, la *conclusion*. Dans le cas particulier où le nombre de prémisses est nul, on dit que la règle est un *axiome*.

Une règle d'inférence a le rôle de formaliser un principe de raisonnement; un exemple typique de règle d'inférence est la règle dite de *modus ponens*, permettant de d'obtenir (déduire) n'importe quelle formule B à partir des formules $A \rightarrow B$ et A. On note cette règle d'inférence :

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

Un système d'inférence pour la logique propositionnelle “acceptable” doit satisfaire au moins les deux propriétés suivantes

1. Ses règles doivent être **correctes**, c'est-à-dire que la conclusion doit être toujours une conséquence logique de l'ensemble des prémisses. (Par exemple, la “règle”

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

   n'est pas correcte.) Si l'ensemble des règles du système est correct, il ne permettra de prouver que des formules valides.

2. L'ensemble des règles du système doit être **complet**, c'est-à-dire qu'il doit suffire à prouver toute formule valide.

Il existe plusieurs sortes de systèmes de preuve. Certains d'entre eux sont directs, c'est-à-dire qu'il se proposent comme objectif immédiat de prouver la validité de la formule A considérée. D'autres systèmes, dits “systèmes de preuve par réfutation”, se proposent plutôt comme but de prouver l'absurdité (plus précisément, l'insatisfiabilité) de $\neg A$, afin d'en conclure la validité de A. Ici, nous considérerons un système de preuve direct, le *calcul des séquents* de Gentzen (en deux versions) et un système par réfutation, dit *système de résolution* (Robinson), qui est en général considéré comme le système sous-jacent au langage de programmation logique PROLOG.

2.3.1 Systèmes de preuve par séquents

Le système de preuve connu sous le nom de **Calcul des Séquents LK** a été proposée par le logicien allemand Gentzen autour de années 30. Dans la suite, les lettres majuscules $A, B, C, \ldots$ dénotent des formules propositionnelles et les lettres grecques majuscules $\Gamma, \Delta \ldots$ dénotent des *multi-ensembles* de formules, çest-à-dire des ensembles de formules pouvant contenir des répétitions. (En tant que multi-ensemble, $\{A, B\}$ est différent de $\{A, A, B\}$, par exemple).[1]

[1] On peut aussi voir $\Gamma, \Delta \ldots$ comme des *listes* de formules, mais dans ce cas il est nécessaire de formuler explicitement une règle d'inférence, l'*échange*, permettant de permuter les listes.

La notion centrale est celle de *séquent*.

Définition 16 Un séquent est un couple de multi-ensembles de formules Γ, Δ, écrite sous la forme $\Gamma \vdash \Delta$. Le multi-ensemble de formules Γ est appelé antécédent, le multi-ensemble de formules Δ est appelé conséquent et le symbole $\vdash$ signe de thèse.

Intuitivement, un séquent est un moyen de séparer un ensemble d'hypothèses Γ (l'antécédent) d'un ensemble Δ de conclusions (le conséquent); l'interprétation intuitive d'un séquent $A_1, \cdots, A_n \vdash B_1, \cdots B_m$ est que sous les hypothèses A_1 et ... et A_n on peut obtenir l'une des conclusions B_1, ... ou B_m. Plus formellement :

Définition 17 Etant donnée un séquent $A_1, \cdots, A_n \vdash B_1, \cdots B_m$, on lui associe la formule :

$$A_1 \wedge \cdots \wedge A_n \rightarrow B_1 \vee \cdots \vee B_m$$

Le séquent est vrai dans une interprétation $\mathcal{I}$ si la formule associée est vraie dans $\mathcal{I}$. Il est valide si la formule associée est valide.

Remarque 18 Dans le cas particulier où $n = 0$, la conjonction $A_1 \wedge \cdots \wedge A_n$ est vide. Puisque une conjonction $A \wedge B$ est vraie par rapport à une interprétation si et seulement si A est vrai et B est vrai, une conjonction $A_1 \wedge \cdots A_n)$ est vraie ssi, pour tout A_i, $1 \leq i, n$, A_i est vrai. Par conséquent, la conjonction vide est toujours trivialement vraie. On peut donc utiliser le symbole spécial, $\mathcal{V}$ (qui est toujours évalué par la valeur booléenne V) pour écrire la conjonction vide. De même, si $m = 0$, la disjonction $B_1 \vee \cdots \vee B_m$ est vide. Puisque une disjonction $A \vee B$ est vraie par rapport à une interprétation si et seulement si au moins une formule parmi A, B est vraie, une disjonction $B_1 \vee \cdots \vee B_n$ est vraie ssi il existe au moins un i, $1 \leq i \leq n$, tel que B_i soit vrai. Par conséquent, la disjonction vide est toujours trivialement fausse. On peut donc utiliser le symbole spécial, $\mathcal{F}$ (qui est toujours évalué par la valeur booléenne F), pour écrire la disjonction vide. Ainsi, si, par exemple, $n = m = 0$, la formule associée au séquent est $\mathcal{V} \rightarrow \mathcal{F}$, qui est toujours fausse.

Etant donné un système de preuve par séquents, c'est-à-dire un ensemble de règles d'inférences dont les prémisses et la conclusion sont des séquents, une *déduction* du séquent $\Gamma \vdash \Delta$ à partir d'un ensemble Λ de séquents (dit ensemble d'hypothèses) est un arbre fini tel que

- tout nœud est un séquent;
- la racine est le séquent $\Gamma \vdash \Delta$; ce séquent est appelé conclusion de la déduction;
- les feuilles sont ou bien les conclusions des règles sans prémisse, c'est-à-dire des axiomes, ou bien des séquents de Λ;
- les fils d'un nœud Σ sont des séquents $\Sigma_1, \cdots, \Sigma_n$ tels que $\frac{\Sigma 1, \cdots \Sigma_n}{\Sigma}$ est une une règle d'inférence du système.

En particulier, une *preuve* du séquent $\Gamma \vdash \Delta$ est une déduction de $\Gamma \vdash \Delta$ à partir de l'ensemble vide de séquents : les seules feuilles possibles sont les *axiomes*.

Définition 19 Un séquent Σ est un théorème d'un système de preuve par séquents s'il existe une preuve de Σ dans ce système.

Calcul des séquents LK

Dans toute règle d'inférence ci-dessus, A et B sont des formules quelconque et Γ, Δ sont des multi-ensembles de formules quelconques. Si Γ est un multi-ensemble de formules et A une formule, l'expression Γ, A indique le multi-ensemble union de Γ et $\{A\}$.

Groupe Identité

$\frac{}{A \vdash A}$ (axiomes) $\quad \frac{\Gamma \vdash A, \Delta; \qquad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}$(coupure)

Règles structurelles

$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}$(contraction-g) $\quad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}$(contraction-d)

$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}$(affaiblissement-g) $\quad \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}$(affaiblissement-d)

Règles logiques

$\frac{}{\vdash \mathcal{V}}$ $\qquad \frac{}{\mathcal{F} \vdash}$

$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}$ (nég-g) $\qquad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \neg A, \Delta}$ (nég-d)

$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}$ (conj-g) $\qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}$(conj-d)

$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}$(disj-g)

$\frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \vee B}$ (disj-d1) $\qquad \frac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \vee B}$ (disj-d1)

$\frac{\Gamma \vdash A, \Lambda \qquad \Gamma, B \vdash \Lambda}{\Gamma A \rightarrow B \vdash \Lambda}$ (impl-g) $\qquad \frac{\Gamma, A \vdash \Delta, B}{\Gamma \vdash \Delta, A \rightarrow B}$ (impl-d)

Dans le cas des axiomes, on peut imposer que la formule A soit atomique, sans perte de généralité.

Voici un exemple de preuve de la formule $(p \wedge (p \rightarrow q)) \rightarrow q$ en **LK**. Pour prouver cette formule, il faut construire une preuve du séquent $\vdash (p \wedge (p \rightarrow q)) \rightarrow q$. Ci-dessous, un arbre de preuve qui convient.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{p \vdash p}\,(axiome)}{p \vdash p, q}\,(aff-d) \qquad \dfrac{\dfrac{}{q \vdash q}\,(axiome)}{p, q \vdash q}\,(aff-g)}{p, p \rightarrow q \vdash q}\,(impl-g)}{p \wedge (p \rightarrow q) \vdash q}\,(conj-g)}{\vdash (p \wedge (p \rightarrow q)) \rightarrow q}\,(impl-d)
$$

EXEMPLES DE RECHERCHE D'UNE PREUVE A PARTIR DE LA RACINE ET EN REMONTANT VERS LES FEUILLES.

Les règles d'inférence de **LK** ne jouent pas toutes le même rôle.

Les règles logiques définissent implicitement les connecteurs; par exemple, les deux règles disj-d disent que pour avoir une preuve de $A \vee B$ il suffit d'avoir ou bien une preuve de A ou bien une preuve de B.

Les règles structurelles établissent des propriétés générales du raisonnement à partir de certaines hypothèses.

La règle de coupure exprime la possibilité d'utiliser des "lemmes" à l'intérieur d'une preuve logique, propriété essentielle de toute formalisation adéquate de la notion de démonstration.

EXEMPLE

On observera que toute règle de **K** est *correcte* : la formule associée à la conclusion est toujours une conséquence logique de la conjonction des formules associées au prémisses. (Exercice)

On peut remarquer que toute règle de **LK**, à l'exception de la règle de coupure, est telle que toute formule qui apparaît dans le(s) séquent(s) prémisse(s) est une sous-formule d'une formule du séquent conclusion. Considérons le processus de construction d'une preuve **sans coupures** d'un séquent $\Gamma \vdash \Delta$, à partir de la racine (le séquent à prouver) et remontant vers les feuilles. Cette propriété des règles de **LK** montre que si on cherche une preuve sans coupures d'un séquent, on ne fait que "décomposer", au fur et à mesure, les formules du séquent conclusion via les règles logiques et appliquer éventuellement des règles structurelles, sans avoir besoin de faire intervenir une formule nouvelle. Si on arrive à construire un arbre dont la racine est le séquent à prouver et les feuilles sont des axiomes, on a prouvé le séquent.

Maintenant, essayons de formuler un **algorithme** permettant de trouver une preuve d'un séquent Σ (quand Σ est prouvable). Si on se donne le droit d'utiliser la règle de coupure dans le processus de construction d'une "preuve candidate" à partir de la racine, on rencontre une difficulté, car dans le cas de la règle de coupure, la formule sur laquelle on fait la coupure peut apparaître dans les prémisses sans apparaître dans la conclusion de la règle. Comment peut-on deviner quand utiliser une coupure et sur quelle formule ? Ceci pose évidemment des problèmes.

Or, un théorème important de Gentzen nous dit que la règle de coupure est redondante :

Théorème 20 Elimination des coupures
Tout séquent $\Gamma \vdash \Delta$ prouvable dans **LK** est aussi prouvable dans **LK**-$\{coupure\}$

Nous ne prouverons pas ici ce théorème. Observons que ce résultat ne veut nullement dire que la règle de coupure n'est pas importante : il dit seulement que toute preuve en **LK** d'un séquent $\Gamma \vdash \Delta$ peut être transformée en une preuve (plus longue !!) n'utilisant pas la règle de coupure; par conséquent, l'espace de recherche de preuves de n'importe quel séquent $\Gamma \vdash \Delta$ peut être réduit aux preuves sans coupures.

Il est possible, en effet, de prouver une forme plus générale de ce résultat, qui considère les déductions à partir d'un ensemble de séquents Λ qui ne soit pas forcément vide (comme dans le cas des preuves).

Théorème 21 Elimination des coupures généralisée
Si un séquent $\Gamma \vdash \Delta$ est déductible dans **LK** à partir d'un ensemble de séquents Λ, alors il existe une preuve qui utilise des coupures exclusivement par rapport à des sous-formules de formules de Λ.

Dans le cas particulier où Λ est vide, on obtient la première version du théorème.

Ainsi, on a vu que l'on peut éviter de considérer la règle de coupure quand on cherche une preuve. Mais il nous reste quand même deux autres difficultés, si nous voulons dégager une façon automatique de tester si un séquent est prouvable ou pas. La première est celle des règles structurelles. Comment décider à quel moment les appliquer, et sur quelle formule ? La seconde est posée par des règles "doubles" comme disj-d, par exemple. Comment décider, étant donnée une formule $A \vee B$, si poursuivre notre recherche de preuve par rapport à A ou bien par rapport à B ? Pour résoudre ces difficultés on considère une autre version du calcul de séquents, dite système **G**, *qui est bien adapté à la recherche automatique des preuves.*

Le calcul G

On obtient le calcul **G** en modifiant le système **LK**-$\{coupure\}$ comme suit :

- On enlève les règles d'affaiblissement mais on élargit l'ensemble d'axiomes à tous les séquents de la forme

  $$\frac{}{\Gamma \vdash \Delta}$$

  où Γ et Δ contiennent au moins une formule atomique commune. (Ceci revient à ne faire des affaiblissements qu'aux feuilles des arbres de preuve.)

- On élimine les règles de contraction en tant que règles structurelles.

- On modifie les règles de disjonction-droite en les remplaçant par :

  $$\frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}$$

Résultat 22 Le système G a la propriété de *réversibilité* : toute règle est telle que la formule associée au séquent conclusion est logiquement équivalent à la conjonction des formules associées aux prémisses.

La preuve de ce résultat est un exercice simple. On observera que **LK** n'a pas la propriété de réversibilité; par exemple, chacune des deux règles disj-d de **LK** est telle que sa prémisse *n'est pas* une conséquence logique de la conclusion.

En utilisant le résultat 22 on peut facilement prouver le théorème suivant.

Théorème 23 Le calcul **G** est un système de preuve correct : s'il existe une preuve du séquent Σ alors Σ est valide.

Preuve. Soit π une preuve de Σ en **G**. On fait une récurrence sur la hauteur h de cette preuve (c'est à dire l'hauteur de l'arbre π).
Base. $h = 1$. Dans ce cas π est un arbre qui consiste seulement d'une feuille axiome. Donc π est $\overline{\Sigma}$ et le séquent Σ a la forme $\Gamma, A \vdash \Delta, A$. Il est clair que ce séquent est valide.
Etape de récurrence. $h > 1$. Si la règle d'inférence r la plus proche à la racine Σ de π a une prémisse, alors π a la forme :

$$\frac{\pi_1}{\Sigma.}\,(r)$$

Si la règle d'inférence r la plus proche à la racine Σ de π a deux prémisse, alors π a la forme :

$$\frac{\pi_1 \qquad\qquad \pi_2}{\Sigma.}\,(r)$$

Dans les deux cas, il s'agit simplement d'appliquer l'hypothèse de récurrence. Par exemple, considérons le deuxième cas. Puisque les hauteurs respectives de π_1 et π_2 sont $< h$, par hypothèse de récurrence les deux séquents qui sont les conclusions respectives de ces deux sous-arbres, disons Σ_1 et Σ_2 sont valides. Par le résultat de réversibilité 22 Σ est donc valide.

De plus, **G** est un système permettant de prouver *tout* séquent valide, comme il est dit par le théorème suivant. Pour le prouver, on a besoin d'une nouvelle notion, celle d'*arbre de déduction systématique.*

D'abord, observons que la prouvabilité d'un séquent $\Gamma \vdash \Delta$ en **G** n'est pas sensible à l'ordre d'application des règles d'inférence. (Mais on ne prouvera pas ce résultat ici). Par exemple, si un séquent de la forme $\vdash A \wedge B, C \vee D$ est prouvable dans **G**, il existera une preuve où la règle la plus proche de la racine est conj-d et aussi une preuve où la règle la plus proche de la racine est une disj-d. Ainsi, si on cherche une preuve de $\Gamma \vdash \Delta$, on peut voir le multi-ensemble Γ, Δ comme une *liste* et traiter les formules de la gauche vers la droite, par exemple.

Un *arbre de déduction systématique* d'un séquent $\Gamma \vdash \Delta$ un arbre de racine $\Gamma \vdash \Delta$ engendré en utilisant systématiquement les règles d'inférence de **G**, pour "décomposer" les formules de Γ, Δ, en allant de la gauche vers la droite. On n'arrête la construction d'une branche que dans deux cas : ou bien on est arrivé à un axiome ou bien à un séquent ne contenant que des formules atomiques (dit "séquent atomique"). On peut remarquer que, étant donné un séquent Σ, il existe un *seul* arbre systématique de déduction T ayant Σ comme racine et que cet arbre est forcément *fini.*

Théorème 24 Le calcul **G** est un système de preuve complet, c'est à dire que si le séquent Σ est valide, alors il existe une preuve de Σ en **G**.

Preuve. Soit Σ un séquent valide. Construisons, à partir de la racine, un arbre de de déduction systématique T de racine Σ. Par le résultat de réversibilité 22, la conjonction des toutes les formules associées à ces feuilles doit être valide. Ceci implique que chaque feuille doit être valide. Maintenant, supposons qu'une feuille ne soit pas un axiome. Par définition d'arbre de déduction systématique, elle doit être un séquent atomique de la forme $G_1, \cdots, G_n \vdash D_1, \cdots, D_m$ avec toute G_i différente de toute D_j. Alors l'interprétation $\mathcal{I}$ telle que $\mathcal{I}(G_i) = V, \mathcal{I}(D_j) = F$, pour tout i, j est telle que le séquent de cette feuille est faux par rapport à $\mathcal{I}$. Ceci contredit le fait que cette feuille est valide. Il en suit que toute feuille de T est un axiome et que T est donc une preuve de Σ.

On observera que la technique utilisée dans la démonstration ci-dessus montre aussi comment calculer une interprétation qui rend faux Σ quand Σ n'est pas valide. Il suffit de construire un arbre de de déduction systématique T de racine Σ. Puisque Σ n'est pas valide, il doit exister au moins une feuille qui n'est pas un axiome. Toute interprétation qui rend cette feuille fausse rend faux Σ aussi.

La correction et complétude de **G** et le fait que tout arbre de déduction systématique est **fini** nous fournissent un algorithme permettant de *décider* si une formule A est valide ou pas. (C'est-à-dire, un algorithme qui, pour l'entrée A, termine toujours, en donnant une réponse "Oui" si A est valide, et "NON" sinon). En effet, étant donnée une formule A, on peut construire l'arbre de déduction systématique T pour le séquent $\vdash A$. Si T est une preuve de $\vdash A$, alors, par la correction de **G**, le séquent $\vdash A$ est valide, ce qui implique que la formule A est valide. Sinon, $\vdash A$ n'est pas prouvable. Dans ce cas, par la complétude de **G**, il n'est pas valide, ce qui implique que la formule A n'est pas valide.

Enfin, quelque remarque sur le liens entre **LK** et **G**. On dit que deux calculs de séquents sont équivalents s'ils ont exactement les mêmes théorèmes. Or, le théorème d'élimination des coupures nous dit que le calcul **LK** est équivalent à **LK**-*coupure*. Il n'est pas dur de montrer que **LK** -*coupure* est équivalent à **G** (mais on ne le fera pas ici). Donc **LK** et **G** sont équivalents; les deux sont des systèmes de preuve correctes et complets pour la logique propositionnelle. Le premier permet - entre autre - de s'adapter aisément aussi à des logiques "non-classiques" (par exemple la logique dite "intuitionniste", que l'on ne traitera pas dans ce cours). L'intérêt du deuxième est par rapport aux problèmes de déduction automatique.

2.3.2 Système de Preuve par Résolution

Formes Normales en Logique Propositionnelle

Le système de preuve par résolution manipule des formules qui ont une forme spécifique, dite *clausale.*

Définition 25

- Un littéral est ou bien une formule atomique ou bien la négation d'une formule atomique.
- Une clause est une formule de la forme $l_1 \vee \cdots \vee \cdots l_m$, où $m > 0$, et chaque l_i est un littéral.
- Une formule F est dite en forme normale conjonctive (f.n.c.) si elle est de la forme $D_1 \wedge \cdots \wedge D_k$, où $k > 0$ et chaque D_i est une clause.
- Une conjonction élémentaire est une formule de la forme $l_1 \wedge \cdots \wedge \cdots l_n$, où $n > 0$ et chaque l_i est un littéral.
- Une formule F est dite en forme normale disjonctive (f.n.d.) si elle est de la forme $C_1 \vee \cdots \vee C_q$, où $q > 0$ et chaque C_i est une conjonction élémentaire.

EXEMPLES

On utilisera la convention suivante : une disjonction "vide" (qui ne contient aucun littéral) est aussi considérée comme un type particulier de clause, même si ce n'est pas une formule. En effet, on a déjà observé qu'une disjonction vide peut être assimilée à la formule $\mathcal{F}$, qui est *insatisfiable.* Dans le contexte des déductions par résolution, on utilisera le symbole $\emptyset$ pour indiquer la clause vide; cette clause joue un rôle spécial. Observons que si une clause insatisfiable est conséquence logique d'un ensemble de clauses E, alors forcément E est insatisfiable (et réciproquement). Donc pour prouver qu'un ensemble de clauses E est insatisfiable, il suffit de montrer que $\emptyset$ est conséquence logique de E. Comme l'on verra, ceci est le principe qui guide la construction des déductions par résolution. Mais avant de pouvoir décrire ce type de déductions on a besoin de quelque résultat préliminaire.

Théorème 26 Etant donnée une formule quelconque F :

- il existe une formule F' qui est en f.n.c. et qui est logiquement équivalente à F;
- il existe une formule $F"$ qui est en f.n.d. et qui est logiquement équivalente à F.

La preuve de ce résultat, par récurrence sur la structure de F, est laissée en exercice. La preuve utilise les équivalences logiques suivantes :

- Loi de Double Négation
- Lois de De Morgan
- Lois de Distributivité

La structure de la preuve indique un *algorithme* qui, pour l'entrée F, permet de construire une formule F' en f.n.c. qui est logiquement équivalente à F et une formule F" en f.n.d. qui est logiquement équivalente à F.

Le résultat suivant nous permettra de ne manipuler que des clauses dès que on voudra utiliser le système de résolution pour prouver des propriétés de validité (ou de conséquence logique) des formules. (Rappelons que pour prouver qu'une formule G est une conséquence logique d'un ensemble de formule E il suffit de prouver que $E \cup \{\neg G\}$ est insatisfiable; en particulier, pour prouver que G est valide il suffit de prouver que G est insatisfiable.)

Résultat 27 Etant donné un ensemble E de formules propositionnelles, il existe un ensemble de clauses C_E tel que E est satisfiable si et seulement si C_E est satisfiable.

Preuve
Soit $E = \{F_1, \cdots F_n\}$. Le théorème 26 nous assure du fait que, étant donné n'importe quelle formule $F_i \in E$, il existe une formule $F_i' = D_{i1} \wedge \cdots \wedge D_{ik}$ qui est en f.n.c. et qui est logiquement équivalente à F_i. Soit C_{F_i} l'ensemble de clauses $\{D_{i1} \cdots D_{ik}\}$ et soit C_E l'ensemble de clauses $\bigcup_{1 \leq i \leq n} C_{F_i}$. Alors, pour toute interprétation $\mathcal{I}$,
I est un modèle de E
ssi
pour tout $i, 1 \leq i \leq n$, $\mathcal{I}$ est un modèle de F_i
ssi
pour tout $i, 1 \leq i \leq n$, $\mathcal{I}$ est un modèle de chaque conjoint D_{ij} de F_i
ssi
pour tout $i \leq i \leq n$, $\mathcal{I}$ est un modèle de l'ensemble de clauses C_{F_i}
ssi
$\mathcal{I}$ est un modèle de C_E.
Donc E et C_E ont exactement les mêmes modèles. Ceci implique que E a un modèle si et seulement si C_F a un modèle.
EXEMPLE

Preuves par Résolution en Logique Propositionelle

On a déjà dit que ce système n'agit que sur des clauses. Dans la suite, on identifiera une clause $l_1 \vee \cdots l_n$ avec toute clause $l_{i1} \vee \cdots l_{in}$ obtenue en changeant l'ordre de ses littéraux. Ainsi $p \vee q \vee r$ sera identifiée avec $p \vee r \vee q$, par exemple. Par contre, $p \vee p$ n'est pas la même clause que p (tout en lui étant logiquement équivalente).

Le système de Preuve par Résolution ne contient que deux règles d'inférence :

1. $$\frac{C \vee \neg p, \quad C' \vee p}{C \vee C'}$$
   où C, C' sont deux clauses et p est une variable propositionnelle. C'est la règle de résolution, qui donne le nom au système.

2. $$\frac{C \vee l \vee l}{C \vee l}$$
   où C est une clause et l un littéral. Cette règle, appelée factorisation, permet d'éliminer les répétitions de littéraux.

Définition 28 Soient E un ensemble de clauses et c une clause. Une déduction par résolution de c à partir des hypothèses E est une suite de clauses $c_1, \cdots c_n$ telle que :

- $c_n = c$
- Pour tout i, $1 \leq i \leq n$, on a un des trois cas suivants :
  1. $c_i \in E$
  2. $\exists j \leq i$ tel que c_i est obtenue à partir de c_j en appliquant la règle de factorisation
  3. $\exists j, j' \leq i$ tels que c_i est obtenue à partir de $c_j, c_{j'}$ en appliquant la règle de résolution.

La longueur de la déduction $c_1, \cdots c_n$ est n.

EXEMPLES DE DEDUCTIONS PAR RESOLUTIONS.
EXEMPLES DE LA POSSIBILITE D'ECRIRE LES DEDUCTIONS SOUS FORME DE GRAPHES.

Définition 29 Soit E un ensemble de clauses. Une réfutation de E est une déduction par résolution de $\emptyset$ à partir de l'ensemble d'hypothèses E.

Résultat 30 Les règles de résolution et factorisation sont correctes, c'est à dire que, pour toute interprétation $\mathcal{I}$, si la conjonction des prémisses est vraie par rapport à $\mathcal{I}$, alors la conclusion est vraie aussi.

La preuve de ce résultat est laissée en exercice. Le résultat 30 permet de démontrer le résultat suivant :

Résultat 31 Soit E un ensemble fini de clauses et c une clause. S'il existe une déduction par résolution de c à partir de E, alors $E \models c$.

Preuve
Soit δ une déduction de c à partir de E. On montre le résultat par récurrence sur la longueur n de δ.
Base. $n = 1$. Dans ce cas E est l'ensemble qui ne contient q'une seule clause, c'est à dire c; il est clair que $c \models c$.
Etape de récurrence. $n > 1$. Soit $c_1, \cdots c_n$, avec $c = c_n$, la déduction δ. On a trois cas.

1. $c_n \in E$.
   Dans ce cas, E a la forme $E' \cup \{c_n\}$. Il est évident que c_n est une conséquence logique de $E' \cup \{c_n\}$.
2. $\exists j \leq n$ tel que c_n est obtenue à partir de c_j en appliquant la règle de factorisation.
   Par hypothèse de récurrence, $E \models c_j$. D'autre part, par le résultat 30, on a $c_i \models c_n$. Il en suit que $E \models c_n$.

3. $\exists j, j' \leq i$ tels que c_i est obtenue à partir de $c_j, c_{j'}$ en appliquant la règle de résolution.
   Par hypothèse de récurrence, $E \models c_j$ et $E \models c_{j'}$. Donc $E \models c_j \wedge c_{j'}$. D'autre part, par le résultat 30, on a $c_j \wedge c_{j'} \models c_n$. Il en suit que $E \models c_n$.

Or, si on considère le cas particulier où $c_n = \emptyset$, le résultat 31 nous donne le théorème suivant

Théorème 32 (**Correction du système de preuve par résolution**). Soit C un ensemble de clauses. S'il existe une réfutation de C, alors C est insatisfiable.

Le théorème ci-dessus suggère une méthode permettant de prouver qu'une formule A est conséquence logique d'un ensemble E de formules (et, en particulier, de prouver qu'une formule A est valide). En effet, on sait que $E \models A$ si et seulement si $E \cup \{\neg A\}$ est insatisfiable. Or, par le résultat 27, on peut construire un ensemble de clauses $C_{E,\neg A}$ tel que $E \cup \{\neg A\}$ est insatisfiable si et seulement si $C_{E,\neg A}$ est insatisfiable. On réduit donc le problème de prouver que $E \models A$ au problème de l'existence d'une réfutation r de $C_{E,\neg A}$, car le théorème 32 nous assure que si une telle réfutation existe, alors $C_{E,\neg A}$ est forcément insatisfiable et on peut donc correctement conclure, à partir de l'existence d'une telle réfutation, que $E \models A$. C'est dans ce sens que le système de preuve par réfutation est *correct*.

Est-il *complet* ? C'est-à-dire : est-il vrai que si $E \models A$ alors on peut toujours prouver ce fait par résolution, en suivant la méthode indiquée ci-dessus ? En autres mots, est-il vrai que si $E \cup \{\neg A\}$ est insatisfiable alors il existe forcément une réfutation de l'ensemble de clauses $C_{E,\neg A}$? Oui, comme énoncé par le théorème suivant, que l'on ne démontrera pas ici. (La preuve est une version simplifiée du théorème analogue pour le calcul des prédicats, qui est prouvé dans le chapitre suivant).

Théorème 33 (**Complétude du système de preuve par résolution**). Soit C un ensemble de clauses. SI C est insatisfiable, alors il existe une réfutation de C.

3 Calcul des Prédicats (du premier ordre)

On a vu que la notion de conséquence logique permet de faire une étude formelle de la correction de certains arguments. Mais si on se limite au cadre de la logique propositionnelle on n'arrivera pas á exprimer la correction de certaines arguments pourtant très intuitifs.

EXEMPLES

3.1 Syntaxe du Calcul des prédicats

BUT : Comme dans le cas propositionnel, ici on s'occupe exclusivement de spécifier quelles expressions sont considérées, indépendamment de leur signification et de leur vérité.

Définition 34 Un alphabet $\mathcal{A}$ d'un langage $\mathcal{L}_{CP}$ du calcul des prédicats consiste des données suivantes :

- L'ensemble de connecteurs logique contenant le connecteur à un argument $\neg$ et les connecteurs binaires $\wedge, \vee, \rightarrow, \leftrightarrow$.
- Les constantes logiques $\mathcal{V}$ et $\mathcal{F}$
- Les parenthèses $($, $)$ et la virgule.
- Le symbole $\forall$, dit quantificateur universel et le symbole $\exists$, dit quantificateur existentiel.
- Un ensemble infini dénombrable de symboles de variables individuelles $X = \{x, y, z...\}$
- Un ensemble dénombrable $F = \{f, g, h, ...\}$ de symboles de fonction, chacun muni de son arité.
- Un ensemble dénombrable $R = \{p, q, r...\}$ de symboles de prédicats (ou relations), chacun muni de son arité.

Les deux dernières composantes d'un alphabet d'un langage du calcul des prédicats constituent la *signature* de ce langage.

Le notions de *terme* sur F, X et *atome* sur R, F, X ayant été déjà définies dans le premier cours (voir la section 1), on peut tout de suite définir la notion de formule d'un langage du calcul des prédicats.

Définition 35 Les expressions bien formées ou "formules" d'un langage $\mathcal{L}_{CP}$ sont les mots sur l'alphabet de $\mathcal{L}_{CP}$ définis récursivement comme suit :

- Tout atome sur R, F, X est une formule, dite aussi formule *atomique*; les expressions $\mathcal{V}$ et $\mathcal{F}$ sont aussi des formules atomiques.
- Si F_1 et F_2 sont des formules, alors $(\neg F_1), (F_1 \wedge F_2), (F_1 \vee F_2), (F_1 \rightarrow F_2), (F1 \leftrightarrow F_2)$ sont aussi des formules.

- Si F est une formule et x une variable, alors $(\forall x F)$ et $(\exists x F)$ sont des formules; la formule F est dite être "la portée" de $\forall$, resp. $\exists$.

On observera qu'un langage propositionnel est un cas particulier de langage du calcul des prédicats où la signature ne contient pas de symboles de fonction et tout prédicat est de arité 0 (du coup, variables et quantificateurs sont inutiles).

On suivra la même convention d'élimination des parenthèses inutiles que pour le cas propositionnel. En plus, on aura les deux conventions suivantes :

1. Si Q, Q' sont des quantificateurs, on aura le droit d'écrire $QxQy(F)$ à la place de $Qx(Qy(F))$. Par exemple, on écrira $\exists z \exists y \forall x(r(x,y) \wedge p(z))$ à la place de $\exists z(\exists y(\forall x(r(x,y) \wedge p(z))))$

2. si F est atomique, on aura le droit d'écrire $\exists x F$ à la place de $\exists x(F)$ et $\forall x F$ à la place de $\forall x(F)$. Par exemple, on aura le droit d'écrire $\exists x p(x) \vee r(x)$ à la place de $\exists x(p(x)) \vee r(x)$.

Définition 36 (Variables libres et liées) Soit F une formule. Les ensembles des variables libres de F et l'ensemble des variables liées de F, notées respectivement $Varlib(F)$ et $Varlie(F)$ sont définis par récurrence sur la structure de F :

- Si F est un atome, $Varlib(F)$ est l'ensemble de toutes les variables qui ont au moins une occurrence en F et $Varlie(F) = \emptyset$.

- Si F est $\neg F_1$, $Varlib(F) = Varlib(F_1)$ et $Varlie(F) = Varlie(F_1)$.

- Si F est $F_1 conn F_2$, où *conn* est un connecteur binaire, $Varlib(F) = Varlib(F_1) \cup Varlib(F_2)$ et $Varlie(F) = Varlie(F_1) \cup Varlie(F_2)$.

- Si F est $Qx(F_1)$, où Q est un quantificateur, $Varlib(F) = Varlib(F_1) - \{x\}$ et $Varlie(F) = Varlie(F_1) \cup \{x\}$.

On remarquera q'une même variable x peut appartenir, en même temps, à $Varlib(F)$ et à $Varlie(F)$. Par exemple, x est en même temps libre et liée dans la formule $\forall x(p(x) \wedge q(x)) \vee r(x)$, car la première occurrence de x est liée par le quantificateur mais la deuxième occurrence n'est pas dans la portée du quantificateur.

3.2 Sémantique du calcul des prédicats

BUT Donner un signification aux symboles de prédicat et de fonction et fixer les domaine dans lequel les variable prennent valeur, afin de pouvoir établir la valeur de vérité (Vrai ou Faux) des formules.

Considérons la formule $\exists y \; \forall x \; r(y,x)$. Est-elle vraie ou fausse ? Tout dépend de comme on "lit" le symbole r et du domaine de discours considéré. Par exemple, supposons que le domaine de discours soit l'ensemble N des entiers ≥ 0. Si on lit r comme la relation $\leq$ sur N, alors la formule est vraie. Si on lit r comme la relation $<$ sur N, alors la formule est fausse, car $0 \not< 0$. Maintenant, supposons que le domaine discours soit l'ensemble Z des entiers. Peut importe si on lit r comme $\leq$ ou $<$ (sur Z) : la formule est en tout cas fausse, car Z n'a pas de minimum. On voit donc que pour pouvoir parler de la valeur de vérité d'une formule du calcul des prédicats il faut d'abord se fixer un domaine de discours, associer des relations sur ce domaine aux symboles de prédicats et des fonctions sur ce domaine aux symboles de fonction. Afin de pouvoir faire cela proprement, on introduit les notions de *structure* et de *interprétation* (en calcul des prédicats). On rappelle qu'une relation à n arguments sur un domaine D n'est rien d'autre qu'un ensemble de n-uplets de D. Par exemple, si $D = N$, la relation binaire $<$ sur N est rien d'autre que l'ensemble $\{< n, m > \mid n, m \in N$ et n est plus petit que $m\}$.

Définition 37 On appelle structure tout triplet $\langle D, F_D, R_D \rangle$ où D est un ensemble non vide appelé domaine de discours, F_D est un ensemble de fonctions sur D et R_D un ensemble de relations sur D.

Exemple 38 Le triplet $\langle D, F_D, R_D \rangle$ où $D = N$, $F_D = \{+, \times, 0, \mathit{successeur}\}$, $R_D = \{=, \leq\}$ est une structure, connue comme "structure de l'arithmétique de Peano". Observer que le nombre 0 est ici vu comme une fonction à 0 arguments. En général, tout élément "distingué" du domaine d'une structure est vu comme une fonction à 0 arguments de la structure (donc comme un élément de F_D). Un autre exemple de structure est donné par D=l'ensemble des nombres réels, $F_D = \{+, -, \times, :, 0, 1\}$, $R_D = \emptyset$: c'est une structure dans laquelle on peut interpréter les expressions arithmétiques. Ici, les éléments distingués 0 et 1 sont vus comme des fonctions à 0 arguments.

On remarquera que quand $R_D = \emptyset$, une structure n'est rien d'autre qu'une algèbre.
Donner une signification aux symboles de relation et de fonction d'un langage L du calcul de prédicats ayant une certaine signature revient à lui associer une structure. Ceci motive la définition suivante.

Définition 39 Une interprétation $\mathcal{I}$ d'un langage L du calcul de prédicats ayant une signature $\langle F, R \rangle$ est la donnée de :

- Une structure $\langle D, F_D, R_D \rangle$. L'ensemble D est appelé domaine de discours de $\mathcal{I}$.
- Pour tout n et tout symbole de fonction $f \in F$ d'arité n, une assignation d'une fonction à n arguments (élément de F_D) à f; cette fonction est notée $\mathcal{I}(f)$.
- Pour tout n et tout symbole de relation $r \in R$ d'arité n, une assignation d'une relation à n arguments (élément de R_D) à r; cette relation est notée $\mathcal{I}(r)$.

Une fois définie la notion d'interprétation, le premier problème à considérer est comment évaluer un terme t. Considérons par exemple le terme $f(a, g(a))$, où f est un symbole de fonction à 2 argument, g est un symbole de fonction à 1 argument et a est une constante, çest-à-dire un symbole de fonction à 0 arguments. Le rôle d'un terme est celui d'indiquer un individu, c'est à dire un élément d'un domaine de discours. Quel élément ? Tout d'abord, ceci dépend de l'interprétation choisie. Par exemple, soit $\mathcal{I}$ l'interprétation du langage de t où la structure considérée est celle de l'arithmétique de Peano, $\mathcal{I}(f) = +, \mathcal{I}(g) = \mathit{successeur}, \mathcal{I}(a) = 0$. Par rapport à cette interprétation, le terme $f(a, g(a))$ indique la somme de 0 et du successeur de 0, donc le nombre 1. Maintenant, considérons le terme $f(x, g(x))$. Quelle est l'évaluation du terme par rapport à $\mathcal{I}$? Ceci dépend de l'entier associé à x... Une définition précise de la valeur d'un terme t dont les variables sont $x_1, \cdots, x_n$, par rapport à une interprétation $\mathcal{I}$, doit donc tenir compte des éléments du domaine de discours $d_1, \cdots, dn$ que l'on a choisi d'associer aux variables.

Définition 40 (Choix de valeurs pour les variables)

- Soit L un langage du calcul des prédicats et soit $X = \{x_1, x_2, \ldots\}$ l'ensemble des variables individuelles de L. Soit $\mathcal{I}$ une interprétation pour L ayant domaine D. Un choix de valeurs ch par rapport à $\mathcal{I}$ est une fonction $ch : X \rightarrow D$.
- Etant donnée une choix de valeurs ch et un objet $d \in D$, la notation $ch[x_i := d]$ indique le choix de valeurs ch' tel que :
  $ch'(y) = ch(y)$ pur toute variable y distincte de x_i et $ch'(x_i) = d$.

Exemple 41 Par exemple, supposons que le domaine D de l'interprétation $\mathcal{I}$ considérée soit l'ensemble $\{3, 8, 9\}$. La fonction ch telle que :

- $ch(x_1) = 9, ch(x_2) = 3, ch(x_3) = 8$
- pour tout $i > 3$, $ch(x_i) = 8$ si i est impair, $ch(x_i) = 9$ sinon

est un choix de valeurs par rapport à $\mathcal{I}$.
Dans ce cas, la notation $ch[x_2 := 9]$ indique le choix de valeurs ch' telle qui ne diffère de ch que pour la valeur de l'argument x_2; $ch'(x_2) = 9$.

Définition 42 (Evaluation d'un terme) Soit t un terme. Soit $\mathcal{I}$ une interprétation du langage de t par rapport à une structure ayant D comme domaine de discours. Soit ch un choix de valeurs par rapport à $\mathcal{I}$. L'évaluation de t par rapport à $\mathcal{I}$ et au choix de valeurs ch, notée $[t]_{\mathcal{I},ch}$, est définie comme suit.

- Cas où $t = x_i$.
  $[x_i]_{\mathcal{I},ch} = ch(x_i)$
- Cas où $t = c$, c étant une constante (un symbole de fonction à 0 arguments).
  $[c]_{\mathcal{I},ch} = \mathcal{I}(c)$.
- Cas où $t = f(t_1, \cdots, t_k)$, où $k \geq 1$.
  $[f(t_1, \cdots, t_k)]_{\mathcal{I},ch} = \mathcal{I}(f)([t_1]_{\mathcal{I},ch}, \cdots, [t_k]_{\mathcal{I},ch})$.

De même, il se pose le problème de donner une valeur de vérité à une formule F par rapport à une interprétation $\mathcal{I}$ donnée. A nouveau, il faut gérer le problème des variables libres de F. Par exemple, soit $F = \forall x r(y, x)$. Si on considère l'interprétation $\mathcal{I}$ dans la structure de l'arithmétique de Peano telle que $\mathcal{I}(r) =\leq$, la formule dit que l'entier y est inférieur ou égal à tout élément de N. Elle est donc vraie par rapport à tout choix de valeurs qui associe l'entier 0 à y, mais fausse par rapport à tout autre choix. La définition suivante, qui dit comme associer un élément de BOOL à une formule, fait donc référence à un choix de valeurs pour les variables.

Définition 43 (Evaluation d'une formule) Soit A une formule, soit $\mathcal{I}$ une interprétation du langage de A par rapport à une structure ayant D comme domaine de discours. Soit ch un choix de valeurs par rapport à $\mathcal{I}$. L'évaluation de A par rapport à $\mathcal{I}$ et au choix de valeurs ch, notée $[A]_{\mathcal{I},ch}$, est définie récursivement comme suit.

- Cas où A est un atome $r(t_1, \cdots, t_k)$.
  $[r(t_1, \cdots, t_k)]_{\mathcal{I},ch} =$

  - V si le n-uplet d'éléments de D $\langle [t_1]_{\mathcal{I},ch}, \cdots, [t_k]_{\mathcal{I},ch} \rangle$ appartient à la relation $\mathcal{I}(r)$
  - F sinon.

- Cas où A est $\mathcal{V}$ ou bien $\mathcal{F}$.
  $[\mathcal{V}]_{\mathcal{I},ch} = V$ et $[\mathcal{F}]_{\mathcal{I},ch} = F$.

- Cas où A est $\neg A_1$.
  $[\neg A_1]_{\mathcal{I},ch} = f_\neg([A_1]_{\mathcal{I},ch})$.

- Cas où A est A_1 *conn* A_2, où *conn* est un connecteur binaire.
  $[A_1 \; conn \; A_2]_{\mathcal{I},ch} = f_{conn}([A_1]_{\mathcal{I},ch}, [A_2]_{\mathcal{I},ch})$.

- Cas où A est $\forall x(A_1)$.
  $[\forall x(A_1)]_{\mathcal{I},ch} =$
  - V si pour tout élément $d \in D$, $[A_1]_{\mathcal{I},ch[x:=d]} = V$,
  - F sinon.

- Cas où A est $\exists x(A_1)$.
  $[\exists x(A_1)]_{\mathcal{I},ch} =$
  - V s'il existe un élément $d \in D$ tel que $[A_1]_{\mathcal{I},ch[x:=d]} = V$,
  - F sinon.

Exemple 44 Soit F la formule $\forall y \; r(x,y)$. Soit $\mathcal{I}$ une interprétation telle que le domaine est N et $\mathcal{I}(r)$ est $\leq$ (sur N). Ci-dessous, on utilise la définition 43 pour montrer formellement que, étant donné n'importe quel choix de valeurs ch **tel que** $ch(x) = 0$, la formule F est "vraie" par rapport à ch.
$[\forall y \; r(x,y)]_{\mathcal{I},ch} = V$
ssi
pour tout $m \in N$, $[r(x,y)]_{\mathcal{I},ch[y:=m]} = V$
ssi
pour tout $m \in N$, le couple d'éléments de N $\langle [x]_{\mathcal{I},ch[y:=m]}, [y]_{\mathcal{I},ch[y:=m]} \rangle$ est dans la relation $\leq$
ssi
pour tout $m \in N$, $0 \leq m$.
Or, il est évidemment vrai que 0 est inférieur ou égal à tout élément de N. Donc $[\forall y \; r(x,y)]_{\mathcal{I},ch} = V$.
Maintenant, considérons la formule $\forall x \forall y \; r(x,y)$. On va montrer que, pour n'importe quel choix de valeurs ch, $[\forall x \forall y \; r(x,y)]_{\mathcal{I},ch} = F$. Soit donc ch un choix de valeurs quelconque.
$[\forall x \forall y \; r(x,y)]_{\mathcal{I},ch} = V$
ssi pour tout $n \in N$, $[\forall y \; r(x,y)]_{\mathcal{I},ch[x:=n]} = V$
ssi
pour tout $n \in N$, pour tout $m \in N$ $[r(x,y)]_{\mathcal{I},ch[x:=n][y:=m]} = V$
ssi
pour tout $n \in N$, pour tout $m \in N$, le couple $\langle [x]_{\mathcal{I},ch[x:=n][y:=m]}, [y]_{\mathcal{I},ch[x:=n][y:=m]} \rangle$ est dans la relation $\leq$

ssi
pour tout $n \in N$, pour tout $m \in N$, $n \leq m$.
Il suffit de prendre $n \neq 0$, $m = 0$, par exemple, pour contredire ce dernier énoncé. Donc, $[\forall x \forall y\ r(x,y)]_{\mathcal{I},ch} \neq V$.

Résultat 45 Il est facile de prouver les faits suivants (même si c'est long et rébarbatif) :

1. Soient $x_1, \cdots, x_n$, où $n \geq 0$, toutes et seules les variables *libres* d'une formule A. Soit $\mathcal{I}$ une interprétation quelconque et soient ch et ch' deux choix de valeurs pour les variables qui ne différent, éventuellement, que par rapport à des variables qui n'appartiennent pas à $\{x_1, \cdots, x_n\}$ (en d'autres termes, pour tout i tel que $1 \leq i \leq n, ch(x_i) = ch'(x_i)$). Alors $[A]_{\mathcal{I},ch} = V$ ssi $[A]_{\mathcal{I},ch'} = V$. Ceci signifie que seulement les valeurs attribuées aux variables *libres* de A sont *pertinentes* par rapport au résultat de l'évaluation de A.

2. En particulier, si A ne contient pas de variable libres (ou, comme on dit, si elle est *close*), alors, étant donnée une interprétation quelconque $\mathcal{I}$ et deux choix de valeurs pour les variables *quelconques* ch et ch', on a : $[A]_{\mathcal{I},ch} = V$ si et seulement si $[A]_{\mathcal{I},ch'} = V$. En d'autres termes, *si une formule A est close, le choix de valeurs fait pour les variables n'a aucune influence sur la valeur de vérité de A... Ou bien A est vraie pour n'importe quel choix de valeurs, ou bien A est fausse pour n'importe quel choix de valeurs.*

Le résultat ci-dessus légitime la **convention d'écriture** suivante. Si $x_1, \cdots, x_n$ sont toutes et seules les variables de la formule A, et, pour tout i tel que $1 \leq i \leq n$ $ch(x_i) = d_i$, on aura le droit d'écrire $[A]_{\mathcal{I},x_1=d_1,\cdots,x_n=d_n} = ...$ pour indiquer la valeur de vérité de A par rapport à $\mathcal{I}$ et à ch. En particulier, si A est close, on aura le droit d'écrire $[A]_{\mathcal{I}} = \mathcal{V}$ (resp.$[A]_{\mathcal{I}} = \mathcal{F}$) pour indiquer que la valeur de vérité de A est $\mathcal{V}$ (resp. $\mathcal{F}$) pour un choix de valeurs quelconque ch.

Définition 46 (vérité, modèle)

- On dit qu'une formule A est vraie par rapport à une interprétation $\mathcal{I}$ si, $[A]_{\mathcal{I},ch} = V$ pour *tout* choix de valeur ch.
- On dit qu'une interprétation $\mathcal{I}$ est un modèle d'une formule A si A est vraie par rapport à $\mathcal{I}$; dans ce cas, on note $\mathcal{I} \models A$. On dit que $\mathcal{I}$ est un modèle d'un ensemble de formules E si elle est modèle de tout élément de E.

Exemple 47 La deuxième partie de l'exemple 44 montre que la formule $\neg\forall y \forall x r(x,y)$ est vraie par rapport à l'interprétation $\mathcal{I}$ ayant N comme domaine et telle que $\mathcal{I}(r) =\leq$. Si on utilise la convention d'écriture indiquée plus haut, ce que l'exemple a montré est le fait suivant : ce n'est pas le cas que pour tout n et pour tout m, m, $r(x,y)_{\mathcal{I},x=n,y=m} = V$.

Remarquez que le résultat 45 nous dit qu'une formule *close* A est ou bien vraie ou bien fausse par rapport à une interprétation donnée $\mathcal{I}$. Par contre, si A contient des variables libres, elle peut avoir la valeur de vérité $\mathcal{V}$ par rapport à certaines choix de valeurs pour les variables et $\mathcal{F}$ par rapport à des autres; dans ce cas, A n'est ni vraie ni fausse par rapport à $\mathcal{I}$ (ceci est le cas, par exemple, de la prèmiere formule de l'exemple 44)

Une fois donnée la définition de modèle d'une formule du calcul des prédicats, les définitions de validité, satisfiabilité et insatisfiabilité sont les mêmes que pour le cas propositionnel.

Définition 48 (valide, satisfiable, insatisfiable)
- On dit qu'une formule A est valide si, pour n'importe quelle interprétation $\mathcal{I}$, $\mathcal{I}$ est modèle de A. On notera $\models A$ pour indiquer que A est valide.
- Une formule A (resp. un ensemble de formules E) est satisfiable s'il existe au moins une interprétation $\mathcal{I}$ qui est modèle de A (resp. E); sinon, A (resp. E) est dite (resp. dit)insatisfiable.

Exemple 49 La formule $A = \forall xp(x) \rightarrow \exists yp(y)$ est valide. On le montre en utilisant les définitions ci-dessus.
D'abord, on observe :
A est valide ssi toute interprétation $\mathcal{I}$ est un modèle de A ssi toute interprétation $\mathcal{I}$ et tout choix de valeur ch (relatif à $\mathcal{I}$) sont tels que $[A]_{\mathcal{I},ch} = V$.

Soit donc $\mathcal{I}$ une interprétation, dont le domaine est D et soit ch un choix de valeurs (relatif à $\mathcal{I}$).
$[A]_{\mathcal{I},ch} = V$
ssi
$f_{\rightarrow}([\forall xp(x)]_{\mathcal{I},ch}, [\exists yp(y)]_{\mathcal{I},ch}) = V$
ssi
ou bien $[\forall xp(x)]_{\mathcal{I},ch} = F$, ou bien $[\exists yp(y)]_{\mathcal{I},ch} = V$
ssi
ou bien ce n'est pas le cas que pour tout $d \in D$ $[p(x)]_{\mathcal{I},ch[x:=d]} = V$, ou bien il existe un $d' \in D$ tel que $[p(x)]_{\mathcal{I},ch[x:=d']} = V$
ssi
ou bien il existe un $d \in D$ tel que $[x]_{\mathcal{I},ch[x:=d]} \notin \mathcal{I}(p)$ ou bien il existe un $d' \in D$ tel que $[x]_{\mathcal{I},ch[x:=d']} \in \mathcal{I}(p)$
ssi
ou bien il existe un $d \in D$ tel que $d \notin \mathcal{I}(p)$ ou bien il existe un $d' \in D$ tel que $d' \in \mathcal{I}(p)$.
Or, puisque étant donné un élément quelconque $d \in D$, on a forcement au moins un des deux cas : a) $d \notin \mathcal{I}(p)$, b) $d \in \mathcal{I}(p)$, la dernière disjonction est toujours vraie. Par conséquence, $[A]_{\mathcal{I},ch} = V$

Par contre, la définition de la notion de "conséquence logique' est plus délicate.

Définition 50 (conséquence logique) Soit $E = \{A_1, \cdots, A_n\}$, $n \geq 0$, un ensemble de formules, soit G une formule. On dit que G est conséquence logique de E, et on note $A_1, \cdots, A_n \models G$, si, pour toute interprétation $\mathcal{I}$ *et tout choix de valeur ch* (relatif à $\mathcal{I}$)

$$[A_1 \wedge \cdots \wedge A_n]_{\mathcal{I},ch} = V$$

implique que

$$G_{\mathcal{I},ch} = V$$

Remarque 51 Observer que, *en présence de variables libres*, si G est conséquence logique de E, alors tout modèle de E est aussi un modèle de G, *mais la réciproque est* *fausse*. Par exemple, soit $E = \{p(x)\}$ et soit $G = \forall x p(x)$. Tout modèle de E est un modèle de G. En fait, si $\mathcal{I} \models p(x)$, alors, pour tout choix de valeurs ch, $[p(x)]_{\mathcal{I},ch} = V$, ce qui implique la vérité de $\forall x p(x)$ pour $\mathcal{I}$. Toutefois, $\forall x p(x)$ *n'est pas une conséquence logique de* $p(x)$. En fait, considérons l'interprétation $\mathcal{I}$ où le domaine est l'ensemble des entiers naturels N et la relation associée au symbole p est "être pair". Soit ch un choix de valeurs tel que $ch(x) = 2$. On a :$[p(x)]_{\mathcal{I},ch} = V$, car 2 est bien pair. Toutefois, $[\forall x p(x)]_{\mathcal{I},ch} = F$, car ce n'est pas vrai que, pour tout $n \in N$, $[p(x)]_{\mathcal{I},ch[x:=n]} = V$ (prendre par exemple $n = 3$). Par contre, si E et G ne contiennent que des formules closes, il est vrai que G est conséquence logique de E si et seulement si tout modèle de E est aussi un modèle de G.

Observer que, si E est l'ensemble de formules vide, dire que G est conséquence logique de E revient à dire que G est valide (comme dans le cas propositionnel).

Exactement comme dans le cas propositionnel, on dira que deux formules A_1 et A_2 sont *logiquement équivalentes* si $A_1 \models A_2$ *et* $A_2 \models A_1$. A nouveau, on notera ce fait par $A_1 \equiv A_2$.

Exemple 52

- $\forall x A \equiv \neg \exists x \neg A$ (définissabilité de $\forall$ par $\exists$)
- $\exists x A \equiv \neg \forall x \neg A$ (définissabilité de $\exists$ par $\forall$)
- Soit A' une formule obtenue à partir de A en remplaçant toute occurrence de la variable x par y, où $y \notin Varlib(A)$. Alors, $\forall x(A) \equiv \forall y(A')$ et $\exists x(A) \equiv \exists y(A')$ (principe de renommage)
- $\exists y \forall x r(y,x) \models \forall x \exists y r(y,x)$ mais il est faux que $\exists y \forall x r(y,x) \equiv \forall x \exists y r(y,x)$

3.3 Systèmes de Preuve : Résolution en Calcul des Prédicats

Comme dans le cas propositionnel, on peut considérer au moins deux systèmes de preuves, celui par *séquents* et celui par résolution. Dans les deux cas, il s'agit juste d'étendre le système propositionnel. Nous ne considérerons que celui par résolution.

3.3.1 Formes Normales en Calcul des Prédicats

La définition de *littéral* du calcul des prédicats est la même que pour la logique propositionnelle (c'est seulement la notion de "formule atomique" qui change). La définition de clause est "presque" la même que pour la logique propositionnelle.

Définition 53 (Clause en Calcul des Prédicats) Une clause est une formule de la forme

$$\forall x_1, \cdots, \forall x_n (L_1 \vee \cdots \vee L_q)$$

où $n \geq 0$ et où tout L_i est un littéral.

Remarque 54 Puisque les seuls quantificateurs d'une clause $\forall x_1, \cdots, \forall x_n (L_1 \vee \cdots \vee L_q)$ sont universels, on aura le droit de la noter $L_1 \vee \cdots \vee L_q$ sans aucun risque d'ambiguïté.

Comme dans le cas propositionnel, le système de preuve par résolution du calcul des prédicats ne manipule que des clauses. Comme dans le cas de la logique propositionnelle, on peut transformer n'importe quel ensemble de formules E en un ensemble de clauses C_E tel que E est satisfiable si et seulement si C_E l'est. Mais pour pouvoir définir la méthode de transformation, on a besoin d'introduire des nouvelles notions de *formes normales* qui sont spécifiques au calcul des prédicats.

Définition 55 Une formule A est dite en *forme prénexe* si elle a la forme

$$Q_1 x_1, \cdots, Q_n x_n M$$

où $n \geq 0$, où chaque Q_i est un quantificateur, et où M ne contient pas de quantificateurs. On appelle la sous-formule M *matrice* de A et la suite de quantificateurs $Q_1 x_1, \cdots, Q_n x_n$ *préfixe* de A.

EXEMPLES

Théorème 56 Etant donnée une formule A, il existe une formule A_{pr} qui est logiquement équivalente à A et qui est en forme prénexe.

Preuve Il suffit d'appliquer itérativement les équivalences suivantes pour "faire sortir" tous les quantificateurs de A.

1. $\neg \forall x (A) \equiv \exists x (\neg A)$
2. $\neg \exists x (A) \equiv \forall x (\neg A)$
3. si $x \notin Varlib(A_2)$, $(\forall x (A_1)) \vee A_2 \equiv \forall x (A_1 \vee A_2)$

4. si $x \notin Varlib(A_2)$, $(\exists x(A_1)) \vee A_2 \equiv \exists x(A_1 \vee A_2)$
5. si $x \notin Varlib(A_2)$, $(\forall x(A_1)) \wedge A_2 \equiv \forall x(A_1 \wedge A_2)$
6. si $x \notin Varlib(A_2)$, $(\exists x(A_1)) \wedge A_2 \equiv \exists x(A_1 \wedge A_2)$
7. le principe de renommage (pour pouvoir être toujours dans sous les conditions d'applications des équivalences ci-dessus).

La preuve de ces équivalences est laissée en exercice (voir TD). Remarquez que la commutativité de $\wedge$ et $\vee$ permet d'obtenir une équivalence "symétrique" pour chacune des équivalences 3-6. Par exemple, 3bis est : si $x \notin Varlib(A_1)$, $((A_1)) \vee \forall x A_2 \equiv \forall x(A_1 \vee A_2)$.
EXEMPLES DE TRANSFORMATION D'UNE FORMULE EN UNE FORMULE PRENEXE LOGIQUEMENT EQUIVALENTE.

Définition 57 On dit qu'une formule A est *rectifiée* si

- $Varlib(A) \cap Varlie(A) = \emptyset$
- Si Q_i et Q_j sont deux occurrences distinctes de quantificateurs en A, alors Q_i et Q_j lient deux variables x et y différentes.

EXEMPLES DE FORMULES RECTIFIEES ET NON RECTIFIEES

Remarque 58 Etant donnée n'importe quelle formule A, on peut toujours la transformer en une formule rectifiée logiquement équivalente : il suffit d'utiliser l'équivalence logique dite *principe de renommage.* Par exemple, la formule $(\forall x r(x,y)) \wedge (\exists x \exists y p(x,y))$ n'est pas rectifiée, mais elle est logiquement équivalente à $(\forall x r(x,y)) \wedge (\exists z \exists v p(z,v))$ qui est rectifiée.

Lemme 59 Soit A une formule prénexe rectifiée ayant la forme

$$\forall x_1, \cdots, \forall x_i \exists x_{i+1} Q_{i+2} x_{i+2}, \cdots, Q_{i+n} x_{i+n} M$$

où $i \geq 0$ et $n \geq 1$. Soit f un symbole de fonction à i arguments tel que f n'apparaît pas en M. Soit M' la formule (sans quantification) obtenue à partir de M en remplaçant toute occurrence de x_{i+1} par le terme $t = f(x_1, \cdots, x_i)$. La formule A est satisfiable si et seulement si la formule

$$\forall x_1, \cdots, \forall x_i Q_{i+2} x_{i+2}, \cdots, Q_{i+n} x_{i+n} M'$$

est satisfiable. Le symbole de fonction f est dit *fonction de Skolem*

Idée de la preuve
On illustre la technique qu'il faudrait utiliser pour prouver le lemme ci-dessous sur un exemple très simple. (Dans le cas général, l'idée reste la même, mais la notation est assez lourde). Soit donc A la formule $\forall x \exists y\ r(x,y)$. Soit f un symbole de fonction à 1 argument. On veut montrer que :
A est satisfiable si et seulement si $\forall x r(x, f(x))$ est satisfiable.
(**Attention** : ceci ne veut nullement dire que les formules A et $\forall x\ r(x, f(x))$ sont logiquement équivalentes!)

- $\Rightarrow$ Supposons que A soit satisfiable, c'est-à-dire qu'il existe un modèle $\mathcal{I}_1$ de A. Soit D le domaine de $\mathcal{I}_1$. Soit ch une choix *quelconque* de valeurs pour les variables. Par définition de modèle,

$$[\forall x \exists y r(x, y)]_{\mathcal{I}_1, ch} = V \tag{1}$$

Toujours en utilisant des définitions, on obtient que pour tout élément $d \in D$ il existe un élément $a \in D$ tel que

$$[r(x, y)]_{\mathcal{I}_1, ch[x:=d][y:=a]} = V \tag{2}$$

Observons que l'élément a DEPEND de d. On va donc plutôt le noter a_d et on va réécrire l'énoncé 2 ainsi :

$$[r(x, y)]_{\mathcal{I}_1, ch[x:=d][y:=a_d]} = V \tag{3}$$

Ce dernier énoncé nous dit que, pour tout élément $d \in D$, il existe un élément $a_d \in D$ tel que

$$\langle [x]_{\mathcal{I}_1, ch[x:=d][y:=a_d]},\ [y]_{\mathcal{I}_1, ch[x:=d][y:=a_d]} \rangle \in \mathcal{I}_1(r) \tag{4}$$

c'est à dire que pour tout élément $d \in D$, il existe un élément $a_d \in D$ tel que

$$\langle d, a_d \rangle \in \mathcal{I}_1(r) \tag{5}$$

Soit $\mathcal{I}_2$ une interprétation obtenue à partir de $\mathcal{I}_1$ en y ajoutant l'interprétation suivante du symbole f : la fonction $F : D \Rightarrow D$ associée à f est telle que, pour tout $d \in D$

$$F(d) = a_d$$

Ainsi, $[f(x)]_{\mathcal{I}_2, ch[x:=d]} = a_d$. Par l'énoncé 5, on obtient que pour tout élément $d \in D$,

$$\langle d, [f(x)]_{\mathcal{I}_2, ch[x:=d]} \rangle \in \mathcal{I}_2(r) \tag{6}$$

Il s'ensuit que pour tout élément $d \in D$,

$$\langle [x]_{\mathcal{I}_1, ch[x:=d]},\ [f(x)]_{\mathcal{I}_2, ch[x:=d]} \rangle \in \mathcal{I}_2(r) \tag{7}$$

Ceci implique que, pour tout $d \in D$,

$$[r(x, f(x))]_{\mathcal{I}_2, ch[x:=d]} = V \tag{8}$$

Donc

$$[\forall x r(x, f(x))]_{\mathcal{I}_2, ch} = V \tag{9}$$

Puisque ch est un choix de valeurs quelconque, il s'ensuit que $\mathcal{I}_2$ est un modèle de $\forall x\ r(x, f(x))$

- $\Leftarrow$ Supposons que $\forall x\ r(x, f(x))$ ait un modèle $\mathcal{I}$. Il est facile de voir que la formule $\forall x \exists y r(x, y)$ est une conséquence logique de $\forall x r(x, f(x))$ (tout en ne lui étant pas équivalente). Donc $\mathcal{I}$ est aussi un modèle de $\forall x \exists y\ r(x, y)$.

Théorème 60 (Existence d'une forme de Skolem) Etant donnée n'importe quelle formule A en forme prénexe, il existe une formule A_{sk}, dite *forme de Skolem* de A telle que :

1. Si A contient n quantificateurs existentiels, n nouveaux symboles de fonction apparaissent dans A_{sk}. (Il s'agit des fonctions de Skolem.)
2. A_{sk} ne contient pas de quantificateurs existentiels
3. A est satisfiable si et seulement si A_{sk} est satisfiable.

Preuve du théorème 60
La preuve utilise de façon essentielle le lemme 59 et se fait par récurrence sur le nombre n de quantificateurs existentiels de A.
Base. $n = 0$. Dans ce cas, on peut prendre $A_{sk} = A$
Etape de récurrence. $n > 0$. Dans ce cas, A a la forme

$$\forall x_1, \cdots, \forall x_i \exists x_{i+1} Q_{i+2} x_{i+2}, \cdots, Q_{i+n} x_{i+n} M$$

où $i \geq 0$ et où $n \geq 0$. En appliquant le lemme 59 à A, on obtient l'existence d'une formule A' telle que :

- A' contient $n - 1$ quantificateurs existentiels, puisque $\exists x_{i+1}$ a été éliminé
- A est satisfiable si et seulement A' est satisfiable.

Puisque A' contient $n-1$ quantificateurs existentiels, par hypothèse de récurrence, il existe une formule A" qui est satisfiable si et seulement si A' est satisfiable et qui a les propriétés 1-3 de l'énoncé du théorème. Par transitivité, A est satisfiable si et seulement si A'' l'est. Donc on peut prendre $A_{sk} = A''$.

Méthode de Skolemisation

A partir de la preuve du théorème 60, on peut dégager une méthode pour mettre n'importe quelle formule prénexe A en forme de Skolem, ou comme on dit, pour "*Skolemiser*" A. Illustrons cette méthode sur un exemple.

Exemple 61 Soit
$$A = \exists x \forall y \forall z \exists v (r(x, g(y, v)) \wedge p(a, y, z)$$
où a est une constante et g un symbole de fonction à 2 arguments. On a :
A est satisfiable
ssi
$\forall y \forall z \exists v (r(c, g(y, v)) \wedge p(a, y, z)$ est satisfiable (c est une nouvelle constante)
ssi
$r(c, g(y, f(y, z))) \wedge p(a, y, z)$ est satisfiable (f est un nouveau symbole de fonction).
La formule $r(c, g(f(y, z))) \wedge p(a, y, z)$ est la forme de Skolem de A.

On arrive enfin au but du jeu : dégager une méthode permettant de transformer un ensemble E de formules du calcul des prédicats en un ensemble de clauses C_E. Ci-dessous, quand on parle de *forme normale conjonctive* d'une formule A **sans quantificateurs** on entend, évidemment, la "même" chose que dans le cas propositionnel, la seule différence étant dans la syntaxe des littéraux.

Théorème 62 (Transformation en Forme Clausale) Soit E un ensemble de formules. Il existe un ensemble de clauses C_E tel que E est satisfiable ssi C_E est satisfiable.

Preuve du théorème 62.
Pour construire C_E à partir de E, il suffit d'appliquer l'algorithme suivant :

1. Eliminer les connecteurs $\rightarrow, \leftrightarrow$
2. Pour toute formule A de E, effectuer les opérations suivantes :
   (a) Réécrire A en une formule A_1 rectifiée équivalente
   (b) Réécrire A_1 en une formule A_2 en forme prénexe équivalente.
   (c) Skolemiser A_2, en obtenant ainsi une formule A_3 en forme de Skolem telle que A_3 est satisfiable si et seulement si A l'est.
   (d) Puisque A_3 ne contient, de toute façon, que des quantificateurs universels, les supprimer, en obtenant ainsi une formule équivalente A_4 qui ne contient pas de quantification.
   (e) Mettre A_4 en forme normale conjonctive et puis clausale (au sens propositionnel). On obtient ainsi un ensemble de clauses C_A.
3. Construire l'ensemble $C_E = \bigcup_{A \in E} C_A$.

3.3.2 Théorie de l'Unification

BUT. On veut étendre la règle de résolution propositionnelle de façon à pouvoir gérer, par exemple, l'inférence suivante :

$$\frac{r(a, f(x)) \vee q(b) \qquad \neg r(y, f(b)) \vee s(a, b)}{q(b) \vee s(a, b)} RES$$

Ici, $r(a, f(x))$ et $r(y, f(b))$ ne sont pas exactement le même atome, car a et y sont des termes distincts; De même pour $f(x)$ et $f(b)$. Toutefois, on peut "rendre égaux" ce termes en substituant la constante a à la variable y et la constante b à la variable x. Les techniques permettant de"rendre égaux" des termes au sens suggéré sont dites techniques d'*unification.*

Définition 63 (Substitution) Une *substitution* est un ensemble fini de couples formés d'une variable (individuelle) x_i et d'un terme t_i différent de x_i. Cet ensemble est noté $\{x_1 \rightarrow t_1, \cdots x_n \rightarrow t_n\}$; on impose aussi que, pour tout $1 \leq i, j \leq n$, x_i et x_j soient deux variables distinctes. On appelle l'ensemble de variables $\{x_1, \cdots, x_n\}$ *domaine* de la substitution. On utilisera les lettres minuscules grecques $\alpha, \beta, \gamma, \cdots \sigma, \tau, ...$ pour indiquer des substitutions.

Notation. Dans la suite, si σ est une substitution, t est un terme et A une formule, on notera :

- $(t)_\sigma$ le terme résultat de l'application de σ à toute occurrence de variable de t;

- $(A)_\sigma$ la formule résultat de l'application de σ à toute occurrence *libre* de variable de A.

EXEMPLES

Etant données deux substitutions σ et γ, on peut les *composer*, au même sens que l'on peut composer deux fonctions mathématiques.

Définition 64 (Composition de substitutions) Soient $\theta = \{x_1 \to s_1, ..., x_m \to s_m\}$ et $\sigma = \{y_1 \to t_1, ..., y_n \to t_n\}$ deux substitutions. La composition de θ avec γ est la substitution obtenue á partir de l'ensemble $\{x_1 \to (s_1)_\sigma, ..., x_m \to (s_m)_\sigma, y_1 \to t_1, ..., y_n \to t_n\}$ en supprimant :

- tout élément de la forme $x \to x$,
- tout element $y_j \to t_j$ tel que v_j appartient au domaine de θ.

On note la composition de σ et γ par $\sigma\gamma$; remarquer que la notation $\sigma\gamma$ se lit comme : "faire *d'abord σ puis γ*".
EXEMPLES DE COMPOSITION.

Définition 65 (Problème d'Unification, Unificateur)
- On appelle *problème d'unification* tout ensemble d'équations de la forme $P = \{t_1 = s_1, \cdots, t_n = s_n\}$ où tout t_i et tout s_i sont des termes. Une substitution σ est un *unificateur* de P si, pour tout $i, 1 \leq i \leq n$, les termes $(t_i)_\sigma$ et $(s_i)_\sigma$ sont (syntactiquement) identiques. On note par $U(P)$ l'ensemble d'unificateurs de P.
- On dit que P est *unifiable* si $U(P) \neq \emptyset$.
- On dit que deux termes t et t' sont *unifiables* par la substitution σ si σ est un unificateur du problème d'unification $\{t = t'\}$. De même, on dit que deux atomes $p(t_1, \cdots, t_n)$ et $p(s_1, \cdots, s_n)$ sont unifiables par la substitution σ si σ est un unificateur du problème d'unification $\{t_1 = s_1, \cdots, t_n = s_n\}$.

En général, étant donné un problème d'unification P, *plusieurs unificateurs différents de P peuvent exister.* Il est aussi possible qu'aucun unificateur n'existe.

EXEMPLES DE L'EXISTENCE DE PLUSIEURS UNIFICATEURS POSSIBLES POUR UN PROBLEME D'UNIFICATION DONNE.

Même si, étant donné un problème d'unification P, ils peuvent exister plusieurs unificateurs differents de P, la notion de composition nous permet de définir une notion d'unificateur "principal" (ou "le plus général possible") d'un problème d'unification.

Définition 66 (Unificateur Principal) Soit P un problème d'unification et soit σ un unificateur de P. On dit que σ est un unificateur *principal* (ou un unificateur *le plus général*) de P si, pour tout unificateur σ' de P, il existe une substitution γ telle que $\sigma' = \sigma\gamma$.

EXEMPLES D'UNIFICATEURS PRINCIPAUX ET PAS PRINCIPAUX.

On adopte les conventions suivantes :

1. On identifie deux unificateurs et σ et σ' qui ne différent que par des renommage de variables; par exemple, si P est $\{f(x,a) = f(y,a)\}$, $\sigma = \{x \to y\}$ et $\sigma' = \{y \to x\}$, on considère que σ et σ' sont le *même* unificateur de P (modulo renommage).
2. On impose que si P est un problème d'unification et σ un unificateur *principal* de P, alors, si le couple $x \to t$ est un élément de σ, forcément x apparaît dans une équation de P.

Grâce à ces conventions, si P a un unificateur, alors il existe *un seul* unificateur principal de P. On peut alors adopter la notation suivante, sans risque d'ambiguïté.

Notation. Soit P un problème d'unification tel que $U(P) \neq \emptyset$. L'unificateur principal de P est noté $mgu(P)$. Si t et s sont des termes unifiables (par une substitution quelconque), on aura le droit d'écrire $mgu(t,s)$ à la place de $mgu(\{s = t\})$.

Il existe des règles mécaniques permettant de décider, pour un problème d'unification donné P, si $U(P) = \emptyset$ ou pas, et, dans le cas où un unificateur de P existe, de calculer $mgu(P)$. Mais, ici, à la place de presenter ces règles, on indiquera un algorithme **très simple** permettant directement de décider si deux atomes $p(t_1, \cdots, t_n)$ et $p(s_1, \cdots, s_n)$ sont unifiables ou pas. D'une part, il est clair que cet algorithme permet aussi de resoudre le problème d'unification $\{t_1 = s_1, \cdots, t_n = s_n\}$, de l'autre, cet algorithme, tout en étant très simple, permet quand même de bien définir l'extension de la résolution au cas du calcul des prédicats.

Ci-dessous, la notation ϵ indique la substitution vide.

Algorithme d'unification de deux atomes

Entrée : deux atomes A_1 et A_2
Sortie : le mgu σ de A_1 et A_2 si A_1 et

1. $\sigma := \epsilon$
2. Tant que $A_{1\sigma} \neq A_{2\sigma}$ faire
   (a) Déterminer le symbole le plus à gauche de $A_{1\sigma}$ qui soit différent du symbole de même rang de $A_{2\sigma}$.
   (b) Déterminer t_1, t_2, les sous-termes de $A_{1\sigma}$ et $A_{2\sigma}$ qui commencent à ce symbole.
   (c) Si "aucun n'est une variable" ou bien "l'un est une variable qui apparaît dans l'autre"
      - alors imprimer "A_1 et A_2 ne sont pas unifiables"; arrêt.
      - sinon faire
         i. déterminer x, une variable qui est ou bien t_1 ou bien t_2, et t, celui de t_1 et t_2 qui n'est pas x

ii. $\sigma := \sigma\{x \rightarrow t\}$

- fin-de-si

(d) fin-de-tant-que

3. Imprimer "σ est le mgu de A_1 et A_2"

4. Arrêt

EXEMPLES

3.3.3 Preuves par Résolution en Calcul des Prédicats

Dans cette section, on adopte les *conventions* suivantes :

1. Etant donnée une clause $\forall x_1, \cdots, \forall x_n(L_1 \vee \ldots L_m)$, on suppose que les seules variables libres de $L_1 \vee \ldots L_m$ sont $x_1, \cdots, x_n$. Ceci sans perte de généralité, car une clause C contenant une variable libre x est satisfiable (a un modèle) si et seulement si $\forall x C$ est satisfiable.

2. Souvent (mais pas toujours), on écrira $L_1 \vee \ldots L_m$ à la place de $\forall x_1, \cdots, \forall x_n(L_1 \vee \ldots \vee L_m)$.

3. Comme dans le cas de la résolution propositionnelle, on considère les clauses modulo l'ordre des littéraux; ainsi, $p(x) \vee q(y) \vee r(a)$ est identifiée à $q(y) \vee r(a) \vee p(x)$, par exemple.

Les règles du système de preuve par résolution sont les suivantes. On a la règle

$$\frac{r(s_1, \cdots, s_n) \vee C_1 \qquad\qquad \neg r(t_1, \cdots, t_n) \vee C_2}{(C_1 \vee C_2)_\sigma} \quad (RES)$$

où

- C_1 et C_2 sont des clauses n'ayant aucune variable commune

- σ est le mgu du problème d'unification $\{s_1 = t_1, \cdots, s_n = t_n\}$.

et la règle

$$\frac{L \vee L' \vee C}{(L \vee C)_\sigma} \quad (FAC)$$

où

- C est une clause

- Ou bien L a la forme $r(s_1, \cdots, s_n)$ et L' a la forme $r(t_1, \cdots, t_n)$, ou bien L a la forme $\neg r(s_1, \cdots, s_n)$ et L' a la forme $\neg r(t_1, \cdots, t_n)$

- σ est le mgu du problème d'unification $\{s_1 = t_1, \cdots, s_n = t_n\}$.

Soit E un ensemble de clauses et C une clause. Les notions de *déduction* par résolution de C à partir de E et de *réfutation* de E sont les mêmes que pour le cas propositionnel.
EXEMPLES DE REFUTATIONS.

Comme dans le cas propositionnel, un ensemble de clauses E est insatisfiable si et seulement s'il existe une réfutation de E; en d'autre termes, le système de preuve par résolution est (réfutationellement) correct et complet. Ainsi, pour prouver qu'une formule A est conséquence logique d'un ensemble fini $S = \{F_1, \cdots F_n\}$ de formules, on peut procéder de la façon suivante :

1. Puisque $F_1, \cdots F_n \models A$ si et seulement si l'ensemble de formules $\{F_1, \cdots F_n, \neg A\}$ est insatisfiable, on construit un ensemble de clauses $C_{S,\neg A}$ et on réduit le problème à la preuve de l'insatisfiabilité de $C_{S,\neg A}$.
2. On construit une réfutation de $C_{S,\neg A}$.

Correction et Complétude du Système de Résolution

Pour prouver la correction, on a besoin du lemme suivant.

Lemme 67 Si une clause C_3 est obtenue en appliquant la règle RES à deux clauses C_1 et C_2, alors C_3 est une conséquence logique de l'ensemble de clauses $\{C_1, C_2\}$. De même, si une clause C_2 est obtenue en appliquant la règle de factorisation à une clause C_1, alors C_2 est une conséquence logique de C_1.

La preuve de ce lemme est laissée en exercice. En faisant une récurrence sur la longueur de la déduction, on obtient alors le corollaire suivant :

Corollaire 68 S'il existe une déduction d par résolution d'une clause C à partir d'un ensemble de clauses E, alors C est une conséquence logique de E.

En considérant le cas particulier ou C est la clause vide, on obtient enfin le théorème suivant.

Théorème 69 (Correction (réfutationnelle) du système de résolution) Soit E un ensemble de clauses. S'il existe une réfutation de E, alors E est insatisfiable.

La preuve de complétude est bien plus délicate. Tout d'abord, on a besoin d'un certain nombre de résultats et notions préliminaires.

Tout d'abord, un lemme purement technique dont on aura besoin par la suite, et, notamment, dans la preuve du théorème d'Herbrand.

Lemme 70 Soit A une formule dont les variables libres sont $x_1, \cdots, x_n$. Soit $\mathcal{I}$ une interprétation, soit ch un choix de valeurs pour les variables relatif à $\mathcal{I}$ et soit $\sigma = \{x_1 \to t_1, \cdots x_n \to t_n\}$ une substitution. Soient $d_1, \cdots, d_n$ des éléments du domaine tels que, pour tout $i, 1 \leq i \leq n$, $[t_i]_{\mathcal{I},ch} = d_i$. On a la propriété suivante :

$$[A]_{\mathcal{I},ch[x_1:=d_1]\cdots[x_n:=d_n]} = [(A)_\sigma]_{\mathcal{I},ch}$$

La preuve de ce lemme se fait par récurrence sur A. Voyons un exemple de la propriété énoncée par le lemme. Soit A la formule $r(x_1, x_2)$. Soit $\mathcal{I}$ une interprétation telle que : le domaine est N, $\mathcal{I}(a)$ est le nombre 0, $\mathcal{I}(f)$ est la fonction "successeur" et $\mathcal{I}(r)$ est la relation $<$. Soit un choix de valeur ch tel que $ch(x_1) = 0$. Soit t_1 le terme a et t_2 le terme $s(a)$. Ici, les éléments d_1 et d_2 du domaine sont, respectivement, 0 et 1, car $[a]_{\mathcal{I},ch} = 0$ et $[s(a)]_{\mathcal{I},ch} = 1$. Soit σ la substitution $\{x_1 \rightarrow a, x_2 \rightarrow s(a)\}$. Vérifions que la propriété du lemme est vraie. Ici, $[A]_{\mathcal{I},ch[x_1:=0][x_2:=1]}$ est $[r(x_1, x_2)]_{\mathcal{I}ch[x_1:=0][x_2:=1]}$, qui s'évalue à Vrai, puisque $0 < 1$. L'expression $[(A)_\sigma]_{\mathcal{I},ch}$ est $[((r(x_1, x_2))_{\{x_1 \rightarrow a, x_2 \rightarrow s(a)\}}]_{\mathcal{I},ch}$. Si on effectue la substitution indiquée, on trouve $[r(a, s(a)]_{\mathcal{I},ch}$ qui, à nouveau, s'évalue à Vrai.

Pour étudier la satisfiabilité d'un ensemble de clauses, on n'est pas obligé de considérer toutes les interprétations possibles : un théorème important dû au logicien français Jacques Herbrand, permet de se limiter à des interprétations d'une certaine forme précise. Ceci motive les définitions suivantes.

Définition 71 (Univers et Base de Herbrand) Soit $\mathcal{L}$ un langage du calcul des prédicats contenant au moins une constante.

- L'ensemble $\mathcal{H}$ des termes *clos* (i.e. sans variables) de $\mathcal{L}$ est dit *univers de Herbrand* (relatif à $\mathcal{L}$).

- L'ensemble $\mathcal{B}$ des atomes *clos* de $\mathcal{L}$ est dit *base de Herbrand* (relatif à $\mathcal{L}$).

EXEMPLES

Définition 72 (Interprétation de Herbrand) Soit $\mathcal{L}$ un langage du calcul des prédicats (auquel on ajoute au moins une constante a s'il ne contient pas de constantes). Une *interprétation de Herbrand* $\mathcal{I}_H$ pour $\mathcal{L}$ est une interprétation telle que

1. Le domaine de $\mathcal{I}_H$ est l'univers de Herbrand relatif à $\mathcal{L}$

2. Tout symbole de fonction f à n arguments ($n \geq 0$) est interprété par "lui-même", c'est-à-dire que $\mathcal{I}_H(f)$ est la fonction qui associe aux termes clos $t_1, \cdots t_n$ le terme clos $f(t_1, \cdots t_n)$. En particulier, $\mathcal{I}_H(c)$ est c, pour toute constante c.

Un modèle de Herbrand d'un ensemble de formules E (dont le langage est $\mathcal{L}$) est une interprétation de Herbrand qui est modèle de E.

On peut remarquer deux choses :

1. Dans la définition d'interprétation de Herbrand, aucune contrainte n'a été posée quant à la façon d'interpréter un symbole de relation r. Donc, comme d'habitude, si r a n arguments, $\mathcal{I}_H(r)$ est un ensemble de n-uplets d'éléments du domaine. Mais ici le domaine est l'univers de Herbrand, donc l'ensemble des termes clos. Ainsi, on peut "identifier" $\mathcal{I}_H(r)$ avec l'ensemble des atomes clos de forme $r(t_1, \cdots, t_n)$ que l'interprétation déclare "vrais". Ainsi, $\mathcal{I}_H(r)$ est souvent défini directement comme un sous-ensemble de l'ensemble d'atomes clos suivant : $\{r(t_1, \cdots, t_n) \mid t_1, \cdots, t_n \in \mathit{Univers\ deHerbrand}\}$

2. Puisque le domaine et la façon d'interpréter les symboles de fonction sont fixés, la seule manière qu'ont deux interprétations de Herbrand $\mathcal{I}_H$ et $\mathcal{I}'_H$ de différer réside dans l'interprétation des symboles de relation. Ainsi, souvent on identifie une interprétation de Herbrand avec un sous-ensemble de la base de Herbrand; intuitivement, ce sous-ensemble représente ces atomes clos que l'interprétation déclare vrais.

EXEMPLES

Ci-dessous, on donne une reformulation du célèbre théorème d'Herbrand qui est traditionnellement utilisée pour prouver la complétude du système de résolution. En réalité, la version du théorème démontrée originellement par Herbrand est assez différente et se place dans le contexte de la *prouvabilité* (notion syntaxique) plutôt que dans celui de la satisfiabilité (notion sémantique).

Théorème 73 (Herbrand) Un ensemble de clauses E est satisfiable si et seulement s'il a un modèle de Herbrand.

Preuve. L'implication "Si E a un modèle d'Herbrand alors E est satisfiable" étant triviale, seulement l'implication réciproque est à prouver. Soit donc $\mathcal{I}$ un modèle quelconque de E. On veut démontrer l'existence d'un modèle de Herbrand de E. Soit $\mathcal{I}_H$ l'interprétation de Herbrand (du langage $\mathcal{L}$ de E) telle que, pour tout symbole de relation r d'arité n, $\mathcal{I}_H(r)$ est l'ensemble de n-uplets de termes clos suivant :

$$\{\langle t_1, \cdots, t_n\rangle \mid \mathcal{I} \models r(t_1, \cdots, t_n)\}$$

Soit $\forall x_1 \cdots \forall x_n(\neg A_1 \vee \cdots \vee \neg A_k \vee B_1 \vee \cdots \vee B_p)$ une clause quelconque de E (les A_i et les B_i sont des atomes). Ce que l'on veut montrer, c'est que cette clause est vraie par rapport à $\mathcal{I}_H$.

Or, puisque $\mathcal{I} \models E$, on a, pour *tout* choix de valeurs ch et *toute* suite d'éléments $a_1, \cdots, a_n$ du domaine de $\mathcal{I}$:

$$[\neg A_1 \vee \cdots \vee \neg A_k \vee B_1 \vee \cdots \vee B_p]_{\mathcal{I},ch[x_1:=a_1]\cdots[x_n:=a_n]} = V \tag{10}$$

Soit $t_1, \cdots, t_n$ une suite *quelconque* de termes *clos*. Soient $d_1, \cdots, d_n$ les éléments du domaine de $\mathcal{I}$ tels que, pour tout $i, 1 \leq i \leq n$, $d_i = [t_i]_{\mathcal{I},ch}$.[2] L'énoncé 10 implique :

$$[\neg A_1 \vee \cdots \vee \neg A_k \vee B_1 \vee \cdots \vee B_p]_{\mathcal{I},ch[x_1:=d_1]\cdots[x_n:=d_n]} = V \tag{11}$$

Soit σ la substitution $\{x_1 \to t_1, \cdots x_n \to t_n\}$. Grâce au lemme 70, 11 implique :

$$[(\neg A_1 \vee \cdots \vee \neg A_k \vee B_1 \vee \cdots \vee B_p)_\sigma]_{\mathcal{I},ch} = V \tag{12}$$

Ceci veut dire que, pour tout ch, ou bien $\exists i, 1 \leq i \leq k$ tel que $[\neg(A_i)_\sigma]_{\mathcal{I},ch} = V$, ou bien $\exists j, 1 \leq j \leq p$ tel que $[(B_i)_\sigma]_{\mathcal{I},ch} = V$. Or, puisque tout t_i est clos, tout atome $(A_i)_\sigma$ et tout atome $(B_j)_\sigma$ sont clos. Il en suit que tout choix de valeur ch' aura le même effet que ch sur l'évaluation de $(A_i)_\sigma$ et $(B_j)_\sigma$ par rapport à $\mathcal{I}$.

Donc, ou bien il existe un $i, 1 \leq i \leq k$ tel que

$$\mathcal{I} \models \neg(A_i)_\sigma \tag{13}$$

[2]Remarquez que, puisque t_i est clos, en réalité le choix de valeurs ch n'a pas d'influence sur le résultat de l'évaluation de t_i, qui est toujours le même.

ou bien il existe un $j, 1 \leq j \leq k$ tel que

$$\mathcal{I} \models (B_j)_\sigma \tag{14}$$

Alors, par construction de l'interprétation de Herbrand $\mathcal{I}_H$, on a : ou bien il existe un $i, 1 \leq i \leq k$ tel que

$$\mathcal{I}_H \models \neg(A_i)_\sigma \tag{15}$$

ou bien il existe un $j, 1 \leq j \leq k$ tel que

$$\mathcal{I}_H \models (B_j)_\sigma \tag{16}$$

Par conséquent,

$$\mathcal{I}_H \models (\neg A_1 \vee \cdots \vee \neg A_k \vee B_1 \vee \cdots \vee B_p)_\sigma \tag{17}$$

Ceci veut dire que, pour tout choix de valeurs ch_H relatif à $\mathcal{I}_H$, on a :

$$[(\neg A_1 \vee \cdots \vee \neg A_k \vee B_1 \vee \cdots \vee B_p)_\sigma]_{\mathcal{I}_H, ch_H} = V \tag{18}$$

En utilisant à nouveau le lemme 70 (dans l'autre direction) on obtient :

$$[\neg A_1 \vee \cdots \vee \neg A_k \vee B_1 \vee \cdots \vee B_p]_{\mathcal{I}_H, ch_H[x_1:=t_1]\cdots[x_n:=t_n]} = V \tag{19}$$

car, dans toute interprétation de Herbrand, un terme clos t s'évalue en lui-même. Or, puisque $\langle t_1, \cdots, t_n \rangle$ est un n-uplet *quelconque* de termes clos et puisque le domaine de $\mathcal{I}_H$ est justement l'ensemble des termes clos, l'énoncé 19 permet de conclure que la clause $\forall x_1 \cdots \forall x_n(\neg A_1 \vee \cdots \vee \neg A_k \vee B_1 \vee \cdots \vee B_p)$ est vraie dans l'interprétation d'Herbrand $\mathcal{I}_\mathcal{H}$. ■

L'ensemble de toutes les interprétations d'Herbrand d'un ensemble de clauses E peut être représenté par un arbre binaire, dit *arbre sémantique complet.* Avant de donner la définition de cet arbre, observons que, puisque les ensemble des symboles de fonctions et des symboles de relations d'un langage $\mathcal{L}$ du calcul des prédicats sont dénombrables, il est possible d'énumérer les atomes clos, c'est-à-dire les éléments de la base de Herbrand $\mathcal{B}$.

Définition 74 (Arbre sémantique complet) Soit $B_0, B_1, B_2....$ une énumération des atomes clos. L'arbre sémantique complet associé à cette énumération est un arbre binaire équilibré dont les nœuds internes sont étiquetés par des atomes clos et les arcs par V ou F comme suit

- La racine est étiquetée par l'atome B_0
- Les successeurs d'un nœud étiqueté par B_i sont étiquetés par B_{i+1}; l'arc gauche issu du nœud étiqueté par B_i est étiqueté par V et l'arc droit par F.

Un arbre sémantique complet a donc la structure indiquée par la Figure 1 (à la fin du chapitre). Chaque branche représente une interprétation d'Herbrand distincte et l'arbre représente *toutes* les interprétations de Herbrand possibles pour le langage consideré.

Exemple 1. Soit $\mathcal{L}$ un langage dont les seuls symboles de fonctions sont les constantes a et b et les seuls symboles de relations les symboles unaires r et q. Ici, l'univers de Herbrand $\mathcal{H}$ est l'ensemble fini $\{a, b\}$ et la base de Herbrand $\mathcal{B}$ est aussi finie. Soit l'énumération suivante de $\mathcal{B}$: B_0 est $q(a)$, B_1 est $q(b)$, B_2 est $r(a)$, B_3 est $r(b)$. L'arbre sémantique complet associé est indiqué dans la Figure 2.

Considerons l'ensemble des clauses $E = \{\neg r(x) \vee q(x), q(a), r(a)\}$. En lisant l'arbre de l'exemple précédant de gauche à droite, la branche la plus à gauche, par exemple, correspond à un modèle de E, mais ce n'est pas le cas (entre autre) de la branche la plus à droite, car, dans cette interprétation, la formule $q(a)$ est fausse. En effet, déjà le tout premier arc de cette dernière branche permet de falsifier l'ensemble des clauses E, car, à ce niveau, $q(a)$ est déjà fausse. Il est donc clair que toute interprétation qui "prolonge" cet arc rend E faux. (En d'autres termes, toute interprétation de Herbrand qui déclare la formule $q(a)$ fausse falsifie E, peu importe comment les autres atomes sont évalués). Donc, si on veut utiliser l'arbre sémantique pour identifier les modèles de E, on peut "bloquer" la construction de la branche la plus à droite après le deuxième nœud. Ceci motive les définitions suivantes.

Définition 75 (Instance) Soit $C = L_1 \vee \cdots \vee L_m$ une clause. Une *instance* de C est une formule $(L_1 \vee \cdots \vee L_m)_\sigma$, où σ est une substitution. Si la substitution σ remplace les variables par des termes *clos*, alors $(L_1 \vee \cdots \vee L_m)_\sigma$ est dite *instance close* de C.

Définition 76 (Nœud d'échec d'un arbre sémantique complet) Soit $\mathcal{L}$ un langage du calcul des prédicats. Soit $B_0, B_1, B_2....$ une énumération des atomes clos de $\mathcal{L}$. Soit E un ensemble de clauses écrites dans $\mathcal{L}$. Soit A un arbre sémantique complet associé à $B_0, B_1, B_2.....$
Un nœud n de A est dit *nœud d'échec* par rapport à E si le segment β_n de la branche qui va de la racine de l'arbre au nœud n suffit à falsifier au moins une instance close d'une clause de E et si aucun prédécesseur de n n'a cette propriété.

Remarque 77 Observez que si la racine d'un arbre sémantique de E est déjà un nœud d'échec, la clause vide appartient à E, car aucune autre clause ne peut être falsifiée déjà à ce niveau.

Définition 78 (Arbre sémantique d'un ensemble de clauses E) Soit $\mathcal{L}$ un langage du calcul des prédicats. Soit $B_0, B_1, B_2....$ une énumération des atomes clos de $\mathcal{L}$. Soit E un ensemble de clauses écrites dans $\mathcal{L}$. Soit A un arbre sémantique associé à $B_0, B_1, B_2.....$ On appelle *arbre sémantique de l'ensemble de clauses E* l'arbre obtenu à partir de A en éliminant les sous-arbres issus des nœuds d'échec et en étiquetant ces derniers par les instances closes des clauses respectivement falsifiées.

Exemple 2 L'arbre sémantique de l'ensemble de clauses $\{\neg r(x) \vee q(x), q(a), r(a)\}$ construit à partir de l'arbre sémantique complet de la figure 2 est indiqué dans la figure 3.
Exemple 3. Soit $\mathcal{L}$ un langage du calcul des prédicats dont les symboles de fonctions sont la constante a et le symbole unaire s et les symboles de relations les symboles unaires r et q. L'univers de Herbrand est l'ensemble *infini* $\{a, s(a), s(s(a)), ...\}$. Soit l'énumération suivante de la base d'Herbrand : $r(a), q(a), r(s(a)), q(s(a)), r(s(s(a))), ...$ L'arbre sémantique complet A associé est infini. Soit E l'ensemble de clauses $\{\neg r(x) \vee q(s(x)), r(a), \neg q(s(a))\}$. L'arbre sémantique A' de E est *fini*, car toute branche de A contient un nœud d'échec. Cet arbre est indiqué dans la Figure 4.

Définition 79 (Arbre sémantique clos d'un ensemble de clauses E) Soit A' un arbre sémantique d'un ensemble de clauses E obtenu à partir d'un arbre sémantique complet A. L'arbre A' est dit *clos* s'il est fini et si toute feuille est un nœud d'échec de A.

Remarque 80 Etant donné un ensemble de clauses E, il se peut qu'un arbre sémantique de E soit fini sans être clos : c'est le cas où l'univers de Herbrand est fini et E est satisfiable.

On a le résultat suivant :

Théorème 81 Un ensemble de clauses E n'a aucun modèle d'Herbrand si et seulement s'il existe un arbre sémantique de E qui est clos.

Preuve.

- Supposons que E n'ait aucun modèle de Herbrand. Alors toute interprétation d'Herbrand I_H rend fausse au moins une instance close d'une clause de E. Soit une énumération quelconque de la base de Herbrand et soit A l'arbre sémantique complet correspondant. Forcement, toute branche β de A rencontre tôt ou tard un nœud d'échec. On a donc un arbre sémantique de E qui est clos.

- Soit A un arbre sémantique clos de E. Toute branche β de A est alors finie et suffit à falsifier au moins une instance close C' d'une clause de E : l'étiquette de la feuille de β. Puisque toute interprétation de Herbrand peut être vue comme le "prolongement" d'une branche de A, il s'ensuit que toute interprétation de Herbrand rend fausse au moins une clause de E. Donc il n'existe pas de modèle de Herbrand de E.

■

Si on combine le résultat 81 avec le théorème 73, on obtient une deuxième version du théorème de Herbrand, qui nous sera très utile pour prouver la complétude du système de preuve par résolution.

Théorème 82 (Herbrand, version 2) Un ensemble de clauses E est insatisfiable si et seulement s'il existe un arbre sémantique clos de E.

Preuve Par le théorème 73, E est insatisfiable si et seulement si E n'a aucun modèle de Herbrand, et par le théorème 81, E n'a aucun modèle de Herbrand si et seulement si il existe un arbre sémantique clos de E. ■

Avant de pouvoir prouver la complétude du système de preuve par résolution, on a besoin d'un dernier lemme, que l'on ne prouvera pas ici.

Lemme 83 (Relèvement) Soient C'_1 et C'_2 deux instances de, respectivement, la clause C_1 et la clause C_2. Soit C'_{res} une clause obtenue à partir de C'_1 et C'_2 en appliquant la règle de résolution une fois (et éventuellement plusieurs fois la règle de factorisation). Il existe alors une clause C_{res} telle que C_{res} est obtenue à partir de C_1 et C_2 en appliquant une fois la règle de résolution (et éventuellement plusieurs fois la règle de factorisation) et C'_{res} est une instance de C_{res}.

Par exemple, soient $C_1 = p(x) \vee r(x,y)$ et $C_2 = \neg p(v) \vee q(v)$. Soient C'_1 et C'_2 les deux instances (closes) $p(a) \vee r(a,b)$ et $\neg p(a) \vee q(a)$. Ici, C'_{res} est $r(a,b) \vee q(a)$. Soit la clause $r(x,y) \vee q(x)$. Cette clause est bien obtenue par résolution à partir de C_1 et C_2 et elle a comme instance C'_{res}. (Il suffit de prendre la substitution $\sigma = \{x \rightarrow a, y \rightarrow b\}$).

Théorème 84 (Complétude (réfutationelle) du système de résolution) Soit E un ensemble de clauses. Si E est insatisfiable, alors il existe une réfutation de E.

Preuve. Supposons que E soit insatisfiable. Par le théorème 82 il existe alors un arbre sémantique clos A de E. On prouve l'existence d'une réfutation r de E par récurrence sur la hauteur h de A. Dans la suite, on dit qu'un littéral L est un *disjoint* d'une clause C si C a la forme $L \vee C'$ (modulo l'ordre des littéraux).
Base La hauteur de A est 1. Dans ce cas, A se reduit à un seul nœud : la racine. Il s'ensuit que la clause vide appartient à E, car aucune autre clause ne peut être falsifiée directement à la racine. Dans ce cas, r est la réfutation triviale qui consiste de la seule clause vide.
Etape de récurrence. La hauteur h de A est supérieure à 1. Alors A possède au moins un nœud interne n dont les deux successeurs gauche et droit, disons n_g et n_d, sont deux feuilles. Soit $A(t_1, \cdots, t_n)$ l'atome clos qui étiquette les deux arcs qui mènent de n à n_g et n_d. Soit C'_1 l'instance close de la clause C_1 de E falsifiée par n_g et soit C'_2 l'instance close de la clause C_2 de E falsifiée par n_d

- Puisque n n'est pas un nœud d'échec, C'_1 doit avoir la forme :

$$\underbrace{\neg A(t_1, \cdots t_n) \vee \cdots \neg A(t_1, \cdots, t_n)}_{p\ fois} \vee D'_1$$

  où $p \geq 1$ et $\neg A(t_1, \cdots t_n)$ n'est pas un disjoint de la clause D'_1. La clause C_1 de E correspondante aura la forme :

$$\underbrace{\neg A(s_1, \cdots s_n) \vee \cdots \neg A(s'_1, \cdots, s'_n)}_{p\ fois} \vee D_1$$

- De même, C'_2 doit avoir la forme :

$$\underbrace{A(t_1, \cdots, t_n) \vee \cdots \vee A(t_1, \cdots t_n)}_{q\ fois} \vee D'_2$$

  où $q \geq 1$ et $\neg A$ n'est pas un disjoint de la clause D'_2. La clause C_2 de E correspondante aura la forme :

$$\underbrace{A(m_1, \cdots m_n) \vee \cdots A(m'_1, \cdots, m'_n)}_{p\ fois} \vee D_2$$

On a alors les faits suivants :

1. Il existe une déduction par résolution de la clause $D'_1 \vee D'_2$ à partir des clauses C'_1 et C'_2, car il suffit d'appliquer respectivement $p-1$ fois et $q-1$ fois la règle de factorisation à partir de ces deux clauses, puis d'effectuer une résolution qui "coupe" les littéraux $\neg A(t_1, \cdots, t_n)$ et $A(t_1, \cdots, t_n)$.

2. Donc, par le lemme 83, il existe une clause C_3 telle que :
   - il existe une déduction par résolution d de C_3 à partir de C_1 et C_2 (donc aussi à partir de E)
   - $D'_1 \vee D'_2$ est une instance close de C_3.

Maintenant, observons que l'instance close $D'_1 \vee D'_2$ de la clause C_3 est forcément falsifiée par le nœud n (sinon ni n_g ni n_d ne pourraient être un nœud d'échec). Il s'ensuit que si l'on modifie l'arbre A en éliminant les successeurs du nœud n, on obtient un arbre sémantique clos A' pour le nouvel ensemble de clauses $E \cup \{C_3\}$. La hauteur de A' est strictement inférieure à celle de A. Donc, par hypothèse de récurrence, il existe une réfutation r' de $E \cup C_3$. On peut enfin conclure qu'il existe une réfutation r de E, car il suffit de faire :

1. D'abord la déduction d de C_3

2. Ensuite prolonger cette déduction en y ajoutant la réfutation r'

∎

FIGURES

4 PROLOG

Dans cette section, nous continuons la présentation du langage PROLOG commencée dans l'Introduction et nous indiquons les liens entre PROLOG et la méthode de preuve par résolution.

4.1 L'évaluation des requêtes

Tout d'abord, un peu de terminologie.

- Si $p(t_1, \cdots, t_n)$ et $r(s_1, \cdots, s_n)$ sont deux **atomes** tels que la substitution σ est un mgu du problème d'unification $\{t_1 = s_1, \cdots, t_n = s_n\}$, on dira que les deux atomes sont unifiables avec mgu σ.
- Soit $R =$:- $A_1, \cdots, A_m$, où $m > 0$, une clause requête et soit $C = A'_1 : -B_1, \cdots, B_p$, où $p \geq 0$ une clause de programme. Si les deux atomes A_1 et A'_1 sont unifiables avec mgu σ, alors la nouvelle clause requête
R' = :- $(B_1, \cdots B_p, A_2, \cdots, A_m)_\sigma$ est dite *résolvante*-PROLOG de R et C. Observer que si $m = 0$, alors R' est vide.

Algorithme non-déterministe d'évaluation de PROLOG

Entrées : un programme PROLOG P et une clause requête R.
Sortie : ou bien une substitution σ, dite "Réponse", pour les variables de R (dans le cas particulier où R ne contient pas de variables, σ est vide et on renvoi le message "YES"), ou bien ECHEC.

1. INITIALISATION. Initialiser la requête courante R_c à R et le mgu courant mgu_c à la substitution vide.
2. BOUCLE CENTRALE.
Tant que la requête courante R_c= :- $G_1, \cdots G_k$ n'est pas vide, **faire** :
   (a) *Choisir* une clause de programme C, *renommée* avec des variables *nouvelles*, disons $C = G'_1 : -D_1, \cdots, D_t$, telle que les atomes G_1 et G'_1 soient unifiables avec un mgu σ. Si une telle clause n'existe pas dans P, sortir de la boucle;
   (b) Affecter R_c à la résolvante-PROLOG de G_1 et C, qui aura la forme : :- $(D_1, \cdots, D_t, G_2, \cdots, G_k)_\sigma$. Affecter mgu_c à la substitution obtenue en composant l'ancienne valeur de mgu_c avec σ.
3. SORTIE. Si R_c est vide, alors
   - Calculer la restriction de mgu_c aux variables de R;
   - si cette restriction n'est pas vide, la renvoyer comme valeur de Réponse
   - si cette restriction est vide, renvoyer "YES".

   Si R_c n'est pas vide, renvoyer ECHEC.

Etant donnés un programme PROLOG P et une requête R, une évaluation de R peut donner lieu a un des trois cas suivants :

1. l'évaluation termine avec succès (cas où Réponse ou bien le message "YES" est renvoyé)
2. l'évaluation termine avec ECHEC
3. l'évaluation ne termine pas.

EXEMPLE

Le programme PROLOG suivant définit la relation *ancêtre.* Intuitivement, on lit un atome de la forme parent(t_1, t_2) comme "t_1 est un parent (père ou mère) de t_2" et un atome de la forme ancetre(t_1, t_2) comme "t_1 est un ancêtre de t_2". On numerote les clauses de programme pour simplifier la description de l'exemple; dans la suite, on fera référence à ce programme comme $ANCETRE_1$.

1. parent(toto, marie)
2. parent(louise, toto)
3. parent(franc, jean)
4. ancetre(X,Y) :- parent(X,Y).
5. ancetre(X,Y) :- ancetre(X,Z), parent(Z,Y).

Soit R la requête :- ancetre(W,marie), dont le rôle est d'interroger le programme sur quels sont les ancêtres de marie.

Nou donnons ci-dessous la trace d'une évaluation de R qui *termine avec succès.* Les nombres 4 et 1 indiquent les clauses de P utilisées pour calculer les résolvantes-PROLOG constituant les nouvelles requêtes courantes. A chaque étape, σ indique la substitution permettant d'unifier le premier atome de la clause courante avec la tête de la clause de programme utilisée. Observer que chaque fois qu'une clause de programme est utilisée, ses variables sont renommées; par exemple, la clause 4 est renommée en : ancetre(X1,Y1) :- parent(X1,Y1).

ancetre(W,marie)

| 4 avec $\sigma = \{W \rightarrow X1, Y1 \rightarrow marie\}$

parent(X1,marie)

| 1 avec $\sigma = \{X1 \rightarrow toto\}$

$\emptyset$

A la sortie, mgu_c a la valeur $\{W \rightarrow toto, Y1 \rightarrow marie\}$; puisque W est la seule variable de R, la valeur de Réponse est $\{W \rightarrow toto\}$.

La trace suivante est celle d'une autre évaluation de R, qui choisit d'autres clauses mais qui *termine toujours avec succès.*

ancetre(W,marie)

5 avec $\sigma = \{W \rightarrow X1, Y1 \rightarrow marie\}$

ancetre(X1,Z1), parent(Z1, marie)

4 avec $\sigma = \{X1 \rightarrow X2, Z1 \rightarrow Y2\}$

parent(X2,Y2), parent(Y2,marie)

2 avec $\sigma = \{X2 \rightarrow louise, Y2 \rightarrow toto\}$

parent(toto,marie)

1 avec $\sigma = \emptyset$

$\emptyset$

A la sortie, mgu_c a la valeur $\{W \rightarrow louise, X1 \rightarrow louise, X2 \rightarrow louise, Y1 \rightarrow marie, Y2 \rightarrow toto, Z1 \rightarrow toto\}$; puisque W est la seule variable de R, la valeur de Réponse est $\{W \rightarrow louise\}$.

L'évaluation ci-dessous de R termine avec ECHEC.

ancetre(W,marie)

5 avec $\sigma = \{W \rightarrow X1, Y1 \rightarrow marie\}$

ancetre(X1,Z1), parent(Z1, marie)

4 avec $\sigma = \{X1 \rightarrow X2, Z1 \rightarrow Y2\}$

parent(X2,Y2), parent(Y2,marie)

3 avec $\sigma = \{X2 \rightarrow franck, Y2 \rightarrow jean\}$

parent(jean,marie)

qui n'est pas $\emptyset$

Le cas suivant est celui d'une évaluation de R qui *ne termine pas.*

ancetre(W,marie)

5 avec $\sigma = \{W \rightarrow X1, Y1 \rightarrow marie\}$

ancetre(X1,Z1), parent(Z1, marie)

5 avec $\sigma = \{X1 \rightarrow X2, Z1 \rightarrow Y2\}$

ancetre(X2,Z2), parent(Z2,Y2), parent(Y2,marie)

5 avec $\cdots$

$\vdots$

5 avec $\cdots$

ancetre(X_i,Z_i),parent(Z_i,Y_i),..., parent(Y2,marie)

5 avec $\cdots$

$\vdots$

pas de terminaison !

Fin de l'exemple.

L'algorithme d'évaluation indiqué est non-déterministe, car la première instruction de la boucle demande de “choisir” une clause appropriée. Mais, évidemment, l'implémentation sous-jacente PROLOG est déterministe : l'évaluateur PROLOG “choisit” la clause selon un critère précis : il considère les clauses selon leur ordre d'écriture. Ceci revient à *explorer en profondeur d'abord* ce qu'on appelle l'*arbre de recherche PROLOG*.

Arbre de Recherche PROLOG

Dans cet arbre, associé à un programme P et une requête R données, chaque branche représente une évaluation possible. Ainsi, on peut avoir des branches finies correspondant à des évaluations qui terminent avec succès, des branches finies correspondant à des évaluations qui terminent avec échec, et des branches infinies correspondant à des évaluations qui ne terminent pas.

Définition 85 (Arbre de Recherche PROLOG) Un arbre de recherche PROLOG pour un programme P et une requête R est un arbre dont les nœuds sont des clauses requête et tel que :

- La racine est R
- Si un nœud est une requête non-vide :- $A_1, \cdots, A_n$ (où $n > 0$) et $C_1, \cdots, C_k$, avec $k > 0$, sont les clauses de programme (dans leur ordre d'écriture en P!) dont la tête s'unifie avec l'atome A_1, alors ce nœud a les k fils $Res_1, \cdots, Res_k$, où, pour $1 \leq i \leq k$, Res_i est la résolvante-PROLOG de A_1 avec la clause C_i.

- Si un nœud est une requête non-vide :- $A_1, \cdots, A_n$ (où $n > 0$) et s'il n'existe pas de clause de programme dont la tête s'unifie avec A_1, alors ce nœud est une *feuille d'échec.*

- Si un nœud est la requête vide, alors ce nœud est une *feuille de succès.*

Exemple. Arbre de recherche pour le programme $ANCETRE_1$ et la requête :- ancetre(W,marie).

Comme déjà dit, l'évaluateur PROLOG explore l'arbre de recherche pour P et R en profondeur d'abord. Il procède ainsi :

- Si la branche courante termine avec succès, l'évaluation s'arrête en donnant la réponse correspondante.
- Si la branche courante termine avec échec, on considére alors la branche suivante. Ceci revient à passer à la prochaine clause applicable pour la requête correspondant au nœud parent du nœud courant ("Backtracking").
- Si, après avoir fourni une réponse, il reçoit par l'utilisateur une instruction de poursuite de recherche, on considére la branche suivante comme si la dernière branche considérée avait échoué ("Backtracking" à nouveau), en donnant ainsi les éventuelles réponses suivantes.
- Si la branche courante ne termine pas, l'évaluateur ne s'arrête pas.

Ainsi :

1. Les réponses à une requête R sont données dans un ordre qui dépend de l'ordre d'écriture des clauses et des atomes des corps des clauses du programme.
2. Si jamais l'arbre de recherche contient une branche infinie, l'évaluateur peut entrer en loop après avoir fournit des réponses, ou même avant de fournir la première réponse.
3. La réponse "Non" correspond au cas où l'arbre est *fini* et *toute* branche termine avec une feuille d'échec.

Ordre des clauses et des atomes

L'ordre d'écriture des *atomes* du corps d'une clause est extrêmement important par rapport à la terminaison de l'évaluation. En effet, *l'ordre des atomes du corps des clauses détermines la structure de l'arbre de recherche.*

Exemple des effets du changement de l'ordre des atomes.
Modifions le programme $ANCETRE_1$ en remplaçant la clause 5 par la clause 5bis suivante : ancetre(X,Y) :- parent(Z,Y), ancetre(X,Z). L'arbre de recherche pour le programme ainsi obtenu - on l'appellera $ANCETRE_2$ - et la requête :- ancetre(W,marie) *ne contient pas de branche infinie*, comme il est facile de le vérifier. Ceci implique que PROLOG peut donner *toutes* le réponses ($W = toto, W = louise$) et puis *s'arrêter*, en vérifiant qu'il n'y a pas d'autres réponses possibles. Ceci n'était pas le cas du programme $ANCETRE_1$, où PROLOG entrait en loop après avoir fourni les deux réponses.

L'ordre d'écriture des *clauses* est aussi important :

1. Si l'on change l'ordre d'écriture des clauses d'un programme dont l'arbre de recherche, pour une requête R donnée, est fini, le nouvel arbre de recherche pour R est toujours fini, mais l'ordre dans lequel on obtient les réponses peut changer.

2. Si l'on change l'ordre d'écriture des clauses d'un programme dont l'arbre de recherche, pour une requête R donnée, contient une branche infinie, on peut obtenir un nouvel arbre de recherche pour R tel que PROLOG entre en loop avant de fournir une réponse.

Exemple des effets du changement de l'ordre des clauses.
Modifions à nouveau le programme $ANCETRE_1$, cette fois-ci en inversant l'ordre des clauses 4 et 5; appelons $ANCETRE_3$ le nouveau programme. Dans la *première* branche de l'arbre de recherche pour ce programme et la requête :- ancetre(W,marie), on applique systématiquement la règle ancetre(X,Y) :- ancetre(X,Z), parent(Z,Y).; il est facile de vérifier que cette branche est infinie; donc l'évaluateur PROLOG entre en loop tout de suite, sans pouvoir fournir aucune réponse.

A la lumière de ces remarques, il est clair que la façon correcte d'écrire le programme pour la relation ancêtre est celle de $ANCETRE_2$. Dans le PROLOG "réel", il existe aussi une instruction, appelée "cut", permettant de couper certains sous-arbres d'un arbre de recherche, et de pallier ainsi au problème causé par la sensibilité de PROLOG à l'ordre d'écriture. Toutefois, dans ce cours, on ne considère que PROLOG "pur", qui ne contient aucune instruction qui n'aie pas de sémantique logique claire (c'est le cas du "cut").

4.2 Termes et structures de données

En PROLOG pur il n'existe pas de structure de donnée prédefinie (même si dans le PROLOG "réel" les listes avec leur opérations standard et les entiers avec les opérations arithmétiques sont prédéfinis). Ainsi, toute représentation d'une structure des données se fait au moyen des termes et c'est à l'utilisateur de choisir des termes aptes à représenter la structure voulue.

Par exemple, supposons que l'on veuille écrire des procédures pour les arbres binaires. Une façon possible de représenter un arbre binaire est la suivante :

- L'arbre vide est représentée par une constante, par exemple le mot "vide".
- Tout arbre binaire non-vide est représenté par un terme de la forme arbre(t_1, t_2, t_3), où le mot "arbre" est un symbole de fonction à 3 arguments; le terme t_1 représentera la racine, le terme t_2 représentera le sous-arbre gauche et le terme t_2 représentera le sous-arbre droit.

Ainsi, par exemple, le terme arbre(a,arbre(b,vide,vide),arbre(c,vide,vide)) représente

```
           a
l'arbre :  /\
          b  c
```

Nous donnons ci-dessous la définition d'un prédicat qui, d'un coté, *établit*, à l'intérieur du programme, la façon choisie de représenter les arbres binaires, et, de l'autre, permet de tester si un terme a bien la syntaxe de la représentation des arbres choisie.
estunarbre(vide).
estunarbre(R,G,D) :- estunarbre(G), estunarbre(D).

La clause suivante suffit à définir un prédicat qui teste si un terme est un arbre binaire non-vide dont la racine est la lettre "a".

racinea(arbre(a,X,Y)) :- estunarbre(X), estunarbre(Y).

Donnons maintenant la définition d'une relation binaire pour tester si un élément apparaît comme nœud d'un arbre binaire.
in(E,arbre(E,G,D)) :- estunarbre(G), estunarbre(D).
in(E,arbre(X,G,D)) :- estunarbre(G), estunarbre(D), in(E,G).
in(E,arbre(X,G,D)) :- estunarbre(G), estunarbre(D), in(E,D).

4.3 Évaluation PROLOG et résolution

Clauses de Horn et clauses PROLOG

Jusqu'à ici, on a utilisé le mot "clause" de façon ambigüe : au sens de "clause logique" (disjonction de littéraux) et au sens d'instruction PROLOG. Cette ambigüité était voulue; dans cette section, on montre que l'écriture logique d'une clause de programme ou d'une clause requête est une clause au sens logique.

La formule logique associée à une clause de programme de la forme A_1 :- $B_1, \cdots, B_m$, avec $m > 0$, dont les variables sont $X_1, \cdots, X_n$, est $\forall x_1, \cdots, \forall x_n((B_1 \wedge \cdots \wedge B_m) \rightarrow A_1)$; cette dernière est logiquement équivalente à la clause (au sens logique) : $\neg B_1 \vee \cdots \vee \neg B_m \vee A_1$. Par exemple, la transcription logique de la clause 5 du programme $ANCETRE_1$ est $\neg ancetre(X,Z) \vee \neg parent(Z,Y) \vee ancetre(X,Y)$. La formule logique associée à une clause de programme de la forme A_1 :-, dont les variables sont $X_1, \cdots, X_n$, est $\forall x_1, \cdots \forall x_n(\mathcal{V} \rightarrow A_1)$, équivalente à la clause logique A_1.

On remarquera que toute clause, au sens logique, correspondant à une clause de programme est telle que *un et un seul littéral positif est membre de la disjonction.*

Considérons une clause requête, de la forme : $-B_1, \cdots, B_m$. Ce que l'on a dit pour les clauses de programme suggère que sa transcription logique est la formule $\forall x_1, \cdots, \forall x_n(\mathcal{F} \rightarrow (B_1 \wedge \cdots \wedge B_m))$, équivalente à la clause logique $\neg B_1 \vee \cdots \vee \neg B_m$. Ceci peut sembler bizarre si on pense à la lecture intuitive d'une telle clause suggérée au début du cours, notamment :
est-il vrai qu'il existe des valeurs des variables telles que $B_1, \cdots, B_m$ sont tous vrais ? Si oui, quelles sont ces valeurs ?
Mais on verra qu'une évaluation PROLOG de la requête : $-B_1, \cdots, B_m$ est une preuve du fait que la formule existentielle $\exists x_1, \cdots, \exists x_n(B_1 \wedge \cdots \wedge B_m)$ est une conséquence logique de l'ensemble C des clauses logiques correspondant aux clauses du programme; cette preuve consiste en une réfutation de l'ensemble de clause $C \cup \{$ forme clausale de la négation de $\exists x_1, \cdots, \exists x_n(B_1 \wedge \cdots \wedge B_m)\}$. Or, la forme clausale de la négation de $\exists x_1, \cdots, \exists x_n(B_1 \vee \cdots \vee B_m)$ est exactement $\neg B_1 \vee \cdots \vee \neg B_m$.

On remarquera que toute clause, au sens logique, correspondant à une clause requête est telle que *aucun littéral positif est membre de la disjonction.*

Dans la définition suivante, le mot "clause" est utilisé au sens logique.

Définition 86 (Clauses de Horn) Une clause *définie* est une clause contenant un et un seul littéral positif. Une clause *négative* est une clause ne contenant pas de littéraux positifs. Une clause *de Horn* est ou bien une clause définie ou bien une clause négative.

On voit donc que :

1. Toute clause de programme correspond à une clause *définie.*
2. Toute clause requête correspond à une clause *négative.*
3. L'ensemble des clauses constitué des transcription logiques des clauses de programme et de la transcription logique d'une requête est un ensemble de *clauses de Horn.*

Evaluation PROLOG comme recherche d'une réfutation

Considérons à nouveau le programme $ANCETRE_1$ et la branche suivante de l'arbre de recherche pour la requête ancetre(W,marie) :-.

ancetre(W,marie)

5 avec $\sigma = \{W \to X1, Y1 \to marie\}$

ancetre(X1,Z1), parent(Z1, marie)

4 avec $\sigma = \{X1 \to X2, Z1 \to Y2\}$

parent(X2,Y2), parent(Y2,marie)

2 avec $\sigma = \{X2 \to louise, Y2 \to toto\}$

parent(toto,marie)

1 avec $\sigma = \emptyset$

$\emptyset$

A la lumière de la correspondance vue entre clauses PROLOG et clauses logiques, elle correspond à la réfutation suivante (pour des raisons d'espace, on écrit "p" à la place de "parent", "a" à la place de "ancetre", "t" à la place de "toto", "l" à la place de "louise", "m" à la place de "marie") :

$$\dfrac{\dfrac{\dfrac{\dfrac{\neg a(W,m) \qquad a(X,Y) \vee \neg a(X,Z) \neg p(Z,Y)}{\neg a(X_1,Z1) \vee \neg p(Z_1,m)}\,RES \qquad a(X,Y) \vee \neg p(X,Y)}{\neg p(X_2,Y_2) \vee \neg p(Y_2,m)}\,RES \qquad p(l,t)}{\neg p(t,m)}\,RES \qquad p(t,m)}{\emptyset}\,RES$$

Une branche qui termine avec succès d'un arbre de recherche PROLOG n'est donc rien d'autre qu'une réfutation par résolution de l'ensemble des clauses de Horn correspondant à la transcription logique du programme e de la requête. On peut donc voir une évaluation qui termine avec succès come une preuve par résolution du fait que une formule de la forme $\exists x_1, \cdots, x_n(B_1 \wedge \cdots \wedge B_m)$ est conséquence logique du programme; de plus cette preuve est "constructive", au sens qu'elle fournit des valeurs appropriées pour les variables $x_1, \cdots, x_n$.

Une réfutation correspondant à une évaluation PROLOG a toujours les propriétés suivantes.
1) La règle de factorisation n'est jamais utilisée.

2) La réfutation a la forme d'arbre binaire "à peigne" suivante :

$$\frac{c_1 \qquad c_2}{c_3}\ RES$$
$$\vdots$$
$$\frac{c_i \qquad\qquad c_{i+1}}{c_{i+2}}\ RES$$
$$\vdots$$
$$\frac{c_n \qquad\qquad c_{n+1}}{\emptyset}\ RES$$

où chaque clause de la branche "centrale" (la clause c_1, dite *racine*, inclue), est négative.

Une réfutation d'un ensemble de clauses C ayant ces deux propriétés est dite *linéaire input.* Si C est un ensemble *quelconque* de clauses, il est faux que si C est insatisfiable alors il existe une réfutation de C qui est linéaire input. Toutefois, si C est un ensemble de clauses de Horn, l'insatisfiabilité de C implique l'existence d'une réfutation linéaire input. Donc, *dans le cas des clauses de Horn, il est possible de se limiter à considérer les réfutations linéaires input quand on étudie l'insatisfiabilité*; en d'autres termes, la *stratégie linéaire input est complète pour les clauses de Horn.* Puisque, comme déjà vu, PROLOG ne manipule que des clauses de Horn, l'évaluateur PROLOG se limite à engendrer des réfutations linéaires input pour répondre à une requête. De plus, étant donnée une clause négative $\neg B_1 \vee \neg B_n$, dont lécriture PROLOG est :- $B_1, \cdots B_m$, l'évaluateur choisit toujours B_1 (le premier atome dans l'écriture PROLOG de la clause) comme atome à unifier avec la tête d'une clause définie (clause de programme). La stratégie de recherche des réfutation suivie par PROLOG est dite *stratégie SLD*; cet acronyme vient de l'anglais : **L**inear **Resolution** with **S**election function (la fonction de sélection est celle qui associe à une clause requête :- $B_1, \cdots B_m$ l'atome B_1 comme prochain atome à traiter).

Une dernière remarque : bien que la stratégie SLD soit complète pour les clauses de Horn, il est tout à fait possible que PROLOG n'arrive pas à fournir *toutes* les réponses souhaitées pour une requête. Dans ce sens, la *stratégie PROLOG est incomplète.* Mais cet inconvénient est du exclusivement au fait que, pour des raisons d'efficacité, l'évaluateur PROLOG fait un parcours de l'arbre de recherche en profondeur d'abord et qu'une branche infinie peut précedér une branche de réusssite; on n'aurait pas ce type de problème si la recherche était faite en largeur (mais on aurait une recherche très coûteuse).

5 Bibliographie

5.1 Quelques livres de logique

1. J-P. Delahaye, Outils Logiques pour l'Intelligence Artificielle, Eyrolles
   Très facile à suivre, mais assez incomplet. En particulier, il ne contient rien sur le calcul des séquents. Par contre, il est très bien pour ce qui concerne la méthode de résolution

2. J. H. Gallier, Logic for Computer Science, John Wiley & Sons.
   Très bon livre. Il contient beaucoup de choses, même trop. Il faut savoir s'y repérer

3. H. Enderton, A Mathematical Introduction to Logic, Academic Press.
   Il traite surtout la sémantique. Il ne contient presque rien sur les systèmes de preuve; en particulier, rien sur la méthode de résolution

5.2 Quelques livres sur PROLOG

1. L. Sterling & E. Shapiro, L'art de Prolog, Eyrolles (traduction en francais de "The art of Prolog", MIT Press).

2. W. F. Clocksin, C. s. Mellish, Programming in Prolog, Springer-Verlag.