Chrysostome Baskiotis

LOGIQUE COMPUTATIONNELLE & PROLOG

Fascicule 2

Calcul des prédicats
Programmation logique

EISTI

Année 2011

6

CALCUL DES PRÉDICATS

Pour pouvoir utiliser la logique en tant que mode de calcul il faut d'abord procéder à une conceptualisation de la partie du monde qui nous intéresse, c'est-à-dire établir les objets du monde et leurs inter-relations. Il s'agit, en réalité, de procéder à une *représentation de la connaissance*. Les objets sur lesquels nous allons exprimer notre connaissance constituent l'*univers du discours*. La représentation de la connaissance est la mise en relation des plusieurs objets de l'univers du discours par l'intermédiaire de la formulation des fbf.

Avec le calcul propositionnel on ne peut pas formaliser toutes les connaissances relatives à un univers du discours. Par exemple la phrase « tous les élèves aiment la logique » sera vue en calcul propositionnele comme une proposition p. On ne pourra donc pas la distinguer d'une autre du type « il existe des élèves qui aiment la logique ». qui sera, elle aussi, notée p. Il faut donc introduire dans le langage des symboles qui désignent soit l'existence, soit l'universalité et, aussi, des variables.

Nous avons ainsi un cas particulier de ce qu'on pourrait considérer comme étant des *fonctions propositionnelles logiques* et qui sont des fonctions au sens classique du terme mais dont le résultat est une valeur de vérité - 0 (faux) ou 1 (vrai). Ces fonctions propositionnelles logiques on les appellera, par la suite, *prédicats*.

Il existe bien sûr toujours des fonctions au sens classique du terme, c-à-d. des fonctions dont le résultat est un objet de l'univers du discours et non pas une valeur de vérité. Pour les distinguer des autres fonctions, qui sont les prédicats, nous les appellerons *foncteurs*.

Nous allons maintenant, comme nous l'avons fait au chapitre précédent, développer un langage qui permettra la création et l'étude des prédicats. C'est l'objet du présent chapitre. On débute avec la définition du langage. On continue avec les évaluations sémantique et syntaxique d'une fbf avant d'aborder leur équivalence. On termine avec la définition des formes clausales.

6.1 Les éléments du langage

Pour l'étude des prédicats nous allons définir un langage formel $\mathcal{L}_1$ dont l'alphabet A_1 est composé des éléments suivants :

- Un ensemble **V**, au plus dénombrable, des variables qui seront notées $x, y, \ldots$ ou $X, Y, \ldots$.
- Un ensemble Ξ, au plus dénombrable, des constantes qui seront notées $a, b, \ldots$.
- Un ensemble **F** de fonctions $f : \mathbf{V} \times \Xi \rightarrow \mathbf{V} \cup \Xi$ d'arité quelconque. Notons que l'arité d'une fonction est le nombre d'arguments de la fonction. Remarquons qu'une constate est une fonction d'arité zéro. On appelle ces fonctions des *foncteurs* que l'on notera $f, g, \ldots$ ou $F, G, \ldots$.
- Un ensemble **P** de fonctions d'arité quelconque $f : \mathbf{V} \times \Xi \rightarrow \{Vrai, Faux\}$ qui seront appelées *prédicats* et qui seront notés par $p, q, \ldots$ ou $P, Q, \ldots$. Un prédicat particulier est la relation d'égalité qui sera notée soit de façon fonctionnelle $eg(x, y)$, soit, lorsqu'il n'y a pas risque de confusion, $x = y$. À ce propos, notons qu'il faut clairement distinguer entre *égalité* "$=$" et *équivalence* "$\equiv$". La égalité entre deux termes est un prédicat qui indique si oui ou non les deux termes ont la même valeur. L'équivalence entre deux fbf indique que les deux fbf ont la même table de vérité. La question peut se poser si deux termes égaux sont équivalents. La réponse est non. Par exemple nous avons "10= 3+7" mais par rapport au prédicat *nombre pair* il n'y a pas d'équivalence. De même deux fbf équivalentes ne sont pas égales. Par exemple la fbf $a \vee \neg a$ est équivalente à la fbf $a \vee (a \rightarrow b)$ car toutes les deux sont des tautologies, mais elles ne sont pas égales.
- Un ensemble **L** des connecteurs et quantificateurs :
  - Connecteur logique unaire : la négation $\neg$
  - Connecteurs propositionnels binaires :
    - Conjonction : $\wedge$
    - Disjonction : $\vee$
    - Implication : $\rightarrow$
    - Équivalence (ou double implication) : $\leftrightarrow$
  - Quantificateurs :
    - Quantificateur existentiel : $\exists$
    - Quantificateur universel : $\forall$
- Les séparateurs (symboles auxiliaires) : $($, $)$, $[$, $]$.

Les séparateurs ne font pas partie, à proprement parler, du langage. Leur présence permet de faciliter la lecture des formules.

Les éléments du langage déterminent un alphabet $A_1 = \{\mathbf{V}, \Xi, \mathbf{F}, \mathbf{P}, \mathbf{L}\}$.

6.1.1 Les termes

La brique élémentaire du calcul des prédicats est le *terme*. Une variable est un terme. Une constante aussi. Plus généralement

DÉFINITION 6.1.1 *Un* terme *est défini de façon itérative comme suit :*

- *une constante est un terme ;*
- *une variable est un terme ;*
- *si f est un foncteur d'arité n et $t_1, t_2, \ldots, t_n$ sont des termes, alors $f(t_1, t_2, \ldots, t_n)$ est un terme.*

Tout terme est obtenu par application des règles précédentes un nombre fini de fois.
L'ensemble de termes sera noté par $\mathbb{T}$.

On peut construire, à partir des termes, des formules bien formées (fbf).

DÉFINITION 6.1.2 *Soit $\mathbb{T}$ l'ensemble des termes sur un alphabet A_1. L'ensemble $\mathbb{F}$ des* formules bien formées *(par rapport à A_1) est le plus petit ensemble tel que :*

- *Si p est un prédicat d'arité n et $t_1, t_2, \ldots, t_n$ sont des termes, alors $p(t_1, t_2, \ldots, t_n) \in \mathbb{F}$.*
- *Si F , $G \in \mathbb{F}$ alors les constructions suivantes :*
  - $\neg F$
  - $F \vee G$, $F \wedge G$
  - $F \rightarrow G$, $F \leftrightarrow G$

  sont aussi des fbf.
- *Si $F \in \mathbb{F}$ et X est une variable, alors $(\forall X) F \in \mathbb{F}$ et $(\exists X) F \in \mathbb{F}$.*

ASCÈSE 6.1 *Donner l'équivalent en français de deux fbf suivantes :*

(1) $\exists X \, (p(X) \wedge (\forall Y \, p(Y) \rightarrow X = Y))$

(2) $(\exists X \, p(X)) \wedge (\forall X \forall Y \, p(X) \wedge p(Y) \rightarrow X = Y)$

L'équivalent de l'atome de la logique des propositions est donné par la définition suivante :

DÉFINITION 6.1.3 *Si p est un prédicat d'arité n et $t_1, t_2, \ldots, t_n$ sont des termes, alors $p(t_1, t_2, \ldots, t_n)$ est une* formule atomique.

DÉFINITION 6.1.4 *Le triplet*

$$\mathcal{L}_1 = \{A_1, \mathbb{T}, \mathbb{F}\}$$

est le langage d'ordre un *ou le* langage du calcul des prédicats

Par construction $\mathbb{F}$ est un ensemble dénombrable, défini itérativement.

ASCÈSE 6.2 *On se place dans l'ensemble des nombres réels. Écrire, en utilisant le langage des prédicats, les propositions suivantes :*

(1) Pour tout réel il existe un autre réel qui est plus grand.

(2) Il existe un réel qui est plus grand que tout autre réel.

(3) *Chaque réel positif est un carré.*

(4) *Si un réel est plus petit qu'un autre réel, alors il existe un troisième réel, différent de deux précédents, et qui est entre les deux.*

ASCÈSE 6.3 *Pour les propositions de l'ascèse précédent, déterminer les prédicats et les foncteurs utilisés.*

ASCÈSE 6.4 *Soit les textes suivants :*

(1) *Tous les hommes sont mortels. Socrate est un homme. Donc, Socrate est mortel.*

(2) *Toute personne saine d'esprit peut comprendre la logique. Aucun des fils de Socrate ne peut comprendre la logique. Aucune personne non saine d'esprit a le droit de vote. Donc, aucun des fils de Socrate n'a le droit de vote.*

Pour chaque texte

(1) *Écrire la fbf correspondante.*

(2) *Montrer que chaque fbf est valide.*

6.1.2 Les quantificateurs et leur portée

Les quantificateurs ont une *portée*.

DÉFINITION 6.1.5 *La* portée *d'un quantificateur dans une formule est la partie de la formule qui se trouve sous l'influence du quantificateur.*
Une variable d'une formule qui est sous la portée d'un quantificateur de la même variable est appelée variable liée *(ou* bornée*). Sinon elle est une* variable libre.
Une formule qui n'a pas des variables libres s'appelle formule close. *Une formule qui n'a pas des variables s'appelle* formule filtrée.
Une formule sans quantificateurs s'appelle formule ouverte.

ASCÈSE 6.5 *Pour les fbf ci-après indiquer les variables libres et les variables liées.*

(1) $\exists y\ (p(r) \wedge f(x,y) \vee q(y) \wedge f(x,y)) \leftrightarrow p(r) \vee q(x) \vee p(y)$

(2) $\forall x \exists y\ (p(w) \vee q(z)) \vee \exists z\ (q(w) \vee p(r))$

(3) $\forall x \exists y\ (p(w) \vee q(z)) \vee \exists z\ (q(w) \vee p(y))$

(4) $\forall x\,(\forall z p(y) \wedge \neg\,(q(z) \wedge s(y)) \rightarrow \exists y\,(q(y) \vee p(x) \vee s(z))) \wedge (p(z) \wedge s(y))$

Pour résoudre les problèmes de l'ambiguïté entre démonstration sous forme conditionnelle et démonstration sous forme générale, on introduit la définition ci-après :

DÉFINITION 6.1.6 *Soit F une fbf avec variables libres $x_1, x_2, \ldots, x_n$.*
La clôture universelle *de F , notée $\forall F$, est la formule close $\forall x_1, \forall x_2, \ldots, \forall x_n\ F$.*
La clôture existentielle *de F, notée $\exists F$, est la formule close $\exists x_1, \exists x_2, \ldots, \exists x_n\ F$.*

Notons qu'il est obligatoire quand on passe d'une fbf F à une clôture (soit universelle, soit existentielle) de renommer, par substitution, les variables qui ont le même nom mais qui ne dépendent pas du même quantificateur afin d'éliminer les ambiguïtés. Par exemple pour la fbf $\forall x \exists y\,(p(x) \wedge q(y)) \vee (\exists y \neg p(x) \wedge q(y))$ on doit renommer la deuxième occurrence de x et de y et obtenir ainsi $\forall x \exists y\,(p(x) \wedge q(y)) \vee (\exists z \neg p(x) \wedge q(z))$.

ASCÈSE 6.6 *On se place dans le cadre des nombres entiers et on suppose que nous disposons des opérations (foncteurs) d'addition ($+$) et de multiplication ($\times$). Donner, en langage des prédicats, la définition des prédicats suivants :*

(1) $pair(x)$ = *x est un nombre pair.*

(2) $div(x,y)$ = *x divise y, c'est-à-dire que* y/x *est un nombre entier.*

(3) $premier(x)$ = *x est un nombre premier.*

ASCÈSE 6.7 *En appliquant la définition des prédicats pair et div, écrire le prédicat* $div2(x)$ = 2 *divise le nombre pair x.*

6.1.3 Propriétés des connecteurs

Les principales propriétés des connecteurs sont données ci-après :

(1) Double négation

$$p \leftrightarrow \neg\neg p$$

(2) Loi de de Morgan

$$\neg(p \vee q) \leftrightarrow \neg p \wedge \neg q$$
$$\neg(p \wedge q) \leftrightarrow \neg p \vee \neg q$$

(3) Définition de l'implication

$$(p \rightarrow q) \leftrightarrow (\neg p \vee q)$$

(4) Introduction de l'implication

$$p \rightarrow (q \rightarrow p)$$

(5) Distributivité de l'implication

$$(p \rightarrow (q \rightarrow r)) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow r))$$

(6) Contradiction

$$(p \rightarrow q) \rightarrow ((p \rightarrow \neg q) \rightarrow \neg p)$$

(7) Négation des quantificateurs universel et existentiel :

$$\neg(\forall X) \leftrightarrow (\exists X)\ ,\ \neg(\exists X) \leftrightarrow (\forall X)$$

Il est important de se rappeler que nous avons pour les négations les relations suivantes :

$$(\neg\forall X\ p(X)) \leftrightarrow (\exists X\ \neg p(X))$$

et

$$(\neg\exists X\ p(X)) \leftrightarrow (\forall X\ \neg p(X))$$

ASCÈSE 6.8 *Traduire en langage des prédicats dans* $\mathbb{N}$ *la phrase suivante :*

(1) $x \equiv y$ *modulo n.*

6.2 Substitution

Pendant le traitement d'une fbf, les variables peuvent être remplacées (substituées) par des termes ou, encore, par des fbf.

DÉFINITION 6.2.1 *Une* substitution *est un ensemble fini de couples de la forme* $(x_1/A_1, x_2/A_2, \ldots, x_n/A_n)$ *où chaque* A_i *est soit un terme, soit une fbf et chaque* x_i *une variable telle que* $x_i \neq A_i$ *et* $A_i \neq A_j$ *si* $x_i \neq x_j$.

La substitution vide sera notée par ε.

La substitution peut être considérée comme une application de l'ensemble des variables V dans celui des termes et des fbf. Ainsi soit l'ensemble E qui contient les termes ou fbf $A_1, A_2, \ldots, A_n$. La substitution $\sigma = (x_1/A_1, x_2/A_2, \ldots, x_n/A_n)$ permet d'avoir la fbf $E\sigma$ qui est le résultat de l'application σ à E et qui est obtenu en remplaçant chaque occurrence dans E de x_i par A_i , $i = 1, \ldots, n$. $E\sigma$ est appelé une *exemplification* (*instance*) de E.

DÉFINITION 6.2.2 *Soient* $\sigma = (x_1/A_1, x_2/A_2, \ldots, x_n/A_n)$ *et* $\tau = (y_1/B_1, y_2/B_2, \ldots, y_m/B_m)$ *deux substitutions. La composition* $\sigma\tau$ *de* σ *et* τ *est obtenue à partir de l'ensemble* $\{x_1/A_1\tau, x_2/A_2\tau, \ldots, x_n/A_n\tau, y_1/B_1, y_2/B_2, \ldots, y_m/B_m\}$ *en supprimant*

- *tous les* $x_i/A_i\tau$ *pour lesquels on a* $x_i = A_i\tau$, $i = 1, \ldots, n$
- *tous les* y_j/B_j *pour lesquels on a* $y_j \in \{x_1, x_2, \ldots, x_n\}$, $j = 1, \ldots, m$.

EXEMPLE 6.2.1 $\sigma = (x/Z, y/W)$, $\tau = (x/a, Z/b, W/y) \Rightarrow \sigma\tau = (x/b, Z/b, W/y)$

La composition des substitutions n'est pas en général commutative.

ASCÈSE 6.9 *Vérifier la non commutativité de la substitution en utilisant l'exemple ci–dessus.*

Les propriétés des substitution sont les suivantes :

PROPRIÉTÉ 6.2.1 $E(\sigma\tau) = (E\sigma)\tau$

PROPRIÉTÉ 6.2.2 $(\rho\sigma)\tau = \rho(\sigma\tau)$

PROPRIÉTÉ 6.2.3 $\varepsilon\theta = \theta\varepsilon = \theta$

PROPRIÉTÉ 6.2.4 *Si* $\models E$, *alors* $\models E\sigma$

6.3 Interprétation sémantique - Modèles

La première tâche du calcul des prédicats est de donner un sens aux fbf, c'est-à-dire d'établir la valeur de vérité de ces formules. Néanmoins il n'est pas possible à chaque fbf d'associer une valeur de vérité, car, pour une formule atomique, il faut en même temps donner une valeur aux arguments du prédicat et au prédicat lui-même. Or la valeur du prédicat dépend des valeurs de ses arguments. Pour cette raison nous avons besoin d'une fonction d'interprétation qui donnera la valeur de vérité au prédicat en fonction des valeurs de ses arguments.

Nous donnons d'abord une présentation informelle de l'interprétation.

On considère un langage $\mathcal{L}$ du 1er ordre et un univers du discours $\mathcal{U}$. Nous allons essayer de mettre en association les éléments du langage d'une part et les éléments de l'univers du discours, d'autre part. Cette correspondance entre éléments du langage et éléments de l'univers du discours constitue ce qu'on appelle une *interprétation* I du langage ou, encore, *structure*. Nous pouvons envisager que cette interprétation I ne couvre pas la totalité de l'univers du discours $\mathcal{U}$ mais une partie seulement, notée $\mathbb{D}(I) \subset \mathcal{U}$ et qui est le *domaine* de l'interprétation.

La correspondance I que nous venons de présenter n'est pas une application au sens habituel du terme mais plutôt un ensemble d'applications, chacune de ses applications étant spécifique à un type d'éléments de $\mathcal{L}$.

Ainsi pour les constantes a de $\mathcal{L}$ il faut trouver des constantes $a_{I.}$ dans $\mathcal{U}$ pour les faire appliquer.

Pour un foncteur f de $\mathcal{L}$ il faut trouver une relation f_I dans $\mathcal{U}$ qui peut être utilisée comme une fonction à valeurs dans $\mathcal{U}$. Dans ce cas, si l'arité de f est n, alors l'interprétation de $f(t_1,\ldots,t_n)$, où t_i ce sont des termes, est $f_{I.}((t_1)_{I.},\ldots,(t_n)_I)$ avec $(t_i)_I$ interprétation du terme t_i.

Pour un prédicat p dans $\mathcal{L}$ il faut trouver une relation p_I dans $\mathcal{U}$ qui peut être utilisée comme une fonction ayant deux valeurs : 0 ou 1. Dans ce cas, si l'arité de p est n, alors l'interprétation de $p(t_1,\ldots,t_n)$, où t_i ce sont des termes, est $p_{I.}((t_1)_{I.},\ldots,(t_n)_I)$ avec $(t_i)_I$ interprétation du terme t_i.

Nous obtenons ainsi la définition suivante :

DÉFINITION 6.3.1 (INTERPRÉTATION) *Une* interprétation *I sur un langage $\mathcal{L}$ est un domaine non vide $\mathbb{D}(I)$ (qui parfois est noté $\mid I \mid$), appelé* domaine de l'interprétation, *et une application qui associe :*

- *chaque constante $a \in \Xi$ avec un élément $a_I \in \mathbb{D}(I)$;*
- *chaque foncteur $f \in \mathbf{F}$ d'arité n avec une fonction $f_I : \mathbb{D}(I)^n \to \mathbb{D}(I)$;*
- *chaque prédicat $p \in \mathbf{P}$ d'arité n avec une relation $p_I \subseteq \mathbb{D}(I)^n$.*

Remarquons que la dernière association peut se comprendre comme étant une application de $\mathbb{D}(I)^n$ dans l'ensemble $\{0,1\}$ (faux, vrai).

ASCÈSE 6.10 *Considérons un langage $\mathcal{L}$ et un prédicat unaire $p \in \mathcal{L}$. Supposons que le domaine de définition de ce prédicat est l'ensemble $D = \{1,2\}$. Établissez pour le prédicat p les quatre interprétations I_i ; $i = 1,\ldots,4$ possibles et distinctes en utilisant comme domaine d'interprétation l'ensemble D.*

Grâce à cette définition de l'interprétation, on peut attribuer des valeurs de vérité aux fbf. En effet la valeur de vérité d'une fbf sera définie en fonction des valeurs de vérité de ses composantes qui sont soit des fbf à leur tour, soit des termes. Il faut donc pouvoir établir l'interprétation des termes. Dans la mesure où les termes contiennent des variables, il faut pouvoir les associer au domaine. Nous voyons donc que l'interprétation des variables du langage $\mathcal{L}$ pose un problème. Car pour pouvoir interpréter une variable x il faudrait savoir quel objet désigne exactement x. Mais dans ce cas x ne serait plus une variable. On s'en sort de cette situation embarrassante par un artifice du type suivant : les variables du langage sont interprétées comme étant des éléments variables de l'univers du discours ! Et pour formaliser cette belle interprétation on utilise ce qu'on appelle l'*assignation* des variables par rapport à une interprétation, qui est une application $\overline{\varphi}_I : V \to \mathbb{D}(I)$ où V l'ensemble des variables du langage $\mathcal{L}$ c'est-à-dire formellement :

DÉFINITION 6.3.2 (SÉMANTIQUE DES VARIABLES) *On appelle assignation d'un ensemble des variables $W \subset V$ relativement à une interprétation I , une application $\overline{\varphi}_I : W \to \mathbb{D}(I)$.*

EXEMPLE 6.3.1 *Considérons un langage $\mathcal{L}$ avec une constante a, une variable x, un foncteur unaire f et un prédicat binaire p. On peut envisager la formule $p(f(a), a)$. La transformation de cette formule par l'interprétation I serait $p_I(f_I(a_I), a_I)$. Mais qu'en est-il de la transformation de la formule $p(f(x), a)$? Si on procède à l'assignation x_I de la variable x, où x_I une variable indiquant un élément quelconque de l'univers du discours, alors on peut écrire pour l'interprétation : $p_I(f_I(x_I), a_I)$. Bien sûr si on pose $x = a$, alors les deux interprétations sont identiques.*

L'assignation des variables nous permet maintenant de définir la signification d'un terme.

DÉFINITION 6.3.3 (SÉMANTIQUE DES TERMES) *La* signification φ_I *d'un ensemble des termes $\mathbb{T}$ relativement à une interprétation I , est définie comme suit :*

- *si $t \in \mathbb{T}$ est une constante, alors $\varphi_I(t) = t_I$;*
- *si $t \in \mathbb{T}$ est une variable, alors $\varphi_I(t) = \overline{\varphi}_I(t)$;*
- *si $t \in \mathbb{T}$ est un terme de la forme $f(t_1, t_2, \ldots, t_n)$, alors $\varphi_I(t) = f_I(\varphi_I(t_1), \varphi_I(t_2), \ldots, \varphi_I(t_n))$.*

Notons que la terminologie n'est pas stabilisée en français. Certains auteurs utilisent le terme *assignation* pour signification, tandis que d'autres auteurs préfèrent parler d'*interprétation*. Il est curieux de constater que personne n'utilise la traduction du terme anglais *meaning* (= signification) qui est, à mon avis, très éclairante ici.

ASCÈSE 6.11 *Considérons un ensemble des termes $\mathbb{T}$ qui contient la constante zéro (l'élément neutre de l'addition), le foncteur unaire s (l'élément successeur) et le foncteur binaire plus (l'addition de deux valeurs). On cherche à établir une signification relativement à une interprétation I dont son domaine $\mathbb{D}(I)$ est l'ensemble des entiers non négatifs muni de l'opération de l'addition $+$.*

(1) Quelle doit-être, selon vous, l'interprétation des termes zéro , s et plus ?

(2) Calculer la signification du terme $plus(s(zero), X)$ où X une variable avec assignation

$$\overline{\varphi}_I(X) = \begin{cases} 0 & si\ X \neq\ nombre\ entier \\ \\ |\,X\,| & si\ X =\ nombre\ entier \end{cases}$$

On peut maintenant, grâce aux deux définitions précédentes, calculer la valeur de vérité d'une fbf. Formellement nous avons la définition suivante :

DÉFINITION 6.3.4 (SÉMANTIQUE DES FBF) *La* valeur de vérité de la signification d'une fbf F relativement à une interprétation I *(ou, de façon plus succincte, la valeur de vérité de F relativement à I), est définie comme suit :*

- *Si la fbf est de la forme $F = p(t_1, t_2, \ldots, t_n)$, alors $I(F)$ est égale à la valeur de vérité de $p_I(\varphi_I(t_1), \varphi_I(t_2), \ldots, \varphi_I(t_n))$.*
- *Si la fbf F a une des formes $\neg G$, $G \vee H$, $G \wedge H$, $G \to H$, $G \leftrightarrow H$, alors $I(F)$ est égale à la valeur de vérité de la forme correspondante.*

- *Si la fbf F est de la forme $\forall x G(x,y,z,\ldots)$, alors $I(F)=1$ si $\forall\sigma=(x/a)$ avec $a\in\mathbb{D}(I)$ nous avons $I(G\sigma)=1$ (vraie). Sinon $I(F)=0$ (fausse).*
- *Si la fbf F est de la forme $\exists x G(x,y,z,\ldots)$, alors $I(F)=1$ si $\exists\sigma=(x/a)$ avec $a\in\mathbb{D}(I)$ nous avons $I(G\sigma)=1$ (vraie). Sinon $I(F)=0$ (fausse).*

EXEMPLE 6.3.2 *Si $\mathbb{D}(I)$ est l'ensemble des nombres réels avec la relation d'égalité "=" qui correspond au prèdicat* eg *du langage $\mathcal{L}_1$, alors la formule atomique $F=\text{eg}(a,b)$ a comme valeur de vérité $I(F)$, la valeur de vérité de la relation $a=b$.*

De cette définition on déduit que la valeur de vérité d'une fbf close, par rapport à une interprétation, dépend seulement de cette interprétation et elle est indépendante de l'assignation des variables. Si nous avons une fbf avec des variables libres, alors on utilise la clôture universelle qui nous permet d'obtenir une fbf close et on applique ensuite la définition précédente. L'utilisation de la clôture universelle est tout à fait concevable car, comme nous venons de le dire, l'assignation des variables n'intervient pas à la détermination de la valeur de vérité d'une fbf.

ASCÈSE 6.12 *Considérons l'ascèse 6.10. Donner, pour une interprétation, la valeur de vérité de fbf*

$$p(x)\vee\forall y(p(y)\rightarrow q)$$

6.3.1 Satisfiabilité et modèles

Nous donnons dans la suite la définition d'une fbf satisfiable.

DÉFINITION 6.3.5 *Une fbf F est* satisfiable *ou* sémantiquement consistante *s'il existe une interprétation I telle que la valeur de vérité de F par rapport I est ègale à* 1. *L'interprétation est alors un* modèle *de F et l'on note par $I\models F$.*
Une fbf qui ne possède pas de modèle est appelée sémantiquement inconsistante *ou* insatisfiable.

ASCÈSE 6.13 *En reprenant l'ascèse 6.10, trouver un modèle pour la fbf*

$$p(x)\vee\forall y(p(y)\rightarrow q)$$

Nous arrivons ainsi à la notion de la fbf valide.

DÉFINITION 6.3.6 *Une fbf F qui est vraie pour toute interprétation est appelée* formule valide *et sera notée par $\models F$.*

ASCÈSE 6.14 *Vérifier si la fbf*

$$p(x)\vee\forall y(p(y)\rightarrow q)$$

est une formule valide.

Nous allons maintenant examiner brièvement la nature de deux quantificateurs $\forall$ et $\exists$. Considérons un prédicat p quelconque, que nous prendrons pour la circonstance et sans perte de généralité, unaire. La fbf $\forall x p(x)$ a comme valeur de vérité pour toute interprétation d'un domaine quelconque $\mathbb{D}(I)$ la valeur $\min\{p_I/x_I\in\mathbb{D}(I)\}$, c'est-à-dire la valuation d'une fbf universellement quantifiée sur la variable x est la valuation minimale de cette formule pour toutes

les interprétations x_I de x appliquées à l'interprétation p_I du prédicat p. Dans la mesure où $\min\{p_I/x_I \in \mathbb{D}(I)\} = \bigwedge\{p_I/x_I \in \mathbb{D}(I)\}$ on peut dire que le quantificateur universel $\forall$ est une généralisation du connecteur $\bigwedge$. De même la valuation de la fbf $\exists x p(x)$est donnée par la valeur $\max\{p_I/x_I \in \mathbb{D}(I)\} = \bigvee\{p_I/x_I \in \mathbb{D}(I)\}$, on peut dire que le quantificateur existentiel $\exists$ est une généralisation du connecteur $\bigvee$.

Examinons maintenant la notion de la conséquence sémantique.

DÉFINITION 6.3.7 *Soit la fbf close B et un ensemble des fbf closes $A = \{A_1, A_2, \ldots, A_n\}$. B est une* conséquence sémantique *des $A_1, A_2, \ldots, A_n$, que l'on note par $A \models B$ ou $A_1, A_2, \ldots, A_n \models B$, si pour toute interprétation I telle que $I(A_i) = 1 \forall i = 1, \ldots, n$, on a $I(B) = 1$.*

En d'autres termes, on peut dire que B doit être vérifiée pour tout modèle de A.

Il faut prendre dorénavant l'habitude d'interpréter un ensemble $\{A_1, A_2, \ldots, A_n\}$ de fbf comme étant une conjonction $A_1 \wedge A_2 \wedge \ldots \wedge A_n$.

ASCÈSE 6.15 *Considérons les deux fbf closes*

$$A = \{\forall X (\forall Y ((p(X) \wedge q(Y,X)) \rightarrow r(X,Y))), p(toto) \wedge q(koko, toto)\}$$

Montrer que la fbf close

$$r(toto, koko)$$

est une conséquence sémantique de A. (Démonstration avec l'utilisation du modus ponens, cf. infra).

En général il est difficile de vérifier si une fbf close B est une conséquence logique d'un ensemble de fbf closes A, car il faut vérifier B pour tout modèle de A. Une autre façon de démontrer que $A \models B$ est de montrer que $\neg B$ est fausse pour tout modèle de A ou, ce qui revient au même, que l'ensemble de fbf $A \cup \{\neg B\}$ est insatisfiable, c'est-à-dire n'a pas de modèle. Ce procédé est plus facile parce qu'il suffit de trouver un modèle de A qui n'est pas un modèle pour B. Formellement nous avons :

THÉORÈME 6.3.1 (DE L'INSATISFIABILITÉ) *Soient A ensemble de fbf closes et B fbf close. Alors $A \models B$ si et seulement si $A \cup \{\neg B\}$ est insatisfiable.*

Une notion importante pour la sémantique des fbf est celle de l'équivalence :

DÉFINITION 6.3.8 *Deux fbf F et G sont* logiquement équivalentes *si et seulement si elles ont la même valeur de vérité pour toute interprétation I.*

EXEMPLE 6.3.3 *Les formules suivantes sont logiquement équivalentes :*

- $\neg\neg A \equiv A$
- $A \rightarrow B \equiv \neg A \vee B$
- $A \rightarrow B \equiv \neg B \rightarrow \neg A$
- $A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$
- *Lois de de Morgan*
  - $\neg(A \vee B) \equiv \neg A \wedge \neg B$
  - $\neg(A \wedge B) \equiv \neg A \vee \neg B$

6.3.2 Examen des quantificateurs

Considérons un prédicat p quelconque, que nous prendrons pour la circonstance et sans perte de généralité, d'arité un. La fbf $\forall x p(x)$ a comme valeur de vérité pour toute interprétation d'un domaine quelconque $\mathbb{D}(I)$ la valeur $\min\{p_I/x_I \in \mathbb{D}(I)\}$, c'est-à-dire la valuation d'une fbf universellement quantifiée sur la variable x est la valuation minimale de cette formule pour toutes les interprétations x_I de x appliquées à l'interprétation p_I du prédicat p. Dans la mesure où $\min\{p_I/x_I \in \mathbb{D}(I)\} = \bigwedge\{p_I/x_I \in \mathbb{D}(I)\}$ on peut dire que le quantificateur universel $\forall$ est une généralisation du connecteur $\bigwedge$. De même la valuation de la fbf $\exists x p(x)$est donnée par la valeur $\max\{p_I/x_I \in \mathbb{D}(I)\} = \bigvee\{p_I/x_I \in \mathbb{D}(I)\}$, on peut dire que le quantificateur existentiel $\exists$ est une généralisation du connecteur $\bigvee$.

(1) $\neg\forall x A(x) \equiv \exists x \neg A(x)$

(2) $\forall x A(x) \equiv \neg\exists x \neg A(x)$

(3) $\neg\exists x A(x) \equiv \forall x \neg A(x)$

(4) $\exists x A(x) \equiv \neg\forall x \neg A(x)$

(5) $\forall x(A \vee B(x)) \equiv A \vee \forall x B(x)$ s'il n'y a pas d'occurrences de x dans A.

Le théorème suivant fournit des formules supplémentaires.

THÉORÈME 6.3.2 *Pour la distributivité des quantificateurs nous avons les relations suivantes :*

(1) $\models \forall x(\varphi \wedge \psi) \leftrightarrow \forall x\varphi \wedge \forall x\psi$

(2) $\models \exists x(\varphi \vee \psi) \leftrightarrow \exists x\varphi \vee \exists x\psi$

(3) $\models \forall x(\varphi(x) \vee \psi) \leftrightarrow \forall x\varphi(x) \vee \psi$ *si x n'est pas une variable libre de* ψ.

(4) $\models \exists x(\varphi(x) \wedge \psi) \leftrightarrow \exists x\varphi(x) \wedge \psi$ *si x n'est pas une variable libre de* ψ.

Attention : Les formules suivantes :
– $\forall x(\varphi(x) \vee \psi(x)) \rightarrow \forall x\varphi(x) \vee \forall x\psi(x)$
– $\exists x\varphi(x) \wedge \exists x\psi(x) \rightarrow \exists x(\varphi(x) \wedge \psi(x))$
ne sont pas vraies.

6.4 Évaluation syntaxique - Démonstration

L'évaluation syntaxique est un procédé mécanique qui permet de déduire le bien fondé d'une formule indépendamment du sens de ses composantes. Ainsi considérée, l'évaluation syntaxique est une théorie de la démonstration. Comme toute démarche déductive, la démonstartion en logique est fondée sir des axiomes qui sont les principes fondamentaux de la logique. La première notion de la théorie de démonstration est le théorème.

DÉFINITION 6.4.1 *Une fbf A est un* théorème, *et l'on note* $\vdash A$ *, si A est un axiome ou si A est une formule obtenue par application des règles d'inférence sur d'autres théorèmes.*

Notons que les règles d'inférence sont celles utilisées par le mécanisme de démonstration en logique propositionnelle, plus des règles spécifiques aux quantificateurs. Pour le calcul des prédicats nous avons donc comme règles d'inférence les suivantes :

R1.- Modus ponens (règle du détachement)

$$\frac{\vdash A, \vdash A \rightarrow B}{\vdash B}$$

R2.- Modus tollens (l'inverse de modus ponens)

$$\frac{\vdash A \rightarrow B, \vdash \neg B}{\vdash \neg A}$$

R3.- Élimination de « ET »

$$\frac{\vdash A \wedge B}{\vdash A, \vdash B}$$

R4.- Introduction de « ET »

$$\frac{\vdash A, \vdash B}{\vdash A \wedge B}$$

R5.- Généralisation

$$\frac{\vdash A}{\vdash \forall x A}$$

R6.- Exemplification universelle (instanciation universelle)

$$\frac{\vdash \forall x A}{\vdash A\sigma}, \ \sigma = (x/a)$$

R7.- Exemplification existentielle (instanciation existentielle)

$$\frac{\vdash \exists x A}{\vdash A\sigma}, \ \sigma = (x/s(x))$$

où $s(\cdot)$ est une constante qui est le résultat de l'application d'une fonction s à x.

Le calcul des prédicats possède les trois axiomes du calcul propositionnel, à savoir :

A1.- Introduction de l'implication

$$A \rightarrow (B \rightarrow A)$$

A2.- Distributivité de l'implication

$$(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$$

A3.- Réalisation de contradiction

$$(\neg B \rightarrow \neg A) \rightarrow (A \rightarrow B)$$

où, encore

$$(\neg B \rightarrow \neg A) \rightarrow ((\neg B \rightarrow A) \rightarrow B)$$

plus deux axiomes supplémentaires :

A4.- Exemplification (instanciation) universelle

$$\forall x A(x) \rightarrow A(c)$$

A5.- Généralisation universelle

$$((A \rightarrow B) \rightarrow (A \rightarrow \forall x B))$$

Nous sommes maintenant en mesure de définir la démonstration et la déduction.

DÉFINITION 6.4.2 *Soit un théorème A. Une* démonstration *de A est une suite finie* $(A_1, A_2, \ldots, A_n, A)$ *où chaque* A_i *est soit un axiome, soit le résultat d'une règle d'inférence appliquée sur des éléments* A_j *précédemment obtenus.*

DÉFINITION 6.4.3 *Une fbf close A est une* déduction *de l'ensemble de fbf closes* $B_1, B_2, \ldots, B_n$*, que l'on note* $B_1, B_2, \ldots, B_n \vdash A$ *s'il existe une suite finie* $(A_1, A_2, \ldots, A_n, A)$ *où chaque* A_i *est soit un axiome, soit un des* B_i *soit il est obtenu par application d'une règle d'inférence sur des éléments* A_j *précédemment obtenus.*

Les fbf B_i sont appelées des *hypothèses*.

THÉORÈME 6.4.1 (DE LA DÉDUCTION) *Soit A une fbf close. Si* $A \vdash B$*, alors* $\vdash A \rightarrow B$ *et vice versa.*

On introduit maintenant la notion de la théorie.

DÉFINITION 6.4.4 *Une* théorie $\mathcal{T}$ *est une collection des théorèmes avec la propriété* $\mathcal{T} \vdash p \rightarrow p \in \mathcal{T}$.

Deux théories peuvent être en relations selon la définition suivante.

DÉFINITION 6.4.5 *Soient* $\mathcal{T}$ *et* $\mathcal{T}'$ *deux théories sur les langages* $\mathcal{L}$ *rt* $\mathcal{L}'$ *respectivement.*

- $\mathcal{T}'$ *est une* extension *de* $\mathcal{T}$ *si* $\mathcal{T} \subseteq \mathcal{T}'$
- $\mathcal{T}'$ *est une* extension conservative *de* $\mathcal{T}$ *si* $\mathcal{T}' \cap \mathcal{L} = \mathcal{T}$*, i.e. tous les théorèmes de* $\mathcal{T}'$ *dans le language* $\mathcal{L}$ *sont aussi des théorèmes de* $\mathcal{T}$.

Nous terminons ce paragraphe par la notion de la consistance.

DÉFINITION 6.4.6 *Une logique est* syntaxiquement consistante *s'il n'existe aucune formule du langage telle que nous avons en même temps* $\vdash A$ *et* $\neg \vdash A$.

Nous avons le

THÉORÈME 6.4.2 *Le calcul des prédicats est syntaxiquement consistant.*

6.5 Équivalence entre modèles et théorie de démonstration

Nous allons voir que les modèles et la théorie de démonstration sont équivalentes en calcul des prédicats comme c'était déjà le cas en calcul propositionnel. Cette équivalence s'établit à l'aide de deux notions : adéquation et complétude.

DÉFINITION 6.5.1 *Une logique est* adéquate *si tout théorème $\vdash A$ est une formule valide $\models A$.*
Une logique est complète *si toute formule valide est un théorème, i.e.* $(\models A) \rightarrow (\vdash A)$.

L'équivalence cherchée est obtenue à l'aide des trois théorèmes suivants :

THÉORÈME 6.5.1 *Le calcul des prédicats est adéquat, i.e.*$(\vdash A) \rightarrow (\models A)$.

THÉORÈME 6.5.2 *Le calcul des prédicats est (fortement) complet, i.e.*
$(\models A) \rightarrow (\vdash A)$.

THÉORÈME 6.5.3 (de complétude) $A \vdash p \leftrightarrow A \models p$, *pour tout fbf p avec* $p \in \mathcal{L}$.

6.6 Quelques méta-théorèmes

Dans les paragraphes précédents nous avons introduit un certain nombre de concepts concernant les fbf prises individuellement. Nous allons étendre ces notions à un ensemble E des fbf.

THÉORÈME 6.6.1 (DE LA RÉFUTATION) *Une fbf A est* conséquence valide *d'un ensemble E de fbf, i.e.* $E \vdash A$, *si et seulement si* $E \cup \{\neg A\}$ *est insatisfiable.*
Une autre forme équivalente (et utile) de ce théoreme est la suivante : Si $E \cup \{A\}$ *est inconsistante, alors* $E \vdash \neg A$.

Notons qu'il faut bien interpréter le symbole $E \cup \{A\}$. *Si E est un ensemble des fbf* $f_1, \ldots, f_n$, $E = \{f_1, f_2 \ldots, f_n\}$, *alors* $E \cup \{A\}$ *signifie que nous avons* f_1 ET f_2 ET ... ET f_n ET A, *c'est-à-dire la virgule dans la notation de l'ensemble E joue le rôle du connecteur* $\bigwedge$.).

Si on note par $\perp$ la fbf qui est toujours fausse, on déduit qu'un ensemble de fbf est insatisfiable si et seulement s'il a comme conséquence la formule $\perp$. Il s'ensuit donc, que toute vérification de la validité d'un ensemble de fbf peut se réduire à la preuve de son inconsistance sémantique.

Cette remarque permet d'établir une autre méthode pour la vérification d'une fbf en calcul des prédicats. Elle est fondée sur le fait que si on accepte la négation d'une fbf et on aboutit à une contradiction, alors la fbf originale est vérifiée. Notons qu'un ensemble de fbf E contient une contradiction si et seulement s'il existe une fbf A telle que nous avons à la fois $E \vdash A$ et $E \vdash \neg A$.

Nous donnons ci-après quelques métathéorèmes supplémentaires du calcul des prédicats.

THÉORÈME 6.6.2 (RÈGLE T) *Si* $E \vdash A_1, E \vdash A_2, \ldots, E \vdash A_n$ *et*
$\{A_1, A_2, \ldots, A_{n-1}\} \vdash B$, *alors* $E \vdash B$.

THÉORÈME 6.6.3 (DE CONTRAPOSITION) $E \cup \{A\} \vdash \neg B$ *si et seulement si* $E \cup \{B\} \vdash \neg A$.

THÉORÈME 6.6.4 (DE LA GÉNÉRALISATION) *Si* $E \vdash B$ *et x est une variable qui n'a pas d'occurrences libres dans E, alors* $E \vdash \forall x B$.

6.7 Formes clausales

Nous avons déjà introduit au chapitre précédent la notion de la clause. Quand on passe au calcul des prédicats il est possible qu'une clause contient des quantificateurs. Afin d'obtenir une normalisation des différents types des clauses nous introduisons la notion de la clause sous forme prénexe.

DÉFINITION 6.7.1 *Une* clause sous forme prénexe *est une clause de la forme* $\Diamond x_1, \Diamond x_2, \ldots \Diamond x_n C$, *où* $\Diamond$ *désigne un quantificateur et C est une fbf sans quantificateurs.*
La clause vide sera notée par $\perp$. *Il s'agit d'une fbf toujours fausse.*

Nous avons le théorème suivant :

THÉORÈME 6.7.1 *Toute fbf du calcul des prédicats admet une forme prénexe équivalente*

La démonstration de ce théorème peut se faire, de manière constructive, par l'algorithme suivant de la transformation d'une fbf en une forme prénexe équivalente :

(1) Élimination du connecteur $\leftrightarrow$: $(A \leftrightarrow B) \equiv (A \rightarrow B) \wedge (B \rightarrow A)$

(2) Élimination du connecteur $\rightarrow$: $(A \rightarrow B) \equiv (\neg A \vee B)$

(3) Application des substitutions d'une variable par une autre de sorte que chaque variable apparaît sous la portée d'un seul quantificateur.

(4) Suppression des quantificateurs non opérationnels, i.e. dont la variable quantifiée n'apparaît pas sous leur portée.

(5) Transfert de la négation au niveau le plus intérieur (i.e. devant les atomes) par utilisation des règles :
   - des lois de de Morgan $\neg(A \wedge B) \equiv \neg A \vee \neg B$, $\neg(A \vee B) \equiv \neg A \wedge \neg B$ en tant que règles de réécriture.
   - de la loi de l'involution $\neg\neg A \equiv A$ qui permet la suppression des doubles négations.
   - $\neg\forall x A \equiv \exists x \neg A$
   - $\neg\exists x A \equiv \forall x \neg A$

(6) Transfert des quantificateurs au début de la fbf en utilisant les règles :
   - $(\forall x A \wedge B) \equiv \forall x (A \wedge B)$ si B ne contient pas x.
   - $(\exists x A \wedge B) \equiv \exists x (A \wedge B)$ si B ne contient pas x.
   - $(\forall x A \vee B) \equiv \forall x (A \vee B)$ si B ne contient pas x.
   - $(\exists x A \vee B) \equiv \exists x (A \vee B)$ si B ne contient pas x.

ASCÈSE 6.16 *Calculer la forme prénexe équivalente de la fbf suivante :*

$$\forall x A(x) \wedge \exists y B(y) \rightarrow \exists y (A(y) \wedge B(y))$$

Dans une forme prénexe la présence des quantificateurs universels n'est pas significative. En effet on considère que "$\forall x\, A(x)$ *est vraie*" et "$A(x)$ *est vraie*" sont deux formules par convention équivalentes (En fait la seconde est une exemplification – instance – de la première). Par contre la présence d'un quantificateur existentiel pour une variable d'une fbf, pose le problème de la « construction » d'une telle variable.

Par exemple si nous avons $\exists x\, A(x)$, on doit pouvoir construire une valeur c pour la variable x qui permet que la fbf $A(c)$ soit vraie. Dans ce cas on dit que nous avons remplacé la variable x par une fonction s d'arité 0, c'est-à-dire par une constante. Une autre possibilité est d'avoir la variable d'un quantificateur existentielle sous la portée d'un (ou plusieurs) quantificateur(s) universel(s) relatif(s) à d'autre(s) variable(s). Ainsi considérons la fbf $\forall x \exists y\, A(y,x)$. Dans cette formule pour chaque x on postule l'existence d'un y tel que $A(y,x)$ soit vérifiée. Pour résoudre ce problème on doit avoir une fonction $s(\cdot)$ d'arité 1, qui permet pour chaque x donné, de fabriquer un y. La fbf devient ainsi $\forall x\, A(s(x),x)$.

Bien évidemment on ne cherche pas à construire cette fonction s ni, a fortiori, à lui donner une interprétation. On se contente de lui donner un nom. Une telle fonction s'appelle fonction de Skolem. Nous avons :

DÉFINITION 6.7.2 *Une* fonction de Skolem *est une fonction qui, dans une fbf close, remplace la variable d'un quantificateur existentiel et prend comme arguments les variables des quantificateurs universels sous la portée desquels était la variable du quantificateur existentiel.*
Si la variable qui est replacée par une fonction de Skolem n'était pas sous la portée d'un quantificateur universel, alors l'arité de la fonction de Skolem serait nulle, c'est-à-dire la fonction de Skolem serait une constante.

La formule qu'on obtient si on remplace une variable sous un quantificateur existentiel par une fonction de Skolem n'est pas logiquement équivalente avec la formule initiale. Seules les consistances de deux formules sont identiques, ce qui est suffisant quand, en utilisant le théorème de réfutation, on procède par preuve de l'inconsistance.

Ainsi la forme de Skolem d'une fbf close n'a que des quantificateurs universels. Ces quantificateurs peuvent être supprimés. On obtient ainsi une *exemplification* (ou une *instance*) de la forme de Skolem.

DÉFINITION 6.7.3 *Une conjonction de clauses $C_1 \wedge C_2 \wedge \cdots \wedge C_n$ du calcul des prédicats dans lesquels*
- *les variables sous des quantificateurs existentiels sont remplacées par des fonctions de Skolem, et*
- *les variables sous des quantificateurs universels sont omises*

est une forme conjonctive normale (fcn) *ou* forme standard.
Une fcn est parfois notée sous forme ensembliste $\{C_1, C_2, \ldots, C_n\}$.

Soit une fbf close F. L'algorithme pour aboutir à une forme standard est le suivant :

(1) Élimination du connecteur $\leftrightarrow$: $(A \leftrightarrow B) \equiv (A \rightarrow B) \wedge (B \rightarrow A)$

(2) Élimination du connecteur $\rightarrow$: $(A \rightarrow B) \equiv (\neg A \vee B)$

(3) Transfert de la négation au niveau le plus intérieur (i.e. devant les atomes) par utilisation des règles :
- des lois de de Morgan $\neg(A \wedge B) \equiv \neg A \vee \neg B$, $\neg(A \vee B) \equiv \neg A \wedge \neg B$ en tant que règles de reécriture.
- de la loi de l'involution $\neg\neg A \equiv A$ qui permet la suppression des doubles négations.
- $\neg \forall x A \equiv \exists x \neg A$
- $\neg \exists x A \equiv \forall x \neg A$

(4) Application des substitutions d'une variable par une autre de sorte que chaque variable apparaî t sous la portée d'un seul quantificateur.

(5) Suppression des quantificateurs non opérationnels, i.e. dont la variable quantifiée n'apparaî t pas sous leur portée.

(6) Suppression des quantificateurs existentiels par application de la fonction de Skolem.

(7) Conversion en forme prénexe, i.e. transfert des quantificateurs au début de la fbf en utilisant les règles :
- $(\forall xA \wedge B) \equiv \forall x(A \wedge B)$ si B ne contient pas x.
- $(\exists xA \wedge B) \equiv \exists x(A \wedge B)$ si B ne contient pas x.
- $(\forall xA \vee B) \equiv \forall x(A \vee B)$ si B ne contient pas x.
- $(\exists xA \vee B) \equiv \exists x(A \vee B)$ si B ne contient pas x.

(8) Suppression des quantificateurs universels.

(9) Conversion en forme standard par application de la distributivité
- de $\vee$ par rapport à $\wedge$: $A \vee (B \wedge C) = (A \vee B) \wedge (B \vee C)$
- de $\wedge$ par rapport à $\vee$: $A \wedge (B \vee C) = (A \wedge B) \vee (B \wedge C)$

en tant que règles de réécriture.

ASCÈSE 6.17 *Appliquer l'algorithme ci-dessus à la fbf*

$$\exists x(p(X) \wedge \forall y(p(y) \rightarrow \neg q(Y,X))) \wedge \neg\exists x(p(X) \wedge \forall y(p(y) \rightarrow \neg q(Y,X)))$$

De façon analogue nous pouvons définir la *forme disjonctive normale (fdn)* : il s'agit d'une disjonction de conjonctions des littéraux.

Un cas particulier de fdn est le problème SAT de satisfaction d'une fdn. Dans la littérature on trouve les problèmes k-SAT qui concernent la satisfiabilité des fdn dont chaque terme a au plus k littéraux. Il y a beaucoup de résultats pour $k = 2$, qui sont des problèmes de classe de complexité P et pour $k = 3$ qui sont des problèmes de classe de copmlexité NP-complet. Notons qu'un problème est de classe NP s'il peut être résolu par une machine de Turing non déterministe en temps polynômial, où NP signifie non déterministe polynômial.
Un problème Π est *NP-difficile* si pour tout autre problème Π' de NP on a un algorithme de complexité inférieure à celle de la classe NP qui transforme toute instance de Π' en une instance de Π.
Un problème est NP-complet s'il est dans NP et s'il est NP-difficile.. Une étude plus complète de ces problèmes se fait dans le cours de Décidabilité.

En utilisant les règles de de Morgan on constate que la négation d'une fcn est une fdn et vice-versa. Par conséquent ce qui vient d'être dit pour les problèmes k-SAT peut être transposé dans le cas des fcn. Ce qui nous intéresse ici est de savoir si un programme logique donné E permet d'induire une fbf A, c'est-à-dire si $E \vdash A$. Pour résoudre ce problème on transforme d'abord E et A sous forme conjonctive normale. Ensuite nous avons deux possibilités :

(1) Soit on construit le programme $E \cup \{A\}$ et on cherche à savoir s'il est valide, c'est-à-dire si toutes les interprétations de ce programme sont des modèles pour ce programme. Il s'agit, comme nous venons de voir, d'un problème NP-complet.

(2) Soit on construit le programme $E \cup \{\neg A\}$ et on cherche à savoir si $E \cup \{\neg A\} \vdash \bot$ ce qui est plus facile que le précédent car il suffit de trouver une interprétation qui rend fausse la fcn $E \cup \{\neg A\}$.
Il est à noter que Prolog fonctionne selon ce principe.

Il est utile de préciser ici que le calcul de la réponse d'un programme à une question est différent de la validité de ce même programme. Ce dernier problème est *NP*-complet et il n'y a pas actuellement une méthode de test de programme qui soit pleinement satisfaisante.

6.8 Exercices

EXERCICE 6.1 *Considérons l'univers du discours* $\{\bigcirc, *\}$ *avec*
- *les constantes* a, b
- *les variables* x, y, z
- *la relation binaire* f

Considérons une interprétation I *telle que*
- *l'interprétation des constantes est :* $a_I = \bigcirc, b_I = *$
- *l'assignation des variables est :* $\varphi_I(x) = \bigcirc, \varphi_I(y) = \bigcirc, \varphi_I(z) = *$
- *l'interprétation de la relation est :* $f_I = \{\langle \bigcirc, * \rangle, \langle *, * \rangle\}$

Calculer la valeur de vérité des termes suivants :

(1) $\neg f(b,a)$

(2) $f(a,b) \wedge f(b,a)$

(3) $f(a,b) \vee f(b,a)$

(4) $f(a,b) \rightarrow f(b,a)$

(5) $f(a,b) \leftrightarrow \neg f(b,a)$

EXERCICE 6.2 *Considérons l'ascèse 2.2 et supposons que l'ensemble de termes contient en plus le prédicat p avec interprétation*

$$p_I = \{1, 3, 5, \ldots\}$$

Calculer la valeur de vérité de la fbf

$$p(zéro) \wedge p(s(zéro))$$

EXERCICE 6.3 *Traduire enf fbf du calcul des prédicats les propositions suivantes :*

(1) Les points X, Y, Z sont sur la même droite.

(2) Les points X, Y, Z ne sont pas sur la même droite.

(3) les lignes l et l′ ont un point commun unique.

(4) Deux droites distinctes ont un point commun unique.

(5) l, l′ sont deux droites parallèles.

(6) (5e postulat d'Euclide.) Pour toute droite L et pour tout X il existe une droite unique qui passe par X est est parallèle à L.

EXERCICE 6.4 *Considérons deux formules closes F et G. Montrer que $F \equiv G$ si et seulement si $\{F\} \models G \wedge \{G\} \models F$.*

EXERCICE 6.5 *Soit $\mathcal{L}$ un langage du 1er ordre. Considérons la formule*

$$\forall x \exists y p(x,y) \rightarrow \exists y \forall x p(x,y)$$

Est-ce que cette formule peut être satisfaite pour n'importe quelle interprétation de $p(x,y)$?

EXERCICE 6.6 *Soit $\mathcal{L}$ un langage du 1er ordre constitué d'un prédicat unaire p et d'un prédicat binaire r.*

Nous considérons les formules suivantes :

(1) $\exists x \forall y \exists z ((p(x) \rightarrow r(x,y)) \wedge p(y) \wedge \neg r(x,y))$

(2) $\exists x \exists z (r(z,x) \rightarrow r(x,z) \rightarrow \forall y r(x,y))$

(3) $\forall y (\exists z \forall x r(x,z) \wedge \forall x (r(x,y) \rightarrow \neg r(x,y)))$

(4) $\exists x \forall y ((p(y) \rightarrow r(y,x)) \wedge (\forall t (p(t) \rightarrow r(t,y)) \rightarrow r(x,y))$

(5) $\forall x \forall y ((p(x) \wedge r(x,y)) \rightarrow ((p(x) \rightarrow \neg r(x,y)) \rightarrow \exists z (\neg r(z,x) \wedge \neg r(y,z))))$

(6) $\forall z \forall u \exists x \forall y ((r(x,y) \wedge p(u) \rightarrow (p(y) \rightarrow r(z,x)))$

Vérifier si ces formules sont satisfaites dans le modèle suivant : Le domaine est $\mathbb{N}$, l'interprétation de r est la relation d'ordre $\leq$ et celle de p est le sous-ensemble des naturels pairs.

EXERCICE 6.7 *La loi dit qu'il est interdit de posséder des armes à feu non enregistrées. Butch possède plusieurs armes à feu non enregistrées, achétées toutes chez Le Kid. Montrer que Le Kid est un hors-la-loi.*

7

LOGIQUE DES PRÉDICATS ET PROGRAMMATION

COMME en Logique Computationnelle, un programme en Prolog qui n'a pas des variables est un programme dont sa sémantique [1] se limite au programme lui-même. Il n'y a donc pas de nouvelles connaissances. Si par contre nous introduisons des variables, alors il est possible d'en déduire des connaissances nouvelles. Ce point peut être vérifier facilement en se reportant à l'exemple du graphe du chapitre précédent. Si notre programme se limite à la base de données qui représente le graphe, nos connaissances se limitent aux successeurs et prédécesseurs immédiats de chaque sommet. Si nous introduisons des variables qui permettent de définir des nouveaux prédicats, alors nous pouvons avoir des connaissances supplémentaires concernant par exemple les successeurs au sens large d'un sommet donné.

La programmation en Prolog consiste essentiellement à construire des prédicats qui permettront d'inférer des connaissances nouvelles à partir des bases de données existantes. On voit ainsi que la Logique Computationnelle est un outil pour ce que, en langage branché, on appelle *forage de données – data mining* et qui est en réalité une activité aussi vieille que le monde.

1. La sémantique d'un programme P sont toutes les formules atomiques closes qui peuvent être induites par P. En d'autre termes c'est toutes les connaissances que nous pouvons obtenir en utilisant les connaissances du programme P.

7.1 Les prédicats de base de Prolog

Pour construire nos propres prédicats nous pouvons utiliser des prédicats de Prolog. En effet SWI-Prolog a une impressionnante collection des prédicats dont vous trouverez la liste complète et leur utilisation dans le guide de référence du langage. Nous présentons dans ce paragraphe quelques prédicats très utiles avec une explication sommaire de leur signification.

Pour la présentation des prédicats, nous utiliserons les conventions suib=vantes :

- nomPredicat/n où n est l'arité du prédicat nomPredicat.
- nomPredicat(+Arg1, -Arg2, ?Arg3), où +Arg1 signifie que l'argument Arg1 est en entrée, -Arg2 signifie que l'argument Arg2 est en sortie et ?Arg3 signifie que l'argument Arg3 est en entrée-sortie.

7.1.1 Entrées-sorties

```
write/1 avec write(+Term)
Exemples :
write(X), write('toto')   afficher le contenu de X ou le mot << toto >>
print/1 avec print(+Term)
Exemples :
print(X), print('toto')   afficher le contenu de X ou le mot << toto >>
tab/1 avec tab(+
tab(+Quantit\'e)            affiche quantit\'e espaces blancs
nl/0                      saut d'une ligne
read(-X)                  lecture d'une valeur et stockage dans X
```

7.1.2 Opérations arithmétiques en Prolog

Voici une liste non exhaustive des opérateurs arithmétiques de Prolog

```
X = Y     X et Y sont les m\^emes nombres. Si une des variables n'est pas
          instaci\'ee, alors elle prend la valeur de l'autre par unification
          (et non par affectation). Remarquez que 4 = 1+3 \'echoue car 1+3
          est vu par Prolog comme un arbre +(1, 3) (faire display(4, 1+3).
          pour le voir).
X =:= Y   comme avec ''='' mais les variables doivent \^etre instanci\'ees
          au pr\'ealable. Si l'une de variables est instaci\'ee et l'autre non,
          l'op\'erateur \'echoue.)
X \= Y    X et Y sont des nombres diff\'erents. (Attention, si on pose X is 3,
          X\=Y., on obtient false.)
X =\= Y   X est non identique \'a Y.
X < Y
X =< Y
X >= Y
X > Y
X == Y    \'egalit\'e sans unification
X =/= Y   X est non identique \'a Y
```

Examinons à l'aide d'un exemple la différence entre les trois types d'égalité :

```
?- 3 = 1+2.
false.

X is 3, Y is 1+2, X=Y.
X = 3,
```

```
Y = 3.

?- X is 1+2, Y = X.
X = 3,
Y = 3.

?- X is 1+2, Y =:=X.
ERROR: =:=/2: Arguments are not sufficiently instantiated

?- X is 1+2, Y = 3, X =:=Y.
X = 3,
Y = 3.

?- X is 1+2, Y = 3, X =\=Y.
false.
```

Dans le premier cas, la représentation par Prolog de deux membres de l'égalité n'est pas identique et on donc échec. Dans le deuxième cas, le contenu de deux variables est identique et le test réussit. Dans le troisième cas, Y est instancié à la valeur de X et le test réussi. Notons que ce test réussira aussi si les deux variables ne sont pas instanciées. Dans le quatrième cas le contenu Y n'a pas de valeur et elle ne peut pas être instanciée avec X. Donc le test échoue. Dans le cinquième cas le contenu de deux variables est le même et le test réussit et par la suite le test du sixième cas échoue.

7.1.3 Fonctions arithmétiques

Il y a les quatre opérations arithmétiques – `+, -, *, /` – ainsi que les fonctions `sqrt, abs, sign, max, min`. Nous avons aussi les fonctions trigonométriques `sin, cos, tan asin, acos, atan` et le s fonctions logarithmiques `log10, log, exp`. Notons aussi les constantes `pi = 3.14159` et `e = 2.71828`.

Notons encore trois procédures Prolog, à savoir

- `between(+L1, +L2, ?Val)` qui réussit si `L1 =< L2`. `Val` est instaciée successivement aux valeurs `L1, ...L2`.
- `succ(?I1, ?I2)` qui réussit si `I2 = I1+1`. Au moins un argument doit être instacié.
- `plus(?I1, ?I2, ?I3)` si `I3 = I1 + I2`. Au moins deux arguments doivent être instaciés.

7.1.4 Opérateurs avec des termes non arithmétiques

Voici une liste non exhaustive d'opérateurs qui s'appliquent à des termes non arithmétiques.

```
:-          si
X = Y       X et Y sont le m\^eme contenu. Si une des variables n'est pas
            instaci\'ee, alors elle prend la valeur de l'autre par unification
            (et non par affectation)
X == Y      egalite sans unification
X \= Y      X et Y ont de valeurs diff\'erentes
X \== Y     X est non identique \'a Y

            En examinant le code ASCII du contenu de X et Y, on a
            les op\'erateurs suivants
X @< Y      X est plus petit que Y
X @=< Y     X est plus petit ou \'egal \'a Y
X @> Y      X est plus grand que Y
```

```
X @>= Y   X est plus grand ou \'egal \'a Y
X @== Y   X est structurellement identique \'a Y
X \=@= Y  X n'est pas structurellement identique \'a Y
```

L'équivalence structurelle `@==/2` est plus faible que l'équivalence `==/2` mais plus forte que l'unification `=/2` . Ainsi on a

```
a =@= A              faux
A =@= B              vrai
x(A,A) =@= x(B,C)    faux
x(A,A) =@= x(B,B)    vrai
x(A,B) =@= x(C,D)    vrai
```

7.1.4.1 Opérateurs établis par l'utilisateur

En dehors des opérateurs établis par `Prolog` le programmeur peut définir ses propres opérateurs en utilisant la requête « `:-` ». La forme générale d'un opérateur est

```
:- op(priorité, type, nom)
```

avec

- `Priorité` : une valeur entre 1 et 32000 qui indique la priorité de l'opérateur (1 est le plus prioritaire, 32000 le moins prioritaire).
- `Type` :
  - infixe : `xfx, xfy, yfx, yfy`
  - préfixe : `fx, fy`
  - postfixe : `xf, yf`
- `Nom` : le nom de l'opérateur.

En ce qui concerne le type de l'opérateur, le symbole « `x` » interdit les associations tandis que le symbole « `y` » les autorise. Ainsi les quatre opérations numériques sont du type `yfx` et par conséquent une expression comme

```
a + b + c + d
```

sera représentée en interne comme suit

```
(((a + b) + c) + d).
```

La virgule est un opérateur du type `xfy` et donc l'expression

```
a, b, c, d
```

sera interprétée comme suit :

```
(a, (b, (c, d)))
```

7.1.5 Prédicats extralogiques

Les prédicats extralogiques sont des prédicats qui soit testent le statut d'un terme, soit induisent une action.

Dans la première catégorie on trouve

```
var(+X)     X est une variable
nonvar(+X)  X n'est pas une variable
atom(+X)    X est une constante non num\'erique
atomic(+X)  X est une constante non num\'erique ou une valeur num\'erique
integer(+X) X est un entier
float(+X)   X ets un flottant
number(+X)  X est un nombre (entier ou flottant)
```

Pour `atom` nous avons les résultats suivants :

```
?- atom('toto').
true.

?- atom(toto).
true.

?- atom(3).
false.

?- atom(pi).
true.

?- X is pi, atom(X).
false.
```

Pour les nombres on a les résultats suivants :

```
atomic(pi).
true.

?- float(pi).
false.

?- X is pi, atomic(X).
X = 3.14159.

?- X is pi, float(X).
X = 3.14159.
```

Dans la deuxième catégorie on trouve d'une part

```
assert(X)   Ajouter \'a la base de donn\'ees le fait X
asserta(X)  Ajouter au debut de la base de donnees le fait X
assertz(X)  Ajouter \'a la fin de la base de donnees le fait X
retract(X)  Supprimer de la base de donn\'ees toutes les occurrences de X
```

et, d'autre part, le prédicat `fail` et le coup-choix (cut) `!` que nous examinerons en détail plus loin.

7.2 Un exemple

Considérons de nouveau l'exemple du graphe du chapitre précédent dont voici de nouveau sa représentation graphique :

Nous avons vu que la base de données du graphe permet de répondre à de questions du type : `?gr(a,b,_)`. Pour améliorer nos connaissances on doit utiliser la logique des prédicats qui, grâce à l'existence des quantificateurs, permet de poser des questions du type

`Existe-t-il des sommets X tels que gr(a,X,_) ?`

et qui sous forme clausale s'écrit

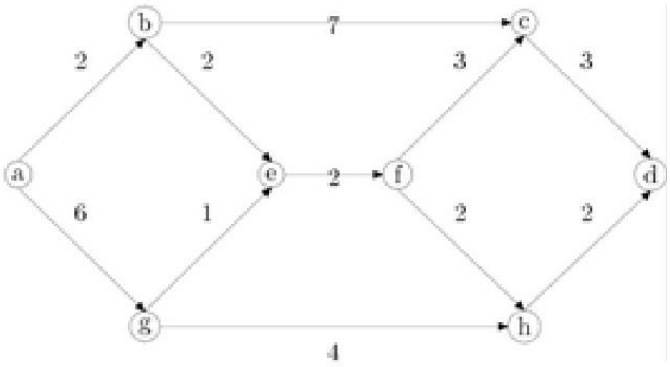

FIGURE 7.1 – Graphe pour un réseau

`?-gr(a,X,_).` [2]

Dans le cadre de la logique des prédicats, les clauses du programme sont considérées comme des prédicats et, par conséquent, leur valeur de vérité dépend de la valeur des arguments.

PROGRAMME 7.2.1

```
successeurImmediat(X,Y) :- gr(X,Y,_).
```

ASCÈSE 7.1 *Examiner le résultat des questions :*

`successeurImmediat(c,Y).`
`successeurImmediat(X,Y).`
`successeurImmediat(X,c).`

Pourriez-vous faire la liaison avec le modèle minimal de Herbrand ?

Pourriez-vous anticiper la réponse de `Prolog` *aux questions :*

`successeurImmediat(X,a).`
`successeurImmediat(d,Y).`

En fonction des résultats obtenus est-il possible d'établir deux nouveaux prédicats qui décrivent les propriétés des sommets `a` *et* `b` *?*

Nous pouvons maintenant envisager de définir la notion du successeur au sens large, à savoir un sommet qu'on peut atteindre d'un sommet fixé en passant par plusieurs sommets intermédiaires. Cette notion peut introduite au programme en y ajoutant les clauses suivantes :

PROGRAMME 7.2.2

```
successeur(X,Y) :- successeurImmediat(X,Y).
successeur(X,Y) :- successeurImmediat(X,Z), successeur(Z,Y).
```

Notons que ce programme est sous forme recursive parce que la deuxième clause contient un appel à cette même clause. Nous examinerons par la suite les méthodes d'écriture des programmes récursifs.

2. Remarquons que la clause écrite en `Prolog` respecte la convention qui veut que les noms des constantes commencent par une lettre minuscule – ici le sommet `a` – et les noms des variables par une lettre majuscule – ici `X` qui représente les sommets reliés directement à `a` et dont `a` est l'extrémité de départ. De plus on utilise aussi le symbole « _ » qui représente la variable anonyme, c'est-à-dire une variable dont la valeur précise ne nous intéresse pas. Pour plus de détails concernant la variable anonyme, voir dans la suite.

ASCÈSE 7.2 *Examiner le résultat des questions :*
`successeur(c,Y).`
`successeur(X,c).`
Pourriez-vous faire la liaison avec le modèle minimal de Herbrand ?
Établir l'arbre de dérivation SLD pour les questions précédentes.

Nous allons par la suite revenir plusieurs fois sur cet exemple du graphe.

7.3 Sémantique de Prolog

Considérons un programme défini E écrit en Prolog. Sa sémantique est l'ensemble des fbf closes que nous pouvons en déduire des clauses du programme E en appliquant les règles de la logique des prédicats. Ainsi une question – qui, en réalité, est une clause – que nous posons, constitue un but à vérifier par E et si ce but est dans la sémantique de E, alors la réponse du programme sera positive.

Remarquons que le comportement d'un programme Prolog depend de l'ordre de clauses – faits et règles – et de l'ordre des prédicats, à l'intérieur de chaque clause. Du fait que, en cherchant à répondre à une question, Prolog examine ces prédicats en les considérant comme des buts, dans la bibliographie l'ordre des prédicats porte le nom d'*ordre des buts*, nom que nous utiliserons par la suite.

7.3.1 Ordre des clauses

L'ordre dans lequel sont écrites les clauses d'un programme Prolog détermine l'ordre dans lequel seront trouvées les différentes solutions. En effet Prolog pour trouver toutes les réponses à un but, parcourt l'arbre de dérivation de la résolution SLD selon l'algorithme *en profondeur d'abord*. Chaque clause du programme est placée sur une branche de cet arbre en fonction de la place qu'elle occupe dans l'ordre des clauses du programme. Si donc nous avons deux programmes qui sont identiques mais dans lesquels l'ordre des clauses n'est pas le même, le parcours de l'arbre ne se fera pas de la même façon mais les deux graphes seront isomorphes et par conséquent si l'un de deux n'a pas une branche infinie, l'autre non plus n'a pas de branche infinie.

Il n'y a pas une règle générale pour l'ordre des clauses dans un programme Prolog. L'usage cependant veut que, dans le cas d'un programme récursif, on ait d'abord le(s) fait(s), c'est-à-dire le(s) test(s) d'arrêt, et ensuite les règles récursives.

ASCÈSE 7.3 *Vérifier l'ordre des solutions trouvées pour le but*
`?-successeur(f,X).`
si à la place du programme 1.3 on utilise le programme

PROGRAMME 7.3.1

```
successeur(X,Y) :- successeurImmediat(X,Z), successeur(Z,Y).
successeur(X,Y) :- successeurImmediat(X,Y).
```

Pourriez-vous expliquer les différences ?

7.3.2 Ordre des buts

L'ordre dans lequel sont présentés les premisses ou (sous-)buts dans une règle d'un programme `Prolog` conditionne l'exécution du programme et, donc, il détermine les solutions qui seront trouvées. Car, contrairement à la permutation des clauses qui aboutit à des arbres de dérivation toujours isomorphes entre eux, la permutation des buts donne naissance à des arbres de dérivation différents. De plus en fonction de la place qu'il occupe un appel récursif dans une règle, le programme peut nous fournir des solutions avant d'aboutir à une branche infinie ou même aboutir à cette branche infinie avant de donner la moindre solution.

ASCÈSE 7.4 *Vérifier les solutions trouvées pour le but*
`?-successeur(f,X).`
si à la place du programme 1.3 on utilise le programme

PROGRAMME 7.3.2

```
successeur(X,Y) :- successeurImmediat(X,Y).
successeur(X,Y) :- successeur(Z,Y), successeurImmediat(X,Z).
```

Pourriez-vous expliquer les différences ?
Pourriez-vous anticiper la réponse aux questions
`?-successeur(d,X).` *et*
`?-successeur(X,a).`

Comme précédemment avec les clauses, il n'y a pas non plus une méthode pour établir l'ordre des buts dans une règle. L'ascèse ci-dessus suggère d'utiliser un appel récursif à la fin des prémisses d'une règle (récursivité droite) plutôt qu'un appel récursif au début des prémisses (récursivité gauche) mais il ne s'agit pas d'une méthode générale.

ASCÈSE 7.5 *On dira que deux sommets sont au même niveau s'ils sont successeurs immédiats du même sommet.*

(1) *Écrire le programme* `memeNiveau(X,Y).`

(2) *Vérifier en posant la question* `?memeNiveau(b,X)` *que deux variables différentes n'ont pas nécessairement des valeurs différentes. Apporter une solution.*

(3) *Compléter ce programme pour tenir compte du fait que la relation* `memeNiveau(X,Y)` *est symétrique, c'est-à-dire que si, par exemple, on a* `memeNiveau(bmg)`*, alors on a aussi* `memeNiveau(g,b).`

8

LISTES ET RECURSIVITÉ

NOUS avons vu jusqu'ici le stockage des données à l'aide des base de données. Mais le traitement des données à partir de ces bases n'est pas toujours très facile à effectuer. On a envie d'avoir des données en mémoire dynamique facilement manipulables. À vrai dire `Prolog` est très chichement doté en types des données. Comme son grand ancêtre, le `Lisp`, `Prolog` ne dispose comme type des données, en tout et pour tout, que les listes. Bien sûr tout programmeur experimenté sait que la profusion des types de données qui sont l'apanage des plusieurs langages de programmation, en commençant par le plus illustre, le `Fortran`, sont très souvent source de confusions. Il est donc important pour l'élève–ingénieur de comprendre que l'esprit humain doit dompter la machine, qui par construction et par essence est bête, et arriver à faire des choses merveilleuses en utilisant très peu des matériaux. `Prolog` constitue un excellent exercice pour cet objectif.

8.1 Les listes et leur représentation

La seule structure des données que `Prolog` reconnaît ce sont les *listes*. Ce qui veut dire qu'en `Prolog` il n'y a pas des tableaux et surtout il n'y a pas des indices de tableau ou des pointeurs.

Une liste est une suite des termes de n'importe quelle nature, separés par des virgules, entourée par deux crochets, [et] . Par exemple `[toto, 1, av_du_parc, cergy, la_logique_est_super]`. Bien évidemment une liste peut avoir des sous-listes, une sous-liste peut avoir des sous-sous-listes et ainsi de suite à la manière des poupées russes mais généralisées en ce sens qu'une pou-

pée peut contenir plusieurs sous-poupées de même taille et qui, à leur tour, puissent avoir plusieurs sous-sous-poupées de même taille. Par exemple `[toto,[ 1,[ av_du_parc], [cergy]], la_logique_est_super]`.

Il faut peut être le repéter : on ne peut pas accéder directement à un élément quelconque d'une liste. (Par contre on peut accéder à un élément dont on connaît effectivement son rang dans la liste.) On peut seulement séparer ce qu'on appelle la *tête* d'une liste du *reste* de la liste, en utilisant le symbole du séparateur « | ». Ainsi, si une liste est représentée par la variable `L`, on peut écrire `L=[Tete | Reste]` où `Tete` représente le premier élément de `L` et `Reste` est la liste composée des autres élément de `L`. Par exemple si
`L=[toto, 1, av_du_parc, cergy, la_logique_est_super],`
alors on a `Tete = toto`
et `Reste = [1, av_du_parc, cergy, la_logique_est_super]`.

De ce qui précède on peut en conclure qu'on peut accéder au premier élément d'une liste. Considérons maintenant une liste ayant n éléments `L=[`$x_1,x_2,...,x_n$`]`. Si on veut accéder au k-ième terme, où k a une valeur précise et connue, nous avons deux possibilités :

- soit directement en posant pour `L` : `[ T`$_1$`,T`$_2$`,...,T`$_k$ `| Reste ]` avec `T`$_k$ $\longleftarrow x_k$.
- soit d'une façon séquentielle, à la manière de la lecture des enregistrements d'un fichier en accés séquentiel.On construit la représentation `L`$_1$ `= [Tete | Reste]`, où `Tete` $\longleftarrow x_1$. On recupère le `Reste` dans une liste notée L_2 et on recommence : L_2 =`[Tete | Reste]` avec maintenant `Tete` $\longleftarrow x_2$. En continuant ainsi on arrive, au bout de k itérations, à accéder au k-ième élément de la liste.

Même si la première possibilité vous paraît plus facile, rappelez-vous ce qu'on vous a toujours dit concernant les apparences et concentrez-vous sur la deuxième possibilité. C'est celle qui est utilisée en `Prolog` mais dans sa version recursive.

8.2 La recursivité

Pour appliquer la recursivité sur les listes il faut savoir que si on introduit une liste, e.g. `[toto, 1, av_du_parc, cergy, la_logique_est_super]`, `Prolog` introduit toujours à la fin de la liste, comme un élément supplémentaire, une liste vide, de sorte qu'on ait `[toto,1,av_du_parc,cergy, la_logique_est_super, []]`. Ainsi quand on progresse à l'intérieur d'une liste, élément par élément, on peut comprendre qu'on est arrivé à la fin de la liste en comparant la liste qui reste chaque fois avec la liste vide. Le test de la liste vide constituera pour beaucoup de programmes recursifs, le test d'arrêt.

En règle générale, un programme récursif est composé de deux parties :

- Une partie concernant le ou les tests d'arrêt.
- Une partie concernant les appels récursifs du prédicat à lui-même.

La programmation recursive est un type particulier de programmation au même titre que la programmation fonctionnelle ou la programmation orienté objet. Il faut savoir que la mise en øeuvre d'un programme est plus facile qu'avec les autres types de programmation et ses résultats sont beaucoup beaucoup plus spectaculaires parce qu'on utilise la puissance de la recursivité. En effet les programmes en `Prolog` expriment seulement ce que le programmeur souhaite réaliser et non pas la manière de faire pour le réaliser comme c'est le cas avec les langages imératifs.

Nous allons examiner en detail le mécanisme des appels récursifs en utilisant la liste `L=[toto, 1, av_du_parc, cergy, la_logique_est_super]`. On cherche à calculer la longueur de cette liste, c'est-à-dire le nombre d'éléments qui la composent. On va donc procéder élément par élément et chaque fois on prendra en compte le premier élément de la liste. Il faut donc pouvoir accéder au premier élément de la liste. Pour accéder au second élément, il faut supprimer de la liste le premier élément et appeler le programme de façon recursive. On a donc le programme :

```
longueur([X | Y],N) :- longueur(Y,N1), N is N1+1.
```

L'appel `longueur(Y,N1)` est un appel recursif. Ce qui signifie qu'avant la réalisation du test d'arrêt : `longueur([], 0)`, les demandes de `N is N1+1` sont empilées et ne sont pas exécutées. Elles vont commencer à être exécutées, et dans l'ordre inverse de leur empilement, après l'exécution du test d'arrêt. Concrètement si on applique ce programme à la liste `L` nous aurons le déroulement suivant :

1er appel : `longueur([toto | 1, av_du_parc, cergy, la_logique_est_super],N)`

- État du programme :
  `longueur([ 1, av_du_parc, cergy, la_logique_est_super ],N1)`
- État de la pile :

N is N1 + 1.

2e appel : `longueur([ 1 | av_du_parc, cergy, la_logique_est_super ],N1)`

- État du programme :
  `longueur([ av_du_parc, cergy, la_logique_est_super],N2)`
- État de la pile :

N1 is N2 + 1.
N is N1 + 1.

3e appel : `longueur([ av_du_parc | cergy, la_logique_est_super],N2)`

- État du programme :
  `longueur([ cergy, la_logique_est_super],N3)`
- État de la pile :

N2 is N3 + 1.
N1 is N2 + 1.
N is N1 + 1.

4e appel : `longueur([ cergy | la_logique_est_super],N3)`

- État du programme :
  `longueur([la_logique_est_super],N4)`
- État de la pile :

N3 is N4 + 1.
N2 is N3 + 1.
N1 is N2 + 1.
N is N1 + 1.

5e appel : `longueur([la_logique_est_super], | []], N4)`

- État du programme :
  `longueur([], N5)`
- État de la pile :

N4 is N5 + 1.
N3 is N4 + 1.
N2 is N3 + 1.
N1 is N2 + 1.
N is N1 + 1.

6e appel : `longueur([],N5)`

- État du programme :

  `N5 ⟵ 0`

- État de la pile :

`N4 is N5 + 1. N3 is N4 + 1.`
`N2 is N3 + 1.`
`N1 is N2 + 1.`
`N is N1 +1.`

Le test d'arrêt sert ici comme initialisation de la valeur de la longueur à 0. Les ordres successifs d'addition de la valeur 1 aux différents valeurs de `N` n'ont pas été exécutés mais seulement stockés dans la pile. Dès que le programme a rencontré un test d'arrêt, les ordres stockés dans la pile commencent à être exécutés. Ainsi la suite du programme se fera par de depilement successifs et exécution des commandes depilées. Nous avons donc :

7e étape : `N5 = 0`

- État du programme :

  `N4 ⟵ N5 +1 = 0+1 = 1`

- État de la pile :

`N3 is N4 + 1.`
`N2 is N3 + 1.`
`N1 is N2 + 1.`
`N is N1 + 1.`

8e étape : `N4 = 1`

- État du programme :

  `N3 ⟵ N4 +1 = 1+1 = 2`

- État de la pile :

`N2 is N3 + 1.`
`N1 is N2 + 1.`
`N is N1 + 1.`

9e étape : `N3 = 2`

- État du programme :

  `N2 ⟵ N3 +1 = 2+1 = 3`

- État de la pile :

`N1 is N2 + 1.`
`N is N1 + 1.`

10e étape : `N2 = 3`

- État du programme :

  `N1 ⟵ N2 +1 = 3+1 = 4`

- État de la pile :

`N is N1 + 1.`

11e étape : `N1 = 4`

- État du programme :

  `N ⟵ N1 +1 = 4+1 = 5`

- État de la pile :

`< pile vide >`

En examinant le déroulement du programme, on constate que le test d'arrêt est le dernier a être évoqué lors des appels récursifs mais le premier a être exécuté lors du depilement des ordres

empilés. Par conséquent il faut considérer les tests d'arrêt comme la partie du programme qui initialisent les valeurs des variables utilisées.

8.3 Techniques de recursivité

Nous présentons ci-après les techniques de base qui permettent de réaliser des programmes recursifs en `Prolog`.

8.3.1 Recursivité pour les fonctions numériques

Bien qu'en `Prolog` nous avons seulement des prédicats, nous pouvons envisager d'avoir des fonctions si leur résultat est stocké dans une variable qui fait partie des arguments de la fonction. Ainsi, par exemple, on peut envisager d'écrire un prédicat qui sera en réalité une fonction qui calcule la somme de N premiers nombres naturels. Ce prédicat peut avoir la forme suivante :

PROGRAMME 8.3.1

```
somme(0,0).

somme(N,Somme) :- N > 0, N1 is N - 1, somme(N1, Somme1),
                                 Somme is Somme1 + N.
```

On constate donc que pour programmer une fonction numérique sous forme récursive, il faut

(1) *Déterminer le(s) test(s) d'arrêt*, c'est-à-dire décider quand la fonction retourne une valeur prédéterminée, sans appel recursif à elle-même.

Le test d'arrêt pour une fonction numérique se fait en comparant la valeur d'une variable, dont son contenu évolue en fonction des appels recursifs avec une valeur fixée par le programme. La valeur prédéterminée que le programme retourne est la valeur d'initialisation de la variable dont nous venons de parler.

(2) *Déterminer le(s) cas recursif(s).* Un appel recursif d'une fonction s'effectue avec un argument plus simple et on utilise le résultat pour calculer la réponse de l'argument courant. Un argument plus simple en recursivité numérique est un argument qui est plus proche de la valeur utilisée pour l'initialisation par le test d'arrêt.

ASCÈSE 8.1 *Calcul du factoriel $n!$.*

ASCÈSE 8.2 *Calcul de la puissance d'un nombre x^n.*

ASCÈSE 8.3 *Calcul des nombres de Fibonacci.*

ASCÈSE 8.4 *Calcul du plus grand diviseur commun et du plus petit commun multiplicateur de deux entiers.*

8.3.2 Recursivité simple

Ce cas concerne les listes qui n'ont pas des sous-listes ou même si elles en ont, on ne les prendra pas en considération.

Si nous avons à faire un traitement sur un élément quelconque, il faut l'avoir soit stocké au préalable dans une base de données, soit introduit dans une liste. Dans ce dernier cas et étant donné que nous ne pouvons pas indexer les listes par des pointeurs, on est obligé de parcourir la liste jusqu'à arriver à atteindre l'élément voulu. Ce parcours se fera par des appels recursifs où un des arguments sera la liste qui, à chaque appel, sera systématiquement debarassée de son premier élément. La technique donc de la programmation recursive pour les listes avec arrêt simple est identique à celle pour les fonctions numériques, mais les tests d'arrêt se font sur les listes et leurs éléments et non pas sur la valeur d'une variable. On distingue trois types de tests d'arrêt que nous présentons séparement ci-après.

8.3.2.1 Arrêt lorsque la liste est vide

Le programme s'arrête lorsque la liste que nous sommes en train de traiter devient vide. Les programmes que nous pouvons mettre sous ce type sont

- soit des programmes d'énumération : nombre d'éléments qu'une liste contient, nombre d'occurences d'un élément dans une liste, etc.
- soit des programmes d'affichage du contenu d'une liste ou de copie d'une liste dans une autre liste ou, encore, la concaténation de deux listes.

Nous donnons comme exemple le calcul de la longueur d'une liste

PROGRAMME 8.3.2

```
longueur([], 0).

longueur([Tete | Reste], Longueur) :- longueur(Reste,Longueur1),
                                     Longueur is Longueur1 + 1.
```

Longueur is Longueur1 + 1.

ASCÈSE 8.5 *Afficher le contenu d'une liste.*

ASCÈSE 8.6 *Copier une liste dans une nouvelle liste.*

ASCÈSE 8.7 *Concatener deux listes en créant une troisième.*

8.3.2.2 Arrêt lorsqu'un élément spécifique a été retrouvé

On s'arrête dès qu'un élément spécifique, fixé au préalable a été retrouvé. Dans cette catégorie des programmes on retrouve des programmes qui fondent leur traitement sur le fait qu'élément particulier appartient à une liste comme, par exemple, le programme `membre`.

PROGRAMME 8.3.3

```
membre(X, [X | _]).
membre(X, [Tete | Reste]) :- membre(X, Reste).
```

Bien sûr un tel programme pose le problème de sa fin dans le cas où l'élément spécifique ne fait pas partie de la liste. Normalement dans ce cas le programme s'arrête en échec. On peut éviter cette sortie en échec, en ajoutant la clause :

```
membre(_, []).
```

Mais en procédant ainsi, on ignore si on a trouvé ou non l'élément spécifique. Si on s'inspirait de la programmation impérative, on serait tenté ici d'ajouter un indicateur qui nous informererait sur le résultat effectif. Mais la bonne solution en Prolog est de distinguer le traitement de deux situations en utilisant l'opérateur " `ou` " comme suit :

```
toto :- (membre(a, [b,a,c]),
          write('L \'el\'ement a fait partie de la liste'));
        (write ('L \'el\'ement a ne fait pas partie de la liste')), nl.
```

Si le traitement particulier dans chaque cas est long, on peut envisager d'utiliser deux clauses qui s'excluent mutuellement :

```
toto :- membre(a, [b,a,c]),
        write('L \'el\'ement a fait partie de la liste')), nl.
toto :- not(membre(a, [b,a,c])),
        write ('L \'el\'ement a ne fait pas partie de la liste')), nl.
```

ASCÈSE 8.8 *Donner la position d'un élément dans une liste.*

ASCÈSE 8.9 *Supprimer un élément d'une liste et obtenir une nouvelle liste sans cet élément.*

ASCÈSE 8.10 *Remplacer dans une liste un élément par un autre et obtenir une nouvelle liste.*

ASCÈSE 8.11 *Insérer dans une liste, après un élément spécifique, un autre élément et obtenir une nouvelle liste.*

8.3.2.3 Arrêt lorsqu'une position spécifique a été atteinte

On s'arrête dès qu'une position spécifique, fixée au préalable a été retrouvée. Dans cette catégorie des programmes on retrouve des programmes qui fondent leur traitement sur l'élément qui occupe une place particulière dans une liste comme par exemple le programme suivant qui affiche le *n*-ième élément d'une liste

PROGRAMME 8.3.4

```
afficheNth(1, [X|_]) :- write(X), nl.

afficheNth(N, [Tete | Reste]) :- N > 1, N1 is N - 1,
                                 afficheNth(N1, Reste).
```

ASCÈSE 8.12 *Supprimer le n-ième élément d'une liste et obtenir une nouvelle liste.*

ASCÈSE 8.13 *Remplacer dans une liste un élément dans une position donnée par un autre et obtenir une nouvelle liste.*

ASCÈSE 8.14 *Insérer dans une liste, avant un élément qui se trouve à une position spécifique, un autre élément et obtenir une nouvelle liste.*

8.3.3 Recursivité multiple

On doit travailler avec des récursivités multiples si la liste que nous sommes en train de traiter contient des sous-listes. Si e.g. on cherche à savoir si un élément donné fait partie d'une liste et si cette liste contient des sous-listes, on doit examiner séparement ses sous-listes.

La programmation des tests d'arrêt pour la recursivité multiple est identique à celle de la programmation simple. Par contre la programmation des cas recursifs est différente. On utilisera la technique de *recursivité Tête – Reste* dont nous présentons ci-après un exposé succint.

Programmation selon la technique de récursivité Tête – Reste. On suppose que nous avons une fonction qui a comme argument une liste représentée par `[Tete | Reste]` et que nous allons appeler cette fonction de façon recursive en modifiant l'argument représenté par la liste.

(1) Déterminer le(s) cas de reste – recursivité.

Ce sont les cas où la tête `Tete` de la liste est un atome et nous appelons recursivement la fonction avec comme argument pour la liste son `Reste`.

Il y a deux type de reste – recursivité.

(a) On retourne simplement les résultats de la fonction appelée récursivement sur le reste de la liste.

(b) On associe les résultats de reste – recursivité avec une valeur dérivée de la tête de la liste.

(2) Déterminer les cas de tête – reste recursivité.

Ce sont les cas où la tête `Tete` de la liste est une liste. Dans ce cas il faut appeler recursivement la fonction avec comme argument la tête de la liste et, ensuite, appeler recursivement la fonction avec comme argument le reste de la liste. À la fin il faut associer les résultats de deux appels, pour obtenir le résultat correct pour la liste entière.

On présente comme exemple d'application de la recursivité Tête – Reste la recherche d'un élément dans une liste contenant des sous-listes.

PROGRAMME 8.3.5

```
membreT(X, [X | _]).   % Test d'arr\^et

membreT(X, [Tete | Reste]) :- atom(Tete),
                              membreT(X, Reste). % Reste - recursivit\'e

membreT(X, [Tete | Reste]) :- not(atom(Tete)),
                              (membreT(X,Tete);
                              membreT(X, Reste)).% Test - reste recursivit\'e
```

On utilise aussi la recursivité multiple dans des cas où on doit traiter en même temps plusieurs listes. Examinons, à titre d'exemple, le programme qui opère un tri d'une liste numérique selon ses valeurs ascendantes .

PROGRAMME 8.3.6

```
tri([X|Xs],Y) :- partition(Xs,X,Petits,Grands),
                 tri(Petits,Ps),
                 tri(Grands,Gs),
                 conc(Ps,[X|Gs],Y).
```

```
tri([],[]).

partition([X|Xs],Y,[X|Petits],Grands) :- X < Y,
                                    partition(Xs,Y,Petits,Grands).

partition([X|Xs],Y,Petits,[X|Grands]) :- X >= Y,
                                    partition(Xs,Y,Petits,Grands).

partition([],Y,[],[]).
```

Bien évidement on n'utilise pas la technique de tête – reste recursivité mais le programme `tri` est appelé deux fois, comme en recursivité simple, avec deux listes différentes.

ASCÈSE 8.15 *Étant donnée une liste qui contient des sous-listes, obtenir un nouvelle liste qui a les mêmes éléments que la première liste mais sans sous-listes.*

ASCÈSE 8.16 *Faire un tri selon l'ordre alphabétique d'une liste qui contient des noms.*

8.4 Exercices

EXERCICE 8.1 *Structurer une base de connaissances de sorte que si nous avons comme faits qu'un cheval est plus rapide qu'un chien qui, à son tour, est plus rapide qu'un lapin, alors on peut répondre, pour des animaux particuliers, lequel est le plus rapide.*

EXERCICE 8.2 *Disséquer une liste.*

Application : `disseq([b,a,c])?; disseq([])?; disseq([b, [a, c]])?`

Table des matières