Notes d'informatique fondamentale (cours pour l'École d'Ingénieurs EIDD, deuxième année)

Roberto M. Amadio

Notes d'informatique fondamentale

Roberto M. Amadio
Université de Paris

3 janvier 2022

Table des matières

Chapitre 1

Langages rationnels

La théorie des langages formels s'intéresse notamment aux problèmes suivants :
— définir des outils (automates, grammaires, ...) qui permettent de décrire de façon synthétique et précise un langage formel,
— classifier les langages formels d'après la *structure* des outils qui les définissent.
— à partir d'une spécification d'un langage L, construire de façon plus ou moins automatique un programme (un automate) qui *décide* si un mot w appartient à L,
— développer des méthodes pour montrer qu'un certain langage n'est pas dans une certaine classe.

Dans ce chapitre, on se focalise en particulier sur une classe de langages formels dits rationnels (ou réguliers ou reconnaissables) et on se limite à présenter des résultats élémentaires qui remontent essentiellement aux années 1960 [RS59]. Le lecteur intéressé au développement ultérieur du sujet peut consulter, par exemple, [Per90].

1.1 Notation pour les langages formels

Un *alphabet* Σ est un ensemble fini et non-vide. On appelle *caractère* un élément de Σ. Si X est un ensemble alors X^* est l'ensemble des séquences finies d'éléments de X :

$$X^* = \{x_1 \dots x_n \mid n \geq 0, x_i \in X\} \ . \tag{1.1}$$

En particulier, si Σ est un alphabet alors on appelle *mots* les éléments de Σ^* et on dénote avec ϵ la séquence vide. On associe avec chaque mot $w \in \Sigma^*$ sa *longueur* $|w| \in \mathbf{N}$ en définissant $|\epsilon| = 0$ et $|aw| = 1 + |w|$ si $a \in \Sigma$ et $w \in \Sigma^*$. Si $w_1, w_2 \in \Sigma^*$ alors on dénote par $w_1 \cdot w_2 \in \Sigma^*$ la *concaténation* de mots. Souvent, on omet le symbole $\cdot$ et on écrit $w_1 w_2$.

On remarque que l'opération de concaténation est associative et que le mot vide ϵ est une identité gauche et droite.

Un *langage formel* L sur un alphabet Σ est simplement un sous-ensemble de Σ^*. L'opération de concaténation est étendue aux langages de la façon suivante : si $L_1, L_2 \subseteq \Sigma^*$ alors

$$L_1 \cdot L_2 = \{w_1 w_2 \mid w_i \in L_i, i = 1, 2\} \ .$$

La concaténation de langages est aussi une opération associative ayant le langage $\{\epsilon\}$ comme identité gauche et droite. Comme pour les mots, souvent on omet le symbole $\cdot$ et on écrit $L_1 L_2$.

L'*itération* d'un langage L est dénotée par L^* et définie par :

$$L^0 = \{\epsilon\} \ , \quad L^{n+1} = LL^n \ , \quad L^* = \bigcup_{n\in\mathbf{N}} L^n \ .$$

En particulier, on remarque que $\emptyset^* = \{\epsilon\} \neq \emptyset$. On définit aussi L^+ comme $L^+ = LL^*$.

La proposition suivante, connue comme lemme d'Arden [Ard61], montre que dans l'espace des langages on peut toujours trouver un point fixe d'une transformation affine (droite).

Proposition 1.1.1 *Soient A et B deux langages sur un alphabet Σ. L'équation $X = AX \cup B$ a une plus petite solution (par rapport à l'inclusion de langages) qui s'exprime par $X = A^*B$.*

Idée de la preuve. On vérifie aisément que A^*B est une solution car :

$$A(A^*B) \cup B = A^+B \cup A^0B = A^*B \ .$$

Soit Y une autre solution. On prouve par récurrence sur $n \geq 0$ que $A^nB \subseteq Y$. Pour $n = 0$, on a :

$$A^0B = B \subseteq AY \cup B \subseteq Y \ .$$

Supposons $A^nB \subseteq Y$. Alors :

$$A^{n+1}B = A(A^nB) \subseteq AY \subseteq AY \cup B \subseteq Y \ .$$

Comme $A^*B = \bigcup_{n\geq 0} A^nB$, on peut conclure que $A^*B \subseteq Y$. □

1.2 Automates finis déterministes (AFD)

On introduit une classe de programmes très simples qu'on appelle automates finis déterministes.

Langages reconnaissables

Définition 1.2.1 (AFD) *Un automate fini déterministe (AFD) M est un vecteur $(\Sigma, Q, q_0$ $F, \delta)$ où Σ est un alphabet, Q est un ensemble fini qui représente l'ensemble des états de l'automate, $q_0 \in Q$ est l'état initial, $F \subseteq Q$ est l'ensemble des états finaux (ou accepteurs), et $\delta : \Sigma \times Q \to Q$ est la fonction de transition.*

Un AFD se représente graphiquement comme un graphe dirigé tel que : (i) les noeuds correspondent aux états, (ii) il y a une arête étiquetée par a de q à q' si et seulement si $\delta(a, q) = q'$, (iii) le noeud qui correspond à l'état initial est marqué par $>$ et (iv) les noeuds qui correspondent aux états finaux ont un double contour.

Dans la suite, on procède en trois étapes :

1. On définit la notion de *configuration* d'un automate.
2. On décrit comment un automate peut se déplacer d'une configuration à une autre.
3. On spécifie quels mots sont acceptés par l'automate.

Une méthodologie similaire est utilisée dans le chapitre 2 pour un type d'automate plus général qu'on appelle *machine de Turing*.

Définition 1.2.2 (langage reconnu) *Soit $M = (\Sigma, Q, q_0, F, \delta)$ un AFD. Une configuration est un couple $(w, q) \in \Sigma^* \times Q$. On définit une relation de réduction $\vdash_M$ par $(aw, q) \vdash_M (w, \delta(a, q))$ et on suppose que $\vdash_M^*$ est la clôture réflexive et transitive de $\vdash_M$. Le langage $\mathcal{L}(M)$ reconnu (ou accepté) par M est défini par :*

$$\mathcal{L}(M) = \{w \in \Sigma^* \mid (w, q_0) \vdash_M^* (\epsilon, q) \text{ et } q \in F\} .$$

On dit aussi que un langage L est *reconnaissable* s'il y a un AFD M tel que $L = \mathcal{L}(M)$.

Exemple 1.2.3 (de l'AFD au langage) *Soit $M = (\{a, b\}, \{1, 2\}, 1, \{2\}, \delta)$ avec fonction de transition δ spécifiée par :*

δ	a	b
1	1	2
2	1	2

Il n'est pas difficile de montrer que $\mathcal{L}(M)$ est l'ensemble des mots qui terminent par b.

Remarque 1.2.4 *Dans la définition de AFD, on insiste pour que pour chaque état q et pour chaque caractère a de l'alphabet il y ait exactement* une arête *sortante de q avec étiquette a. En pratique, on peut relâcher cette condition et demander juste qu'il y ait* au plus une arête *sortante de q avec étiquette a. Un tel automate peut être transformé facilement en un AFD en introduisant un état 'puits' q_s et en étendant la fonction de transition δ de façon telle que $\delta(a, q_s) = q_s$ pour tout $a \in \Sigma$ et $\delta(a, q) = q_s$ chaque fois que $\delta(a, q)$ n'est pas défini.*

Indice d'un langage et minimisation d'AFD

Tout langage induit une relation d'équivalence sur les mots. On va montrer que cette relation a un nombre fini de classes d'équivalence ssi le langage est reconnaissable et que dans ce cas il y a un AFD qui reconnaît le langage et qui a un nombre minimum d'états. L'ensemble de ces résultats et connu comme théorème de Myhill-Nerode [Ner58].

Définition 1.2.5 (équivalence de mots) *Si $L \subseteq \Sigma^*$ alors on dénote par $\equiv_L$ la relation d'équivalence sur les mots dans Σ^* définie par :*

$$w \equiv_L w' \text{ si } \forall z \in \Sigma^* \ (wz \in L \text{ ssi } w'z \in L) .$$

On dénote par $\Sigma^*/\equiv_L$ l'ensemble quotient des classes d'équivalences induites par $\equiv_L$ et on dit que L est d'*indice fini* si l'ensemble quotient est fini.

Proposition 1.2.6 *Si L est reconnu par un AFD avec n états alors $\sharp(\Sigma^*/\equiv_L) \leq n$.*

Idée de la preuve. Par contradiction, supposons $w_1, \ldots, w_{n+1}$ mots tels que $w_i \not\equiv_L w_j$ si $i \neq j$. Comme on a moins d'états que de mots il doit y avoir un état q et deux mots w_i, w_j $(i \neq j)$ tels que $(q_0, w_i) \vdash^* (q, \epsilon)$ et $(q_0, w_j) \vdash^* (q, \epsilon)$. Mais dans ce cas, pour tout mot z, M accepte $w_i z$ ssi M accepte $w_j z$, ce qui revient à dire que $w_i \equiv_L w_j$. Contradiction. □

Proposition 1.2.7 *Si L est d'indice fini n alors il y a un AFD avec n états qui reconnaît L.*

Idée de la preuve. On construit un AFD qui reconnaît L :

— les états sont les classes d'équivalence $L/\equiv_L$,
— l'état initial est $[\epsilon]_{\equiv_L}$,
— les états accepteurs sont les classes d'équivalence qui contiennent un mot dans L,
— la fonction de transition est $\delta(a,[w]_{\equiv_L}) = [wa]_{\equiv_L}$. □

Corollaire 1.2.8 *Pour tout langage reconnaissable il existe un AFD qui reconnaît le langage et qui a un nombre minimum d'états.*

IDÉE DE LA PREUVE. Soit L un langage reconnu par un AFD avec m états. Par la proposition 1.2.6, L doit être d'indice fini n et $n \leq m$ et par la proposition 1.2.7 il y a un AFD avec exactement n états qui reconnaît L. □

En pratique, on dispose d'une description du langage par un AFD (ou par un autre formalisme qu'on peut traduire en AFD) et on construit un AFD minimum en calculant un quotient des états.

Proposition 1.2.9 (minimisation AFD) *Soit $M = (\Sigma, Q, q_0, F, \delta)$ un AFD. Alors on peut construire un AFD $M' = (\Sigma, Q', q'_0, F', \delta)$ qui reconnaît le même langage et qui a un nombre minimum d'états.*

IDÉE DE LA PREUVE. On va supposer que tous les états dans Q sont accessibles depuis l'état initial q_0. A défaut, on peut les supprimer sans changer le langage reconnu par l'automate. Ensuite on définit une suite de relations $\sim_n$, $n \geq 0$, sur les états :

$$\begin{aligned} \sim_0 &= Q \times Q \ , \\ \sim_{n+1} &= \{(q,q') \mid (q \in F \text{ ssi } q' \in F) \text{ et } \forall a \in \Sigma \ (\delta(a,q) \sim_n \delta(a,q'))\} \ . \end{aligned}$$

Il est immédiat de vérifier que : (i) $\sim_n$ est un relation d'équivalence, (ii) $\sim_n \supseteq \sim_{n+1}$ et (iii) $\sim_n$ converge en un nombre fini de pas à une relation $\sim$. On peut maintenant construire l'automate minimum M'. Si $q \in Q$ est un état on dénote par $[q]_\sim$ sa classe d'équivalence par rapport à la relation $\sim$ et on définit :

$$Q' = \{[q]_\sim \mid q \in Q\}, \quad q'_0 = [q_0]_\sim, \quad \delta'(a,[q]_\sim) = [\delta(a,q)]_\sim, \quad F' = \{[q]_\sim \mid q \in F\} \ .$$

□

Exemple 1.2.10 (minimisation) *Soit $M = (\{q_0,q_1,q_2,q_3,q_4,q_5\},\{0,1\},q_0,\delta,\{q_2,q_3,q_4\})$ un AFD avec transitions :*

	q_0	q_1	q_2	q_3	q_4	q_5
0	q_1	q_0	q_4	q_4	q_4	q_5
1	q_2	q_3	q_5	q_5	q_5	q_5

En calculant $\sim$, on trouve $q_0 \sim q_1$ et $q_2 \sim q_3 \sim q_4$ et on obtient ainsi un AFD avec 3 états.

1.3 Expressions rationnelles

On introduit une classe de langages formels.

Définition 1.3.1 (langages rationnels) *L'ensemble des langages rationnels sur un alphabet Σ est le plus petit ensemble de langages qui contient les sous-ensembles finis de Σ^* (les langages finis) et qui est stable par rapport aux opérations d'union, concaténation et itération.*

Les expressions rationnelles sont une notation pratique pour dénoter les langages rationnels.

Définition 1.3.2 (expressions rationnelles) *L'ensemble E des expressions rationnels sur un alphabet Σ est défini comme $\bigcup_{n\geq 0} E_n$ où :*

$$\begin{aligned} E_0 &= \Sigma \cup \{\emptyset\} \cup \{\epsilon\} \\ E_{n+1} &= E_n \cup \{(+, \alpha, \beta) \mid \alpha, \beta \in E_n\} \cup \{(\cdot, \alpha, \beta) \mid \alpha, \beta \in E_n\} \cup \{(*, \alpha) \mid \alpha \in E_n\} \ . \end{aligned}$$

Il est entendu que $\emptyset$ et ϵ sont deux symboles différents qui ne font pas partie de l'alphabet Σ. Si α est une expression rationnelle on définit par $[\![\alpha]\!]$ le langage rationnel associé. Ce langage est défini de façon unique une fois qu'on a fixé les interprétations (attendues !) pour les symboles $\Sigma \cup \{\emptyset, \epsilon, +, \cdot, *\}$. A savoir :

Symbole	Interprétation	
$a \in \Sigma$	$\{a\}$	(ensemble singleton)
$\emptyset$	$\emptyset$	(ensemble vide)
ϵ	$\{\epsilon\}$	(ensemble contenant le mot vide)
$+$	union langages	
$\cdot$	concaténation langages	
$*$	itération langage	

Ce qu'on vient de décrire est la *syntaxe abstraite* des expressions rationnelles. En pratique, on utilise une *syntaxe concrète* dans laquelle le symbole + est infixe, le symbole $\cdot$ est omis et le symbole $*$ est postfixe :

Syntaxe abstraite	Syntaxe concrète
$(+, \alpha, \beta)$	$(\alpha + \beta)$
$(\cdot, \alpha, \beta)$	$(\alpha\beta)$
$(*, \alpha)$	(α^*)

Par ailleurs, on supprime des parenthèses en supposant que $*$ à priorité sur $\cdot$ qui a priorité sur $+$ et que $\cdot$ et $+$ associent à gauche (par exemple). Avec ces conventions, on écrira $(a+b)cd^*$ pour $(\cdot, (\cdot, (+, a, b), c), (*, d))$.

Le résultat suivant est connu comme théorème de Kleene et sa preuve est basée sur un certain nombre de constructions intéressantes décrites dans cette section et la prochaine.

Proposition 1.3.3 ([Kle56]) *Les langages rationnels sont exactement ceux reconnus par un AFD.*

Pour l'instant, on explique comment associer à un AFD une expression rationnelle qui dénote le langage reconnu par l'AFD. Pour ce faire, on procède en deux étapes.

1. On voit un AFD comme un système d'équations linéaires sur les langages de façon telle que la solution du système est le langage reconnu par l'AFD.
2. On donne une méthode pour résoudre le système d'équations et exprimer la solution comme une expression rationnelle.

De l'AFD à un système linéaire droit

Soient $q_0, \ldots, q_{m-1}$ les m états de l'AFD ; ces états deviennent les variables du système. Si l'alphabet Σ a n symboles $a_1, \ldots, a_n$ alors pour chaque état q_i, $i = 0, \ldots, m-1$, on écrit l'équation :

$$q_i = a_1\delta(a_1, q_i) + \cdots + a_n\delta(a_m, q_i) + \beta_i \tag{1.2}$$

où $\beta_i = \Sigma_{\delta(q_i,a_i)\in F} a_i$ est la somme de tous les symboles qui permettent d'aller de q_i dans un état accepteur. Cette règle a une exception si l'état initial q_0 est aussi un état accepteur ; dans ce cas, on ajoute l'expression ϵ à la somme β_0.

Exemple 1.3.4 (construction du système linéaire droit) *Soit $M = (Q, \{0,1\}, q_0, \delta, \{q_0\})$ un AFD tel que $Q = \{q_0, q_1, q_2\}$ et la fonction de transition est spécifiée par :*

	q_0	q_1	q_2
0	q_0	q_0	q_1
1	q_1	q_2	q_2

Le système linéaire droit associé est :

$$\begin{cases} q_0 &= 0q_0 + 1q_1 + 0 + \epsilon \\ q_1 &= 0q_0 + 1q_2 + 0 \\ q_2 &= 0q_1 + 1q_2 \end{cases} \tag{1.3}$$

Remarque 1.3.5 *Les équations (1.2) peuvent s'orienter de gauche à droite. On parle alors de* règles *(ou productions) d'une* grammaire linéaire droite*. Ainsi, si $[\![\beta]\!] = \{a_{i_1}, \ldots, a_{i_k}\}$, alors on écrit :*

$$q_i \to a_1\delta(a_1, q_i) \mid \cdots \mid a_n\delta(a_m, q_i) \mid a_{i_1} \mid \cdots \mid a_{i_k} \tag{1.4}$$

pour dire que le symbole q_i peut être remplacé *par une suite de symboles qui se trouve sur la droite de la flèche.*

Solution du système

Par la proposition 1.1.1, on sait que si A et B sont des langages rationnels alors la plus petite solution de l'équation $X = AX \cup B$ est le langage rationnel A^*B. Pour résoudre le système (1.2), on réécrit une équation sous la forme :

$$q = \alpha q + \beta \ ,$$

où q n'apparaît pas dans α et β. Ensuite on remplace q par $\alpha^*\beta$ dans les équations qui restent et on se ramène à un système avec $m-1$ équations en $m-1$ variables.

Exemple 1.3.6 (solution du système) *En continuant l'exemple (1.3), on va résoudre le système. De la dernière équation, on dérive :*

$$q_2 = 1^*0q_1 \tag{1.5}$$

En remplaçant dans la deuxième équation, on obtient : $q_1 = 1^+0q_1 + (0q_0 + 0)$, qu'on peut résoudre :

$$q_1 = (1^+0)^*(0q_0 + 0) \tag{1.6}$$

En remplaçant dans la première équation, on obtient :

$$q_0 = 0q_0 + 1(1^+0)^*0q_0 + 1(1^+0)^*0 + 0 + \epsilon \ . \tag{1.7}$$

En factorisant par rapport à q_0, l'équation se réécrit comme :

$$q_0 = (1(1^+0)^*0 + 0)q_0 + (1(1^+0)^*0 + 0 + \epsilon) \ , \tag{1.8}$$

et elle est résolue par :

$$q_0 = \alpha^*(\alpha + \epsilon) = \alpha^*. \tag{1.9}$$

où $\alpha = (1(1^+0)^*0 + 0)$.

1.4 Automates finis non-déterministes (AFN)

Pour montrer que tout langage rationnel est reconnu par un AFD il est utile d'introduire une notion d'automate *non-déterministe* (AFN). Pour ce faire on va augmenter les possibilités d'un AFD en permettant :

1. zéro ou plusieurs transitions étiquetées avec le même symbole,
2. un changement d'état sans lecture d'un caractère (ϵ-transition).

Définition 1.4.1 (AFN) *Un automate fini non-déterministe (AFN) N est un vecteur $(\Sigma, Q, q_0, F, \delta)$ où Σ est un alphabet, Q est un ensemble fini d'états, q_0 est l'état initial, $F \subseteq Q$ est l'ensemble des états finaux et $\delta : (\Sigma \cup \{\epsilon\}) \times Q \to 2^Q$ est une fonction de transition. Une* configuration *pour un AFN est un couple $(w, q) \in \Sigma^* \times Q$. La* relation de réduction $\vdash_N$ *est définie par :*

$$(w, q) \vdash_N (w', q') \text{ si } w = w''w' \text{ et } q' \in \delta(w'', q) \ ,$$

et le langage reconnu $\mathcal{L}(N)$ *est défini par :*

$$\mathcal{L}(N) = \{w \in \Sigma^* \mid (w, q_0) \vdash_N^* (\epsilon, q) \text{ et } q \in F\} \ .$$

Dans un AFD, étant donné un mot w on trouve un chemin de calcul unique qui va de (w, q_0) à (ϵ, q), pour un certain q. Par opposition, dans un AFN on peut avoir plusieurs chemins, et le w est accepté si *au moins un chemin* mène à un état final. Un problème fondamental est de comprendre si et dans quel mesure le calcul non-déterministe est plus puissant que le calcul déterministe.

Le plan est maintenant le suivant :

1. on va associer à une expression rationnelle un AFN qui reconnaît le langage dénoté par l'expression.
2. on va montrer qu'on peut *déterminiser* un AFN, c'est-à-dire qu'on peut construire un AFD qui accepte le même langage.

En combinant ces deux constructions, on a une méthode pour associer à toute expression rationnelle α un AFD M_α qui reconnaît $[\![\alpha]\!]$.

Proposition 1.4.2 (des expressions rationnelles aux AFN) *On peut associer à toute expression rationnelle α un AFN qui accepte le langage $[\![\alpha]\!]$.*

Idée de la preuve. La preuve est par récurrence sur la structure de l'expression α. La construction d'un AFN pour $\emptyset$, ϵ et $a \in \Sigma$ est immédiate. Pour le pas de récurrence, on suppose avoir construit les AFN N_1 et N_2 qui acceptent les langages α_1 et α_2.

union $\alpha_1 + \alpha_2$ On définit un AFN N dont les états consistent d'un nouveau état q qui est l'état initial de N et des états de N_1 et de N_2. Les états accepteurs de N sont ceux de N_1 et N_2. Les transitions de N sont celles de N_1 et N_2 plus deux nouvelles transitions ϵ de q aux états initiaux de N_1 et N_2.

concaténation $\alpha_1\alpha_2$ On définit un AFN N dont les états sont ceux de N_1 et N_2, l'état initial est celui de N_1, les états accepteurs sont ceux de N_2, les transitions sont celles de N_1 et N_2 plus des transitions ϵ des états accepteurs de N_1 à l'état initial de N_2.

itération α_1^* On définit un AFN N dont les états sont ceux de N_1 plus un nouveau état q qui est l'état initial de N. Les états accepteurs sont ceux de N plus q. Les transitions sont celles de N, plus une transition ϵ de q à l'état initial de N_1, plus des transitions ϵ des états accepteurs de N_1 dans l'état initial de N_1. □

Exemple 1.4.3 (de l'expression à l'AFN) *Soit $\alpha = (a + b)^*a(a + b)$ une expression rationnelle. Si l'on suit littéralement la preuve on obtient un AFN avec* 13 *états ce qui est loin d'être optimal car* 3 *états suffisent.*

Proposition 1.4.4 (déterminisation) *Pour tout AFN on peut construire un AFD qui accepte le même langage.*

Idée de la preuve. D'abord on peut éliminer les ϵ-transitions, en ajoutant une transition étiquetée par a de q à q_1, chaque fois qu'il y a un chemin de q à q_1 dont toutes les arêtes sont étiquetées par ϵ sauf une qui est étiquetée par a. Formellement, on introduit une notion de ϵ-clôture d'un état q comme suit :

$$E(q) = \{q' \mid (\epsilon, q) \vdash_N^* (\epsilon, q')\} \ .$$

Ensuite on construit un nouveau automate $N' = (\Sigma, Q, q_0, F', \delta')$ où $F' = \{q \in Q \mid E(q) \cap F \neq \emptyset\}$ et $\delta' : \Sigma \times Q \to 2^Q$ est définie par

$$\delta'(a, q) = \bigcup_{q' \in E(q)} \{E(q'') \mid q'' \in \delta(a, q')\} \ .$$

En d'autres termes, $(a, q) \vdash_{N'} (\epsilon, q_1)$ ssi

$$(a, q) \vdash_N^* (a, q') \vdash_N (\epsilon, q'') \vdash_N^* (\epsilon, q_1) \ .$$

On peut donc supposer que l'automate N a une fonction de transition δ avec le type suivant $\delta : \Sigma \times Q \to 2^Q$. Supposons que de l'état q, en lisant a, l'automate peut aller ou bien dans q_1 ou bien dans q_2, c'est-à-dire $\delta(a, q) = \{q_1, q_2\}$. Pour simuler ce comportement non-déterministe avec un AFD M, on dit que M placé dans l'état q, en lisant a, peut aller dans un 'nouveau état' $\{q_1, q_2\}$ qui est capable de 'simuler' le comportement à la fois de q_1 et q_2. Formellement, on construit un AFD $M = (\Sigma, 2^Q, \{q_0\}, F_M, \delta_M)$ dont les états sont des *sous-ensembles* de l'ensemble des états de N et tel que :

$$F_M = \{X \subseteq Q \mid X \cap F \neq \emptyset\}, \qquad \delta_M(a, X) = \bigcup_{q \in X} \delta(a, q) \ .$$

□

$$\begin{array}{ccccc} \text{AFD} & \leftarrow_{\text{minim.}} & \text{AFD} & \rightarrow & \text{Éq. Lin.} \\ & & \uparrow_{\text{dét.}} & & \downarrow_{X=AX+B} \\ & & \text{AFN} & \leftarrow & \text{Exp. Rat.} \end{array}$$

TABLE 1.1 – Langages rationnels : formalismes et transformations

Exemple 1.4.5 (déterminisation) *Considérons un AFN* $N = (\{q_0, q_1, q_2\}, \{a, b\}, q_0, \delta, \{q_2\})$ *avec transitions :*

	q_0	q_1	q_2
a	$\{q_0, q_1\}$	$\{q_2\}$	$\emptyset$
b	$\{q_0\}$	$\{q_2\}$	$\emptyset$

qui accepte le langage $[\![(a+b)^*a(a+b)]\!]$. *Si on déterminise, on obtient un AFD* $M = (\{q_0, q_{01}, q_{02}, q_{012}\}, \{a, b\}, q_0, \delta', \{q_{02}, q_{012}\})$ *avec transitions :*

	q_0	q_{01}	q_{02}	q_{012}
a	q_{01}	q_{012}	q_{01}	q_{012}
b	q_0	q_{02}	q_0	q_{02}

1.5 Lemme d'itération

La table 1.1 résume les formalismes utilisés pour représenter les langages rationnels et les transformations associées.

Les langages rationnels contiennent tous les langages finis et ils sont stables par rapport à l'union, l'intersection, le complémentaire et l'itération. Comment trouver un langage qui n'est pas rationnel ? Et bien, il ne faut pas regarder trop loin. Par exemple :

$$L = \{a^m b^m \mid m \geq 0\} \tag{1.10}$$

n'est pas rationnel. Un façon de prouver ce résultat est de noter que l'indice du langage n'est pas fini. En effet, si on prend i, j nombres naturels avec $i \neq j$ on a que $a^i \not\equiv_L a^j$ car $a^i b^i \in L$ et $a^j b^i \notin L$. Une autre approche possible est d'utiliser le résultat suivant connu comme *lemme d'itération* (ou *pumping lemma*, en anglais).

Proposition 1.5.1 (itération) *Si* L *est un langage rationnel alors il existe un nombre naturel* n *tel que si* $w \in L$ *et* $|w| \geq n$ *alors* w *se décompose en* 3 *parties* $w = xyz$ *telles que : (i)* $|y| > 0$, *(ii)* $|xy| \leq n$ *et (iii)* $\forall i \geq 0$ $(xy^i z) \in L$.

IDÉE DE LA PREUVE. Soit n le nombre d'états d'un AFD qui reconnaît L. Si $w \in L$ et $|w| \geq n$ alors w correspond à un chemin dans l'AFD qui va de l'état initial à un état accepteur. En suivant ce chemin, soit q le premier état par lequel on passe deux fois. Le mot x correspond au chemin pour aller de état initial à q, le mot y à la boucle simple qu'on fait pour aller de q à q et le mot z au chemin de q à un état accepteur. □

Exemple 1.5.2 (langage non-rationnel) *Pour un* m *assez grand, on doit avoir* $w = a^m b^m = xyz$ *avec* $|y| > 0$. *Si* y *est composé seulement de* a *(ou seulement de* b*) l'itération de* y *va produire un mot qui a plus de* a *(ou plus de* b*). Si* y *est composé de* a *et de* b *son itération va produire un mot dans lequel on a plusieurs alternances de* a *et de* b *et ce mot n'est pas dans le langage.*

On peut utiliser ce résultat pour montrer que $L' = \{w \mid w \text{ contient autant de } a \text{ que de } b\}$ *n'est pas rationnel non-plus. En effet, il est facile de vérifier que* $L'' = \{a^m b^p \mid m, p \geq 0\}$ *est rationnel. Donc si* L' *était rationnel alors :*

$$L' \cap L'' = \{a^m b^m \mid m \geq 0\}$$

le serait aussi. Contradiction.

Exercices

Exercice 1 (propriétés algébriques) *Montrez que :*

1. *l'union de langages est une opération associative, commutative et idempotente avec le langage vide comme identité,*
2. *la concaténation de langages est une opération associative avec le langage* $\{\epsilon\}$ *comme identité et le langage vide comme élément absorbant,*
3. *on peut distribuer l'union sur la concaténation à gauche et à droite,*
4. *pour tout langage* L, $L^* = (L^*)^*$,
5. *on peut trouver des langages* L_1 *et* L_2 *tels que* $(L_1 \cup L_2)^* \neq L_1^* \cup L_2^*$ *et*
6. *on peut trouver des langages* L_1 *et* L_2 *tels que* $(L_1 \cdot L_2)^* \neq L_1^* \cdot L_2^*$.

Exercice 2 (solution unique) *Soient* A *et* B *deux langages sur l'alphabet* Σ. *Par la proposition 1.1.1, on sait que l'équation* $X = AX \cup B$ *a toujours une solution. Montrez que si* $\epsilon \notin A$ *alors la solution est unique.*

Exercice 3 (langage reconnu) *Considérons l'automate fini* $M = (Q, \Sigma, \delta, q_0, F)$, *où* $Q = \{q_0, q_1, q_2, q_3\}$, $\Sigma = \{0, 1\}$, $F = \{q_0\}$ *et la fonction* δ *est définie par le tableau suivant :*

	q_0	q_1	q_2	q_3
0	q_2	q_3	q_0	q_1
1	q_1	q_0	q_3	q_2

Les mots 1011010 *et* 101011 *sont-ils acceptés par* M *? Prouvez que* $\mathcal{L}(M)$ *est l'ensemble des mots composés d'un nombre pair de* 0 *et d'un nombre pair de* 1.

Exercice 4 (programmation automates) *Pour chacun des langages suivants, construire un automate fini non déterministe qui l'accepte.*

1. *Les représentations binaires des nombres pairs.*
2. *Les représentations binaires des multiples de* 3 *(on lit le nombre de gauche à droite, donc à partir du chiffre le plus significatif).*
3. *Le langage des mots sur l'alphabet* $\{a, b\}$ *contenant ou bien la chaîne aab ou bien la chaîne aaab.*
4. *Les mots sur l'alphabet :*

$$\{(0,0,0), (0,0,1), (0,1,0), (0,1,1), (1,0,0), (1,0,1), (1,1,0), (1,1,1)\}$$

tels que la troisième composante de chaque tuple peut être vue comme le résultat de la somme en base 2 *des deux premiers composantes. Par exemple, le mot suivant est dans le langage* $(0,0,1)(1,0,0)(0,1,0)(1,1,0)$ *car* 1000 *est la somme en base* 2 *de* 0101 *et* 0011. *Comme dans la question 3., on lit les nombres à partir du chiffre le plus significatif.*

Exercice 5 (transducteurs) *Un* transducteur *à états finis est une sorte d'AFD qui calcule une fonction sur* Σ^* *au lieu de reconnaître un sous-ensemble de* Σ^*. *Les* transducteurs *sont notamment utilisés dans la conception de circuits séquentiels.*[1] *Par rapport à la définition*

1. On entend par circuit séquentiel un circuit constitué de circuits combinatoires et d'éléments (registres, bascules,...) qui permettent de mémoriser l'état du système ; le fonctionnement d'un circuit séquentiel peut être synchronisé par un signal d'horloge ou pas.

1.2.1 d'AFD, on omet l'ensemble d'états accepteurs et on introduit un alphabet de sortie Σ_S et une nouvelle fonction δ_S qu'on appelle fonction de sortie et pour laquelle on distingue deux possibilités :

$$\delta_S : \Sigma \times Q \to \Sigma_S \quad \textit{(machine de Mealy)}$$
$$\delta_S : Q \to \Sigma_S \quad \textit{(machine de Moore).}$$

Dans une machine de Mealy, la sortie est fonction de l'état et de l'entrée, alors que dans une machine de Moore elle dépend seulement de l'état. On suppose que sur une entrée $w = a_1 \cdots a_n \in \Sigma^*$, une machine de Mealy calcule un mot de longueur n : :

$$\delta_S(a_1, q_0) \cdots \delta_S(a_n, q_{n-1}) \ ,$$

où q_0 est l'état initial et on suppose $q_i = \delta(a_i, q_{i-1})$, pour $i = 1, \ldots, n$. Sur la même entrée, une machine de Moore calcule un mot de longueur $n+1$ à savoir :

$$\delta_S(q_0) \cdots \delta_S(q_n)$$

où q_0 est l'état initial et on suppose $q_i = \delta(a_i, q_{i-1})$, pour $i = 1, \ldots, n$.

1. *Programmez une machine de Mealy qui prend en entrée un nombre n représenté en base 2 et émet $n \mod 3$ en base 10. Par exemple, sur une entrée 1010 la machine doit émettre 1221 car elle voit les nombres $1, 2, 5, 10$ (en base 10) et $1, 2, 2, 1 = 1 \mod 3, 2 \mod 3, 5 \mod 3, 10 \mod 3$.*
2. *Montrez que pour toute machine de Mealy on peut construire une machine de Moore qui calcule la même sortie avec un pas de décalage. Appliquez la construction à la machine du point 1.*
3. *D'autre part, montrez que pour toute machine de Moore, on peut construire une machine de Mealy qui calcule la même sortie à condition d'ignorer la première sortie de la machine de Moore.*

Exercice 6 (stabilité langages reconnus) *Soient M_1, M_2 deux AFD qui reconnaissent les langages L_1, L_2, respectivement (sur un alphabet Σ fixé). Montrez que les langages suivants sont aussi reconnaissables par un AFD : (1) le complémentaire $\Sigma^* \backslash L_1$ et (2) l'union $L_1 \cup L_2$. Conclure que la classe des langages reconnus par les AFD est stable par union, intersection et complémentaire.*

Exercice 7 (déterminisation) *Transformez les AFN suivants en AFD minimaux.*

1. *$M = (Q, \Sigma, \delta, q_0, F)$, où $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1, 2\}$, $F = \{q_0, q_2\}$, et la fonction de transition δ est définie par le tableau suivant :*

	q_0	q_1	q_2
0	$\{q_0, q_1, q_2\}$	$\emptyset$	$\emptyset$
1	$\{q_1, q_2\}$	$\{q_1, q_2\}$	$\emptyset$
2	$\{q_2\}$	$\{q_2\}$	$\{q_2\}$

2. *$M = (Q, \Sigma, \delta, q_0, F)$ où $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1, 2\}$, $F = \{q_2\}$, et la fonction de transition δ est définie par le tableau suivant :*

	q_0	q_1	q_2
0	$\{q_0\}$	$\emptyset$	$\emptyset$
1	$\{q_1\}$	$\{q_1\}$	$\emptyset$
2	$\emptyset$	$\emptyset$	$\{q_2\}$
ϵ	$\{q_2\}$	$\{q_2\}$	$\emptyset$

Exercice 8 (des expressions régulières aux AFD, et retour) *1. Soit $\Sigma = \{a, b\}$ un alphabet et $\alpha_k = (a+b)^* a (a+b)^k$, $k \geq 0$ des expressions régulières. Construire un AFN qui reconnaît le langage $[\![\alpha_k]\!]$ et donnez une borne inférieure en fonction de k au nombre d'états d'un AFD minimal qui reconnaît $[\![\alpha_k]\!]$.*

2. Reprendre l'AFD de l'exercice 4(2) qui reconnaît les nombres représentés en base 2 qui sont des multiples de 3. Calculez une expression régulière qui dénote le même langage.

Exercice 9 (inversion) *Si w est un mot on dénote par w^R le même mot lu de droite à gauche. Par exemple $abbc^R = cbba$. Montrez que si L est rationnel alors $L^R = \{w^R \mid w \in L\}$ est rationnel aussi.*

Exercice 10 (synchronisation) *Soit $M = (\Sigma, Q, q_0, \delta, F)$ un AFD. Un mot $s \in \Sigma^*$ est une séquence de synchronisation si*

$$\exists\, q_m \;\; \forall\, q \in Q \;\; (q, s) \vdash_M^* (q_m, \epsilon).$$

En d'autres termes, il y a un état 'maison' q_m dans lequel on peut aller à partir de n'importe quel état q en appliquant la séquence s. Trouvez un AFD qui n'admet pas de séquence de synchronisation et prouvez que si l'AFD a n états et admet une séquence de synchronisation alors il en a une de longueur au plus n^3.

Exercice 11 (préfixe, postfixe, quotient droit) *Soient L et L' deux langages. On définit :*

$$\begin{array}{lll} Pre(L) & = \{w \mid \exists\, w' \;\; ww' \in L\} & \textit{(prefixes)} \\ Post(L) & = \{w \mid \exists\, w' \;\; w'w \in L\} & \textit{(postfixes)} \\ L/L' & = \{w \mid \exists\, w' \in L' \;\; (ww' \in L)\} & \textit{(quotient droit).} \end{array}$$

Prouvez que si L est rationnel alors : (1) $Pre(L)$, (2) $Post(L)$ et (3) L/L' sont rationnels.

Exercice 12 (un problème de décision) *Soit Σ un alphabet. Proposez un algorithme qui prend en entrée deux ensembles finis de mots non vides sur Σ : $U = \{u_1, \ldots, u_k\} \subset \Sigma^+$ et $W = \{w_1, \ldots, w_h\} \subset \Sigma^+$ et décide s'il existe un mot z qui peut être obtenu par concaténation de mots dans U et par concaténation de mots dans W. Il est entendu qu'on peut utiliser les mots dans U ou dans V plusieurs fois. Par exemple, si $\Sigma = \{a, b\}$, $U = \{ab, ba\}$ et $W = \{aba, bab\}$ alors une solution au problème est $z = ababab = (ab)(ab)(ab) = (aba)(bab)$.*

Exercice 13 (deux perles) *Soit L un langage rationnel.*

1. Soit

$$L_{1/2} = \{w \mid \exists\, w' \;\; (ww' \in L \text{ et } |w| = |w'|)\}.$$

Il s'agit donc de prendre la première moitié des mots dans L dont la longueur est un multiple de 2. Prouvez que $L_{1/2}$ est rationnel.

2. Soit :

$$L_{1/3-1/3} = \{w_1 w_3 \mid \exists\, w_2 \;\; (w_1 w_2 w_3 \in L \text{ et } |w_1| = |w_2| = |w_3|)\}$$

Il s'agit donc de prendre le premier et dernier tiers des mots dans L dont la longueur est un multiple de 3. Montrez que L n'est pas forcement rationnel.

Chapitre 2

Calculabilité

Certains problèmes calculatoires demandent une mémoire qui est fonction de la taille de l'entrée (par exemple, la multiplication de deux nombres). De tels problèmes ne peuvent pas être résolus par des automates finis dont la mémoire est bornée *a priori*. On considère le problème de formaliser un modèle de calcul suffisamment général pour calculer tout ce qu'un 'ordinateur' pourrait calculer en disposant d'une quantité illimitée de temps et de mémoire. Plusieurs modèles *équivalents* ont été proposés à partir des années 1930. On base la présentation sur les *machines de Turing* (MdT) pour 3 raisons : (i) c'est un modèle qui généralise les automates finis, (ii) il est assez simple (ceci est une opinion) et (iii) il fait partie des modèles qui peuvent exprimer toutes les fonctions calculables (ceci est une thèse).

Le fait qu'on puisse *définir* ce qui est calculable constitue une avancée scientifique majeure comparable à la découverte de la loi de gravitation universelle pour expliquer le mouvement des planètes ou de l'ADN pour comprendre les mécanismes de reproduction d'une cellule. A priori ce n'est pas du tout clair qu'une telle définition soit possible. Par exemple, on pourrait penser qu'une telle définition va dépendre fortement du mécanisme de calcul utilisé. Or la thèse est que ce n'est pas le cas ! Par ailleurs, le fait de disposer d'une définition robuste de ce qui est calculable va nous permettre de développer une méthodologie pour montrer que certains problèmes ne sont *pas* calculables.

2.1 Machines de Turing

Un automate fini dispose d'un contrôle fini et d'un ruban sur lequel il peut déplacer sa tête de lecture de gauche à droite. Une machine de Turing peut en plus :

— remplacer le symbole lu sur le ruban par un autre symbole,
— déplacer la tête de lecture dans les deux directions (droite/gauche),
— disposer d'un ruban illimité pour mémoriser des informations.

A la différence des AFD/AFN dont la définition est assez standardisée, on trouve une grande variété de définitions de Machines de Turing qui en fin de compte sont toutes équivalentes.

Définition 2.1.1 (machine de Turing) *Une machine de Turing (déterministe) M est un vecteur $M = (Q, \Sigma, \Gamma, \sqcup, q_0, q_a, q_r, \delta)$ où :*

— *Q est un ensemble fini d'états, Σ est l'alphabet d'entrée et Γ est l'alphabet du ruban. On suppose que $Q \cap \Gamma = \emptyset$, $\Sigma \subset \Gamma$ et $\sqcup \in \Gamma \backslash \Sigma$ est un symbole spécial qu'on appelle* blanc.

— $q_0, q_a, q_r \in Q$ *sont des états. En particulier* q_0 *est l'état initial et* q_a, q_r *sont deux états finaux distincts qui entraînent l'arrêt du calcul.*
— $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *est la fonction (déterministe) de transition où* L *pour* left *et* R *pour* right *sont deux symboles.*

Une configuration *de la machine* M *est un mot de la forme* wqw' *où* $q \in Q$, $w, w' \in \Gamma^*$ *et on suppose que* w' *ne termine pas avec le symbole* $\sqcup$. *Une* configuration initiale *est un mot* $q_0 w$ *où* $w \in \Sigma^*$ *représente l'entrée de la machine.*

Intuitivement, une MdT calcule sur un ruban dont la taille n'est pas bornée à droite et qui à partir d'un certain point contient seulement le symbole $\sqcup$. Soit $\sqcup^\omega$ le mot infini $\sqcup\sqcup\sqcup\cdots$ Une configuration wqw' décrit une situation où le ruban contient le mot w à gauche de la tête de lecture et le mot $w'\sqcup^\omega$ à partir de la position de la tête de lecture. En particulier, avec $w' = \epsilon$ on exprime le fait que le ruban contient des symboles $\sqcup$ à partir de la tête de lecture en procédant vers la droite.

Un *pas de calcul* est décrit par la fonction δ. En fonction de l'état courant et du symbole en lecture, la machine se déplace dans un nouvel état, écrit un symbole à la place du symbole lu et déplace la tête de lecture à gauche ou à droite. Le déplacement de la tête de lecture à gauche est impossible si le mot w de la configuration courante wqw' est vide. Dans ce cas la tête de lecture reste sur place.

Pour formaliser ces idées, on introduit d'abord une fonction $\_ : \Gamma^* \to \Gamma^*$ telle que $\underline{w}$ est le plus long préfixe de w qui ne termine pas par $\sqcup$; en d'autres termes $\underline{w}$ est obtenu de w en éliminant tous les symboles $\sqcup$ qui se trouvent à la fin de w.

Définition 2.1.2 (pas de calcul) *La relation* $\vdash_M$ *qui décrit un pas de calcul est la plus petite relation binaire sur les configurations qui satisfait les conditions suivantes où on suppose* $q \notin \{q_a, q_r\}$ *:*

$$\text{si } \delta(q,a) = (q', b, R) \quad \text{alors} \begin{cases} wqaw' \vdash_M wbq'w' \\ wq \vdash_M wbq' & \text{si } a = \sqcup \end{cases}$$

$$\text{si } \delta(q,a) = (q', b, L) \quad \text{alors} \begin{cases} wcqaw' \vdash_M wq'\underline{cbw'} \\ qaw' \vdash_M q'\underline{bw'} \\ wcq \vdash_M wq'\underline{cb} & \text{si } a = \sqcup \\ q \vdash_M q'\underline{b} & \text{si } a = \sqcup \end{cases}$$

La complexité relative de la définition du pas de calcul est due au fait qu'on doit distinguer entre un déplacement droite et gauche et traiter deux cas limites : (i) à partir de la tête de lecture on a seulement des symboles $\sqcup$ à droite et (ii) la tête de lecture ne peut pas se déplacer à gauche. On remarque que, la fonction δ étant totale, le calcul de M s'arrête si et seulement si la machine arrive à un état final (q_a ou q_r). Un automate fini peut accepter ou refuser un mot, une MdT peut aussi boucler. Dans la définition de langage accepté par une MdT, il faut prendre en compte cette troisième possibilité. On arrive ainsi à la notion de langage (semi-)décidable. On verra dans la suite que cette notion est très robuste et largement indépendante des détails du modèle de calcul choisi (dans notre cas les MdT).

Définition 2.1.3 (langage (semi-)décidable) (1) *Un ensemble* $L \subseteq \Sigma^*$ *est semi-décidable s'il existe une MdT* M *telle que* $L = \{w \mid \exists w', w''(q_0 w \vdash_M^* w' q_a w'')\}$. *Dans ce cas, on dit que* M *semi-décide (ou accepte)* L.

(2) *Un ensemble L est décidable s'il existe une MdT M dont le calcul termine toujours et qui semi-décide L. Dans ce cas on dit que M décide L.*

Exemple 2.1.4 *On construit une MdT qui décide $\{a^n b^m \mid n, m \geq 0\}$. On a $\Sigma = \{a, b\}$, $\Gamma = \Sigma \cup \{\sqcup\}$ et $Q = \{q_0, q_a, q_r, q_1\}$. On remarque qu'il est inutile de spécifier le comportement de la fonction δ sur les états q_a et q_r car par définition la MdT s'arrête quand elle arrive à ces états. Par ailleurs, il est aussi inutile de spécifier le caractère écrit et le déplacement effectué par la tête de lecture pour toute transition qui va dans les états finaux. En effet, pour les problèmes de décision on s'intéresse seulement à l'état final et on ignore le contenu du ruban et la position de la tête de lecture. Enfin, on peut interpréter l'absence de spécification comme une transition dans l'état q_r. Avec ces conventions, on peut décrire le comportement de la fonction δ par le tableau :*

	a	b	$\sqcup$
q_0	q_0, a, R	q_1, b, R	$q_a, \_, \_$
q_1	$\_$	q_1, b, R	$q_a, \_, \_$

Comme dans les automates finis, on peut introduire une notation graphique. *Par exemple, on écrira :*

$$q \overset{a/b,L}{\to} q'$$

pour signifier que la MdT dans l'état q et en lisant a, écrit b, se déplace à gauche (L) et va dans l'état q'. On remarquera que dans l'exemple notre MdT se comporte comme un automate fini : elle se déplace seulement à droite et elle ne modifie pas le contenu du ruban.

Exemple 2.1.5 *On construit une MdT qui décide $\{a^n b^n \mid n \geq 0\}$ (l'exemple 1.5.2 de langage non-rationnel). On a $\Sigma = \{a, b\}$, $\Gamma = \Sigma \cup \{X, Y, \sqcup\}$ et $Q = \{q_0, q_a, q_r, q_1, q_2, q_3, q_4\}$. La fonction δ est spécifiée comme suit :*

	a	b	X	Y	$\sqcup$
q_0	q_1, X, R	$\_$	$\_$	$\_$	$q_a, \_, \_$
q_1	q_1, a, R	q_2, Y, L	$\_$	q_1, Y, R	$\_$
q_2	q_2, a, L	$\_$	q_3, X, R	q_2, Y, L	$\_$
q_3	q_1, X, R	$\_$	$\_$	q_4, Y, R	$\_$
q_4	$\_$	$\_$	$\_$	q_4, Y, R	$q_a, \_, \_$

Exemple 2.1.6 *Soit $\Sigma = \{0, 1, \sharp\}$ et $L = \{w\sharp w \mid w \in \{0,1\}^*\}$. On peut construire une MdT qui décide L en prenant $\Gamma = \Sigma \cup \{\sqcup, X\}$. La machine lit le premier caractère b de w, le remplace par X, puis déplace sa tête de lecture à droite pour vérifier que le premier symbole à droite de $\sharp$ est b, le remplace par X, puis revient à gauche du $\sharp$ et ainsi de suite. Un observateur qui regarderait le contenu du ruban verrait par exemple :*

$$01\sharp01\sqcup^\omega \quad X1\sharp01\sqcup^\omega \cdots \quad X1\sharp X1\sqcup^\omega \cdots \quad XX\sharp X1\sqcup^\omega \cdots \quad XX\sharp XX\sqcup^\omega$$

Si un calcul termine on peut aussi voir le 'contenu du ruban' comme le *résultat du calcul*. Plus précisément on considère comme 'résultat du calcul' la concaténation de tous les symboles dans l'alphabet d'entrée qui sont sur le ruban à la fin du calcul. Par exemple, si le ruban a la forme $\sqcup a \sqcup \sqcup ba \sqcup^\omega$ et a, b sont des symboles de l'alphabet d'entrée, le résultat du calcul est aba. On écrit $M(w) \downarrow$ si la MdT M avec entrée w termine et $M(w) = w'$ si $M(w) \downarrow$ avec résultat w'.

Définition 2.1.7 (fonctions (partielles) récursives) (1) *Une fonction partielle* $f : \Sigma^* \to \Sigma^*$ *est une fonction* partielle récursive *s'il existe une MdT* M *avec alphabet d'entrée* Σ *telle que* $f(w) = w'$ *si et seulement si* $M(w) = w'$.

(2) *Une fonction* récursive *est une fonction partielle récursive totale, c'est-à-dire qui est définie sur chaque entrée.* [1]

Remarque 2.1.8 *Les notions de langage (semi-)décidable et de fonction (partielle) récursive sont deux faces de la même pièce. Par exemple, si un langage* L *est semi-décidable alors il y a une fonction partielle récursive dont le domaine de définition coïncide avec* L. *D'autre part, si* $f : \Sigma^* \to \Sigma^*$ *est une fonction partielle récursive alors le graphe de la fonction* $\{(w, w') \mid f(w) = w'\}$ *est un ensemble semi-décidable, à un codage des couples près.*

2.2 Énumérations et MdT universelle

Une variété de structures finies comme arbres, graphes, polynômes, grammaires, MdT,... peuvent être codées comme mots finis d'un alphabet.

Problèmes et langages

Un graphe dirigé fini est un couple (N, A) où N est un ensemble fini de noeuds et $A \subseteq N \times N$ est un ensemble d'arêtes. Deux graphes dirigés (N, A) et (N', A') sont isomorphes s'il existe une bijection $f : N \to N'$ telle que $(n, n') \in A$ ssi $(f(n), f(n')) \in A'$. Notre objectif est de fixer un alphabet Σ et de représenter les graphes dirigés comme un langage sur cet alphabet. Plus précisément on va représenter les graphes dirigés à 'isomorphisme près'. Ceci est justifié par le fait qu'en général on s'intéresse aux propriétés des graphes qui sont invariantes par isomorphisme (connectivité, diamètre,...). On suppose que l'ensemble des noeuds N est un segment initial des nombres naturels codés en binaire, par exemple $N = \{0, 1, 10, 11\}$. En conséquence, A est maintenant un ensemble de couples de nombres naturels codés en binaire. On peut ajouter un symbole $\sharp$ qui agit comme un séparateur. Maintenant le graphe $(\{0, 1, 2, 3\}, \{(2, 0), (1, 3), (2, 3)\})$ peut être représenté par le mot fini sur l'alphabet $\Sigma = \{0, 1, \sharp\}$:

$$\sharp 0 \sharp 1 \sharp 10 \sharp 11 \sharp\sharp 10 \sharp 0 \sharp 1 \sharp 11 \sharp 10 \sharp 11 \sharp \ .$$

Par le biais de ce codage, on peut considérer à isomorphisme près l'ensemble des graphes dirigés comme un certain langage de mots finis sur un alphabet. Si G est un graphe dirigé, on dénote par $\langle G \rangle$ son codage. Supposons maintenant qu'on s'intéresse au problème de savoir si deux graphes dirigés sont isomorphes. [2] On peut reformuler ce problème comme le problème de la reconnaissance du langage :

$$L = \{\langle G \rangle \sharp\sharp\sharp \langle G' \rangle \mid G \text{ et } G' \text{ sont isomorphes}\} \ .$$

1. En français, on utilise aussi le terme *fonction* pour les fonctions partielles et le terme *application* pour les fonctions totales.

2. Notez qu'on peut avoir plusieurs codages qui représentent le même graphe à isomorphisme près.

Fixer un alphabet

On applique maintenant la même méthode aux MdT. Une MdT est un programme. Il est clair que les 'noms' des états n'affectent pas le comportement d'une MdT. Ainsi on peut supposer que les états sont codés, par exemple, en binaire. Considérons maintenant l'alphabet Γ. Il est possible de simuler le comportement d'une MdT M qui utilise un alphabet Γ avec une autre MdT M' qui utilise seulement un alphabet $\{0, 1, \sqcup\}$. Si Γ a n éléments on code chaque élément de Γ par une suite binaire de longueur $k = \lceil log_2(n) \rceil$. Pour simuler un pas de calcul de M, M' doit : (i) lire k symboles consécutifs et en fonction de ces k symboles et de l'état courant (ii) écrire k symboles et (iii) déplacer la tête de lecture de k symboles à droite ou à gauche. Donc, à un codage près, le comportement de toute MdT qui opère sur un alphabet arbitraire peut être simulé par une MdT qui opère sur un alphabet qui est fixé une fois pour toutes.

Énumération de MdT

On s'intéresse maintenant à la représentation comme mots finis des MdT sur un alphabet donné. On peut fixer un codage pour le symbole $\sqcup$, pour les états q_0, q_a, q_r et pour les symboles L, R. Ensuite, la fonction δ peut être représentée en listant son graphe (on peut éventuellement ajouter un symbole spécial pour séparer les différents éléments de la liste comme on l'a fait dans le cas des graphes). En procédant de la sorte, toute MdT est représentée par un mot fini sur un alphabet. Soit $MdT(\Sigma) \subseteq \Sigma^*$ l'ensemble des codages de MdT sur l'alphabet Σ choisi. Les mots qui composent cet ensemble doivent représenter comme une liste la fonction δ d'une MdT. Il est donc décidable de savoir si un mot appartient à $MdT(\Sigma)$. Par ailleurs, on peut définir une fonction récursive et surjective $\varphi : \Sigma^* \to MdT(\Sigma)$. Soit w_0 le codage d'une MdT. La fonction φ est définie par :

$$\varphi(w) = \begin{cases} w & \text{si } w \text{ code une MdT} \\ w_0 & \text{autrement.} \end{cases}$$

Proposition 2.2.1 (bijection entre mots et couples de mots) *Il y a une bijection calculable* $\langle \_, \_ \rangle : \Sigma^* \times \Sigma^* \to \Sigma^*$ *entre les mots et les couples de mots.*

Idée de la preuve. La preuve de ce fait est élémentaire et elle est développée dans les exercices à la fin du chapitre. □

Par le biais de cette bijection, une MdT peut interpréter tout mot w comme un couple de mots $\langle w_1, w_2 \rangle$. Par ailleurs, par le biais de la fonction φ une MdT peut interpréter tout mot comme le codage d'une MdT. Ce fait a une conséquence remarquable.

Proposition 2.2.2 (MdT universelle) *On peut construire une MdT* U *qu'on appelle* MdT universelle *telle que :*

$$U(\langle w_1, w_2 \rangle) = \varphi(w_1)(w_2) \ .$$

Idée de la preuve. La machine U –dont on omet les détails de construction– reçoit un mot w qui est interprété comme un couple de mots w_1, w_2. Ensuite, le mot w_2 est interprété comme l'entrée de la MdT décrite par le premier mot w_1. La MdT U simule la MdT $\varphi(w_1)$ sur l'entrée w_2. Ainsi, la machine U se comporte comme un *interprète* qui reçoit en argument

un programme et une entrée et calcule le résultat du programme sur l'entrée. □

On peut résumer les remarques de cette section par les points suivants.
— Un problème algorithmique peut être (souvent) reformulé comme un problème de reconnaissance d'un langage.
— Sans perte de généralité, nous pouvons limiter notre attention aux MdT qui opèrent sur un alphabet Γ fixé une fois pour toutes.
— On peut coder une MdT comme un mot fini et on peut énumérer tous les codages de MdT sur un alphabet donné.
— On peut construire une MdT *universelle* qui reçoit en entrée le codage d'une MdT M et une entrée w et simule le calcul de M sur w.

2.3 Thèse de Church-Turing

Plusieurs variantes de MdT ont été considérées. Ces variantes n'affectent pas la notion de langage semi-décidable ou décidable mais peuvent changer de façon significative la complexité du calcul.

MdT multi-rubans

Une MdT multi-rubans est une MdT qui dispose d'un nombre fini k de rubans. Sa définition formelle suit celle d'une MdT standard modulo le fait que le type de la fonction de transition δ est maintenant

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k \ .$$

Un pas de calcul se déroule de la façon suivante : en fonction de l'état courant et des symboles lus sur les k rubans, la machine va dans un autre état, remplace les symboles lus par d'autres symboles et déplace les têtes de lecture. Avec la directive S pour *stay* on a la possibilité de garder une tête de lecture à la même place.

Proposition 2.3.1 (simulation MdT multi-rubans) *Soit M une MdT multi-rubans. On peut construire une MdT standard M' qui simule M. Si la complexité de M est $t(n) \geq n$ la complexité de M' est $O(t(n)^2)$.*

Idée de la preuve. Supposons que la MdT M dispose de 3 rubans dont le contenu est $0101\sqcup^\omega$, $aab\sqcup^\omega$ et $ba\sqcup^\omega$ et dont les têtes de lecture sont en deuxième, troisième et première position respectivement. La MdT M' mémorise les trois rubans sur un seul ruban de la façon suivante :

$$\sharp 0\underline{1}01\sharp aa\underline{b}\sharp\underline{b}a\sharp \sqcup^\omega \ .$$

On notera que M' dispose d'un nouveau symbole $\sharp$ pour séparer les rubans et que pour chaque symbole a de M on introduit un nouveau symbole $\underline{a}$. Le symbole souligné indique la position de la tête de lecture.

Un pas de calcul de M est simulé de la façon suivante :
— M' commence par parcourir son ruban de gauche à droite pour calculer les symboles en lecture et déterminer les actions à effectuer.

— Ensuite, M' effectue un deuxième passage dans lequel elle remplace le symbole en lecture (les symboles soulignés) par des nouveaux symboles et éventuellement déplace la tête de lecture (c'est-à-dire, remplace un symbole par un symbole souligné).
— Si le symbole souligné précède le symbole $\sharp$ et le calcul prévoit un déplacement à droite il est nécessaire d'allouer une nouvelle case. A cette fin, la machine M' décale à droite le contenu du ruban.

La borne $O(t(n)^2)$ sur le temps de calcul de la simulation est obtenue de la façon suivante. D'abord on observe que si la complexité de M est $t(n)$, la taille des rubans manipulés par M ne peut jamais dépasser $t(n)$. Ensuite on détermine le nombre d'opérations nécessaires à simuler un pas de calcul de M. Le premier passage est $O(t(n))$. Le deuxième passage est aussi $O(t(n))$ car le décalage à droite peut être effectué au plus k fois si la machine M comporte k rubans et chaque décalage peut être effectué en $O(t(n))$. □

Les machines multi-rubans permettent de donner une preuve simple du fait suivant.

Proposition 2.3.2 (caractérisation décidabilité) *Un langage L est décidable si et seulement si L et son complémentaire L^c sont semi-décidables.*

Idée de la preuve. ($\Rightarrow$) Par définition un langage décidable est semi-décidable. D'une MdT M qui décide L on obtient une MdT M' qui décide L^c simplement en échangeant les états finaux q_a et q_r.

($\Leftarrow$) Soient M et M' les MdT qui décident L et L^c, respectivement. On dérive une MdT N avec 2 rubans qui copie d'abord l'entrée w du premier au deuxième ruban et qui simule ensuite alternativement un pas de réduction de la machine M et un pas de réduction de la machine M'. La machine N accepte si M arrive à l'état q_a et elle refuse si M' arrive à l'état q'_a. La machine N termine toujours car tout mot w est accepté soit par M soit par M'. □

MdT non-déterministes

Une MdT *non-déterministe* M est une MdT dont la fonction de transition δ a le type :

$$\delta : Q \times \Gamma \to 2^{(Q \times \Gamma \times \{L,R\})} .$$

La notion de pas de calcul est adaptée immédiatement. Par exemple, on écrira :

$$waqaw' \vdash_M wbq'w' \quad \text{si } (q', b, R) \in \delta(q, a) .$$

La définition 2.1.3 de langage semi-décidable et décidable s'applique directement aux MdT non-déterministes. [3] On remarquera que pour qu'une entrée w soit acceptée il suffit qu'il existe un calcul qui mène de la configuration initiale à l'état q_a.

Proposition 2.3.3 (simulation du non-déterminisme) *Soit N une MdT non-déterministe. On peut construire une MdT standard M qui simule N. Si la complexité de N est $t(n) \geq n$ la complexité de M est $2^{O(t(n))}$.* [4]

3. Ce n'est pas le cas pour la notion de fonction partielle récursive car il faut décider d'abord quel est le résultat d'une MdT non-déterministe...

4. La notation O et la notion de complexité asymptotique qu'on utilise ici sont celles du cours d'algorithmique et elles seront rappélées dans la section 5.1.

IDÉE DE LA PREUVE. Dans une MdT non-déterministe N il y a une constante k qui borne le nombre d'alternatives possibles dans la suite du calcul. Ainsi on peut représenter le calcul d'une MdT non-déterministe comme un arbre éventuellement infini mais dont le branchement est borné par la constante k.

Les noeuds de cet arbre correspondent à des mots sur $\{0, \ldots, k-1\}^*$. On peut énumérer tous les noeuds de l'arbre en explorant l'arbre en largeur d'abord :

$$\epsilon, 0, \ldots, k-1, 00, \ldots, 0(k-1), 10, \ldots, 1(k-1), \ldots (k-1)0, \ldots, (k-1)(k-1), 000, \ldots$$

Une MdT peut calculer le successeur immédiat d'un mot π par rapport à cette énumération.

Pour simuler la machine N on utilise une MdT M avec 3 rubans. La proposition 2.3.1 nous assure qu'on peut toujours remplacer M par une MdT standard. Le premier ruban de M contient l'entrée w, le deuxième contient le chemin de l'arbre π qui est actuellement exploré et le troisième contient le ruban de la machine N lorsqu'elle calcule en effectuant les choix selon le chemin π.

Pour un chemin donné π, la machine M copie l'entrée du premier ruban au troisième et effectue ensuite un calcul en simulant l'exécution de N sur le chemin π.

— Le calcul peut bloquer car le chemin π ne correspond pas à un choix possible. Dans ce cas on considère le successeur immédiat de π et on itère.
— Le calcul arrive à la fin du chemin π mais la machine ne se trouve pas dans l'état q_a. Dans ce cas aussi on considère le successeur immédiat de π et on itère.
— Le calcul arrive à la fin du chemin π et la machine se trouve dans l'état q_a. Dans ce cas on accepte et on arrête le calcul.
— La simulation peut aussi noter qu'il ne reste plus de chemins à explorer et dans ce cas elle s'arrête et refuse.

Si la complexité de N est $t(n)$, la taille des chemins à considérer est aussi $O(t(n))$. Le nombre de chemins à simuler est $2^{O(t(n))}$. Donc la complexité de M est $2^{O(t(n))}$. Enfin, la MdT standard qui simule M est aussi $2^{O(t(n))}$ car $(2^{cn})^2$ est $2^{O(n)}$. □

Machines à compteurs

Une alternative à la manipulation du ruban est de considérer des machines qui disposent d'un certain nombre de compteurs (ou registres) qui contiennent des nombres naturels de taille arbitraire. Le fait de manipuler des *nombres* plutôt que des *mots* n'affecte pas vraiment la théorie car l'ensemble des mots sur un alphabet peut être mis en correspondance bijective avec l'ensemble des nombres naturels et ces bijections sont calculables de façon efficace. Le contrôle du programme est constitué d'une liste finie d'instructions élémentaires (ou états). Typiquement on dispose d'instructions pour :

— incrémenter un compteur et sauter à une autre instruction,
— décrementer un compteur (s'il contient un nombre positif) et sauter à une autre instruction et
— sauter à une instruction si un compteur contient zéro et à une autre sinon (un branchement conditionnel).

Il a été montré qu'il faut au moins 2 compteurs pour simuler le comportement d'une MdT et pour cette raison on parle aussi de *machines à 2 compteurs*. Une configuration d'une machine à 2 compteurs est donc un triplet (q, n, m) où q est l'instruction à exécuter et n et m sont les nombres naturels contenus dans les 2 registres.

Fonctions primitives et générales récursives

Une approche assez différente (mais équivalente) à la définition de fonction calculable (et donc de langage semi-décidable) consiste à supposer que :

— certaines fonctions primitives sur les nombres naturels sont évidemment calculables et
— les fonctions calculables sont stables par rapport à certains mécanismes de définition de fonctions.

En particulier, on peut prendre comme fonctions primitives la *fonction constante* 0, les *projections* et la fonction *successeur*. Plus formellement, on stipule que les fonctions suivantes sont récursives :

$$\begin{array}{llll} z_n : \mathbf{N}^n \to \mathbf{N} & n \geq 0 & z_n(x_1, \ldots, x_n) = 0 & \text{(zéro)} \\ p_{n,j} : \mathbf{N}^n \to \mathbf{N} & 1 \leq j \leq n & p_{n,j}(x_1, \ldots, x_n) = x_j & \text{(projections)} \\ s : \mathbf{N} \to \mathbf{N} & & s(x) = x + 1 & \text{(successeur).} \end{array}$$

Et on utilise les mécanismes de définition suivants :

Composition Si $f : \mathbf{N}^m \to \mathbf{N}$ et $g_i : \mathbf{N}^n \to \mathbf{N}$, $i = 1, \ldots, m$ sont récursives alors la fonction $h : \mathbf{N}^n \to \mathbf{N}$ suivante définie par *composition* est récursive aussi :

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)) \ .$$

Récursion primitive Si $f : \mathbf{N}^m \to \mathbf{N}$ et $g : \mathbf{N}^{m+2} \to \mathbf{N}$ sont récursives alors la fonction $h : \mathbf{N}^{m+1} \to \mathbf{N}$ suivante définie par *récursion primitive* est récursive aussi :

$$\begin{array}{ll} h(0, x_1, \ldots, x_m) & = f(x_1, \ldots, x_m) \ , \\ h(n+1, x_1, \ldots, x_m) & = g(n, h(n, x_1, \ldots, x_m), x_1, \ldots, x_m) \ . \end{array}$$

Minimisation Si $f : \mathbf{N}^{m+1} \to \mathbf{N}$ est une fonction récursive alors la fonction $g : \mathbf{N}^m \to \mathbf{N}$ suivante est récursive : $g(x_1, \ldots, x_m)$ calcule $f(0, x_1, \ldots, x_m)$, $f(1, x_1, \ldots, x_m)$,... jusqu'à trouver un x tel que $f(x, x_1, \ldots, x_m) = 0$ et dans ce cas elle termine et retourne x. Si un tel x existe n'existe pas ou si un des appels à f diverge alors g diverge.

La construction de minimisation introduit donc la possibilité de définir des fonctions partielles. Par contre, sans le mécanisme de minimisation on obtient une classe de fonctions *totales* qu'on appelle fonctions *primitives* récursives. Cette classe contient les fonctions arithmétiques usuelles : somme, produit, exposant, factoriel,... mais on peut exhiber des fonctions comme celle de Ackermann [Ack28], qui sont totales et calculables (par une MdT) mais qui ne sont pas primitives récursives. En ajoutant la minimisation, on obtient une classe de fonctions dites *générales récursives* qui modulo un codage des mots comme nombres coïncide avec la classe des fonctions calculables par une MdT (à savoir, les fonctions partielles récursives de la définition 2.1.7).

Thèse de Church-Turing

On peut prendre les MdT comme le formalisme qui permet de *définir* quand un langage est semi-décidable (ou une fonction est partielle récursive). Une telle définition est mathématiquement correcte et relativement simple à formaliser. Par ailleurs, le fait que le calcul d'une MdT est *effectif* semble incontestable : une personne (ou une machine électronique) peut simuler le calcul d'une MdT à condition de disposer d'une quantité de papier (d'une quantité de mémoire) qui peut être étendue indéfiniment.

Ceci dit, il y a un point qui n'est pas très satisfaisant : d'une certaine façon la définition est trop concrète car l'utilisation des MdT n'est pas du tout essentielle : on peut choisir aussi bien les machines à compteurs, ou les fonctions générales récursives, ou tout langage de programmation générale (C, Java, Python, OCaml, ...) à condition de ne pas borner la mémoire disponible pour son exécution. Ces formalismes et bien d'autres encore permettent de semi-décider exactement les mêmes langages que les MdT.

On est donc face à un phénomène fascinant : pour définir l'ensemble des langages semi-décidables on a besoin de fixer les détails d'un modèle d'exécution mais cet ensemble est extremement robuste et assez insensible aux détails du modèle d'exécution. Une façon de mettre en évidence ce phénomène est d'enoncer une *thèse* qui a été formulée d'abord dans les travaux de Church et de Turing et qui affirme que :

> Tout langage semi-décidable par une "procédure effective" est semi-décidable par une MdT.

D'un point de vue épistémologique, on peut comparer le statut de la thèse de Church-Turing à celui d'une loi physique qui ne peut pas être prouvée au sens mathématique mais seulement falsifiée par une expérience qui consisterait, par exemple, à présenter un nouveau type d'ordinateur capable de 'calculer' une fonction qui n'est pas calculable par une MdT.

La thèse a été formulée dans les années 1930 et pour l'instant elle n'a pas été falsifiée. En pratique, on l'utilise tout le temps dans la mesure où pour montrer qu'un langage est semi-décidable on se satisfait d'une description à haut niveau d'un procédé algorithmique sans rentrer dans les détails (bien pénibles) de sa programmation avec une MdT.

2.4 Langages indécidables

On rappelle qu'il y a une bijection $\langle \_, \_ \rangle$ entre les mots finis et les couples de mots finis et que tout mot w peut être vu comme la représentation d'une MdT $\varphi(w)$. En particulier, on utilise la notation $M, M', \ldots$ pour des mots qui sont considérés comme des MdT. On écrit aussi $\varphi(M)(w)$ pour indiquer le résultat du calcul de la MdT représentée par $\varphi(M)$ sur une entrée w ; en particulier, on écrit $\varphi(M)(w)\downarrow$ si ce calcul termine.

Définition 2.4.1 (problème de l'arrêt) *Le langage H est défini par :*

$$H = \{\langle M, w\rangle \mid \varphi(M)(w)\downarrow\} \ .$$

Le langage H formalise un problème intéressant qu'on appelle *problème de l'arrêt* : étant donné une MdT (un programme) M et une entrée w on se demande si le calcul de M sur l'entrée w termine.

On peut aussi considérer le comportement d'une machine M lorsque elle reçoit comme entrée le codage d'une machine M'. En particulier, on peut s'intéresser au résultat de l'application de la machine M à son propre codage.

Définition 2.4.2 (diagonalisation du problème de l'arrêt) *Le langage K est défini par :*

$$K = \{M \mid \varphi(M)(M)\downarrow\} \ .$$

On va montrer que les langages H et K *ne sont pas* décidables. Au passage, par la proposition 2.3.2 cela implique que les langages complémentaires H^c et K^c ne sont même pas semi-décidables.

Proposition 2.4.3 (un langage indécidable) *Le langage K n'est pas décidable.*

IDÉE DE LA PREUVE. Si K est décidable il doit y avoir une MdT $\varphi(M)$ telle que :

$$\varphi(M)(M') \downarrow \text{ ssi } M' \in K^c \ .$$

Si on applique $\varphi(M)$ à M on a deux possibilités :

1. Si $\varphi(M)(M) \downarrow$ alors $M \in K^c$ et donc $\varphi(M)(M) \not\downarrow$.
2. Si $\varphi(M)(M) \not\downarrow$ alors $M \notin K^c$ et donc $\varphi(M)(M) \downarrow$.

Les deux possibilités mènent à une contradiction, donc K^c n'est pas semi-décidable (on appelle cette technique de preuve *diagonalisation).* □

Plutôt que démontrer directement que H n'est pas décidable on va introduire une technique pour *réduire* l'analyse d'un langage à l'analyse d'un autre langage.

Définition 2.4.4 (réduction) *Soient L, L' deux langages sur un alphabet Σ. On dit que L se réduit à L' et on écrit $L \leq L'$ s'il existe une fonction récursive (totale) $f : \Sigma^* \to \Sigma^*$ telle que*

$$w \in L \text{ ssi } f(w) \in L' \ .$$

Si $L \leq L'$ alors les méthodes de décision qu'on développe pour L' peuvent être appliquées à L aussi.

Proposition 2.4.5 (réduction et (semi-)décidabilité) *Si $L \leq L'$ et L' est semi-décidable (décidable) alors L est semi-décidable (décidable).*

IDÉE DE LA PREUVE. On sait qu'il existe une fonction récursive f telle que $w \in L$ ssi $f(w) \in L'$. Supposons que M_f soit une MdT qui calcule f et M' une MdT qui semi-décide L'. Pour semi-décider (décider) L il suffit de composer M' et M_f. □

Exemple 2.4.6 *On obtient que $K \leq H$ en utilisant la fonction $f(M) = \langle M, M \rangle$. Comme K n'est pas décidable, H ne peut pas être décidable non plus.*

L'indécidabilité du problème de l'arrêt est la pointe d'un iceberg !

Définition 2.4.7 (équivalence extensionnelle de MdT) *On dit que deux MdT sont extensionnellement équivalentes si elles terminent sur les mêmes entrées en donnant la même réponse (accepter/refuser).* [5]

Définition 2.4.8 (propriété extensionnelle) *On dit qu'un langage $P \subseteq \Sigma^*$ est une propriété* extensionnelle *si P ne distingue pas les codages de deux machines qui sont extensionnellement équivalentes.* [6] *On dit aussi que P est triviale si P ou P^c est l'ensemble vide.*

5. Il y a des variations possibles de cette définition. Par exemple, on peut dire que les machines sont équivalentes si elles calculent la même fonction partielle ou alors qu'elles sont équivalentes si elles terminent sur les mêmes entrées.

6. En d'autres termes, si M et M' sont extensionnellement équivalentes alors soit $\{M, M'\} \subseteq P$ soit $\{M, M'\} \cap P = \emptyset$.

Proposition 2.4.9 ([Ric53]) *Toute propriété extensionnelle non triviale est indécidable.*

IDÉE DE LA PREUVE. Soit P une propriété extensionnelle telle que $P \neq \emptyset$ et $P^c \neq \emptyset$. Soit $M_\emptyset$ le codage d'une MdT qui boucle sur toute entrée. Supposons que $M_\emptyset \notin P$ (autrement on montre que P^c est indécidable). Supposons aussi que $M_1 \in P$. Soit f la fonction qui associe au codage d'une MdT M le codage d'une MdT qui reçoit une entrée w, calcule $\varphi(M)(M)$ et si elle termine calcule $M_1(w)$. La machine $f(M)$ est extensionnellement équivalente à M_1 (et donc appartient à P) si et seulement si $M \in K$. Donc la fonction f montre que $K \leq P$. □

Une conséquence de la proposition 2.4.9 est qu'il ne peut pas y avoir un langage de programmation dans lequel on peut programmer exactement les fonctions totales. Il ne serait pas décidable de savoir si un programme de ce langage est bien formé. Il est donc nécessaire de donner des critères décidables qui assurent la terminaison mais qui excluent certains programmes qui terminent (par exemple les fonctions primitives récursives de la section 2.3). Une autre conséquence est qu'on ne peut pas automatiser le problème de l'équivalence de deux programmes. Dans ce cas aussi on est amené à faire des approximations.

Exemple 2.4.10 (problèmes indecidables) *On termine en mentionnant (sans preuve) quelques problèmes indécidables remarquables.*

Problème de correspondance de Post : *soit Σ un alphabet et soit $(v_1, w_1) \cdots (v_k, w_k)$ une suite finie de couples de mots dans Σ^*. Le* problème de correspondance de Post *(PCP) consiste à déterminer s'ils existent $n \geq 1$ et $i_1, \ldots, i_n \in \{1, \ldots, k\}$ tels que :*

$$v_{i_1} \cdots v_{i_n} = w_{i_1} \cdots w_{i_n} \ .$$

Par exemple, trouvez une solution pour le problème $\{(a, baa), (ab, aa), (bba, bb)\}$. En général, on ne peut pas concevoir un algorithme qui pour tout PCP décide si le problème a une solution. En d'autres termes, le problème de correspondance de Post est indécidable.

Dixième problème de Hilbert : *soit $p(x_1, \ldots, x_n)$ un polynôme de degré arbitraire avec variables $x_1, \ldots, x_n$ et avec coefficients dans $\mathbf{Z}$. Par exemple, $p(x, y, z) = 6x^3yz^2 + 3xy^2 - x^3 - 10$. Le* dixième problème de Hilbert *consiste à déterminer si le polynôme p a des racines dans $\mathbf{Z}$, c'est-à-dire :*

$$\exists\, x_1, \ldots, x_n \in \mathbf{Z} \ \ p(x_1, \ldots, x_n) = 0 \ .$$

Ce problème a été proposé comme un challenge parmi d'autres en 1900 par D. Hilbert [Hil00] et il a été montré indécidable par Matijasevich en 1970 [Mat93]. Il est essentiel ici de considérer des polynôme à plusieurs variables car le problème avec une seule variable est décidable. Par ailleurs, il est remarquable que le même problème sur les réels *est décidable [Tar48].*

Validité en logique : *la logique du premier ordre, qui sera introduite dans le chapitre 4, est une source de problèmes indécidables. Par exemple, on peut montrer que l'ensemble des formules* valides*, c'est-à-dire vraies dans toute interprétation, est indécidable (proposition 4.4.1). On peut aussi fixer une interprétation naturelle comme celle des nombres naturels avec les opérations arithmétiques d'addition et de multiplication et montrer que l'ensemble des formules vraies dans cette interprétation est indécidable (proposition 4.4.3).*

Exercices

Exercice 14 (définition MdT) *Examinez la définition 2.1.1 d'une MdT et répondez aux questions suivantes.*

1. *Une MdT peut-elle écrire le symbole $\sqcup$ sur le ruban ?*
2. *L'alphabet d'entrée et du ruban peuvent-ils être égaux ?*
3. *La tête de lecture peut-elle rester au même endroit pendant deux étapes consécutives ?*
4. *Une MdT peut-elle contenir un seul état ?*

Exercice 15 (programmation MdT)

1. *Programmez une MdT M déterministe avec alphabet d'entrée $\Sigma = \{(0,0),(0,1),(1,0),(1,1)\}$ et alphabet du ruban $\Gamma = \Sigma \cup \{\sqcup, 0, 1\}$ qui a la propriété suivante : à partir de la configuration initiale $q_0(x_{n-1}, y_{n-1}) \cdots (x_0, y_0)$, M va parcourir l'entrée de gauche à droite et la remplacer par $z_{n-1} \cdots z_0$ où :*

$$(z_{n-1} \cdots z_0)_2 = max\{(x_{n-1} \cdots x_0)_2, (y_{n-1} \cdots y_0)_2\} \ ,$$

   et s'arrêter dans un état accepteur q_a. En d'autres termes, M calcule le maximum des entrées avec une représentation des nombres en base 2.
2. *Programmez une MdT avec alphabet d'entrée $\Sigma = \{0, 1, \sharp\}$ qui a la propriété suivante : à partir d'une configuration initiale $q_0 \sharp w$ où w est un mot fini composé de 0 et 1 la machine s'arrête dans un état accepteur q_a avec un ruban qui contient le mot $\sharp\sharp w$. En d'autres termes, la fonction de la machine est de décaler d'une case vers la droite le mot w en insérant le symbole $\sharp$ dans la case qui est ainsi libérée.*
3. *Donnez la description formelle d'une MdT qui décide le langage $\{w \sharp w \mid w \in \{0,1\}^*\}$.*
4. *Donnez la description formelle d'une MdT qui décide le langage des mots sur l'alphabet $\{0\}$ dont la longueur est une puissance de 2 : 2^0, 2^1, 2^2, ...*
5. *Décrivez informellement une MdT qui décide le langage : $\{a^i b^j c^k \mid i \cdot j = k$ et $i, j, k \geq 1\}$.*

Exercice 16 (énumérations)

1. *On peut énumérer les couples de nombres naturels en procédant 'par diagonales' :*

$$(0,0), \quad (1,0),(0,1), \quad (2,0),(1,1),(0,2), \quad (3,0) \ldots$$

   Montrez que la fonction $\langle m, n\rangle = (m+n)(m+n+1)/2 + n$ est une bijection *entre $\mathbf{N} \times \mathbf{N}$ et $\mathbf{N}$. Décrivez un algorithme pour calculer la fonction inverse.*
2. *On définit les fonctions $\langle \_ \rangle_k : \mathbf{N}^k \to \mathbf{N}$ pour $k \geq 2$:*

$$\langle m, n\rangle_2 \ = \langle m, n\rangle \ , \quad \langle n_1, \ldots, n_k\rangle_k \ = \langle \langle n_1, \ldots, n_{k-1}\rangle_{k-1}, n_k\rangle \text{ si } k \geq 3.$$

   Montrez que les fonctions $\langle \_, \ldots, \_ \rangle_k$ sont des bijections.
3. *On considère l'ensemble $\mathbf{N}^*$ des mots finis de nombres naturels. Notez que $\mathbf{N}^*$ est en correspondance bijective avec $\bigcup_{k \geq 0} \mathbf{N}^k$. Définissez une bijection entre $\mathbf{N}^*$ et $\mathbf{N}$.*
4. *Soit $\Sigma = \{a, b, \ldots, z\}$ un alphabet. On peut énumérer les éléments de Σ^* comme suit :*

$$\epsilon, \quad a, b, \ldots, z, \quad aa, \ldots, az, ba, \ldots, bz, za, \ldots, zz, \quad aaa, \ldots$$

   Si Σ contient k éléments on aura k^0 mots de longueur 0, k mots de longueur 1, k^2 mots de longueur 2, ... Définissez une bijection entre Σ^* et $\mathbf{N}$.

Exercice 17 (Cantor) *1. Rappel : tout nombre naturel $n \geq 2$ admet une décomposition unique comme produit $p_1^{n_1} \cdots p_k^{n_k}$ où $k \geq 1$, $p_1 < \cdots < p_k$ sont des nombres premiers et $n_1, \ldots, n_k \geq 1$. En utilisant ce fait, définissez une fonction surjective de $\mathbf{N}$ dans les parties finies de $\mathbf{N}$.*

2. *On ne peut pas généraliser aux parties de $\mathbf{N}$! Ce résultat dû à Cantor utilise une technique dite de* diagonalisation *similaire à celle rencontrée dans la proposition 2.4.3. Supposez une énumération $e : \mathbf{N} \to 2^{\mathbf{N}}$. Considérez $X = \{n \mid n \notin e(n)\}$. Comme e est surjective, il existe n_X tel que $e(n_X) = X$ et soit $n_X \in X$ soit $n_X \notin X$. Montrez que dans les deux cas on arrive à une contradiction.*
3. *On dit qu'un ensemble X est dénombrable s'il y a une fonction bijective entre X et les nombres naturels $\mathbf{N}$. Montrez que l'ensemble des langages sur un alphabet Σ n'est pas dénombrable ; concluez qu'il y a des langages qui ne sont pas semi-décidables.*

Exercice 18 (semi-décidable et récursivement énumérable) *On dit qu'un langage $L \subseteq \Sigma^*$ est* récursivement énumérable *s'il y a une fonction partielle récursive f dont L est l'image, c'est-à-dire, L est l'ensemble des mots produits par f en sortie quand la fonction termine. Montrez que :*

1. *un langage est semi-décidable ssi il est l'ensemble des mots sur lesquels une MdT termine (dans q_a ou q_r) et*
2. *un langage L est récursivement énumérable si et seulement si il est semi-décidable.*

Exercice 19 (décidable ou pas) *Montrez ou donnez un contre-exemple aux assertions suivantes.*

1. *Il y a une MdT qui accepte les mots sur l'alphabet $\{0, 1\}$ qui contiennent autant de 0 que de 1 (si la MdT existe, il suffira d'en donner une description informelle).*
2. *Rappel : si A et B sont deux langages, on écrit $A \leq B$ s'il existe une réduction de A à B. Si A est semi-décidable et $A \leq A^c$ alors A est décidable.*
3. *L'ensemble des (codages de) MdT qui semi-décident un langage fini est décidable.*
4. *L'ensemble des (codages de) MdT qui terminent sur le mot vide est décidable.*
5. *L'ensemble des (codages de) MdT qui divergent sur le mot vide est semi-décidable.*
6. *L'ensemble des (codages de) MdT qui terminent sur le mot vide en 10^{100} pas de calcul est décidable.*
7. *L'ensemble Tot des (codages de) MdT qui terminent sur toute entrée est décidable.*
8. *L'ensemble Eq des (codages de) couples de MdT qui sont extensionnellement équivalentes.*

Exercice 20 (mission impossible) *Le chef du service programmation systèmes embarqués a un petit problème. Les programmes de contrôle sont conçus dans un langage de haut niveau (type* C*) et ensuite compilés vers du code assembleur. Le problème est que très souvent le code assembleur généré ne rentre pas dans la mémoire du processeur embarqué... Pour régler la question, il envisage de construire un* optimiseur *qui prend en entrée un programme assembleur (arbitraire) et produit en sortie un programme avec le même comportement et avec un nombre* minimum *d'instructions. Proposez un argument qui montre qu'il s'agit d'une* mission impossible*. Dans votre argument, vous pouvez faire l'hypothèse que le code assembleur prend la forme des machines de Turing décrites dans ce chapitre. Addenda : il va sans dire que ce n'est pas forcement une bonne idée de ridiculiser le chef du dit service... et qu'il faudra beaucoup de tact et de diplomatie pour le mettre sur la bonne piste...*

Chapitre 3

Calcul propositionnel

Ce chapitre introduit les rudiments du calcul propositionnel. On suppose que le lecteur est déjà familier avec une grande partie des notions traitées car elles sont utilisées couramment dans la conception de circuits combinatoires et plus en général dans la pratique de la programmation et des mathématiques. Dans ce contexte, les objectifs principaux sont de fixer les notations qui seront reprises dans le cadre plus compliqué du calcul des prédicats et d'introduire une propriété fondamentale dite de *compacité* qui sera utilisée pour prouver que les formules valides de la logique du premier ordre sont récursivement énumérables (proposition 4.3.4).

3.1 Algèbre initiale

Un zeste d'algèbre va nous permettre de préciser les notions de 'syntaxe' et de 'sémantique' (ou 'interprétation de la syntaxe').

Définition 3.1.1 (signature) *Une signature est un couple* (Σ, ar) *où* Σ *est un ensemble et* $ar : \Sigma \to \mathbf{N}$ *est un fonction qui associe à chaque élément de* Σ *un nombre naturel.*

On peut penser aux éléments de Σ comme à des *symboles* de fonction et à ar comme la fonction qui associe à chaque symbole de fonction son *arité*, c'est-à-dire le nombre d'arguments attendus par le symbole de fonction. Pour indiquer un symbole de fonction f avec $ar(f) = n$, on écrira aussi f^n.

Exemple 3.1.2 *Voici quelques exemples de signatures avec des dénominations qui seront justifiées dans la suite (exemple 3.1.9).*

$\{z^0, s^1\}$	*signature des nombres unaires*
$\{\epsilon^0, a^1, b^1, \ldots, z^1\}$	*signature des mots finis sur* $\{a, b, \ldots, z\}$
$\{nil^0, b^2\}$	*signature des arbres binaires (ordonnés et enracinés)*
$\{\emptyset^0, \epsilon^0, a^0, \ldots, z^0, *^1, +^2, \cdot^2\}$	*signature des expressions rationnelles sur* $\{a, \ldots, z\}$
$\{\neg^1, \wedge^2, \vee^2, x^0, y^0, z^0, \ldots\}$	*signature des formules du calcul propositionnel.*

Définition 3.1.3 (Σ-algèbre) *Soit* (Σ, ar) *une signature. Une* Σ*-algèbre est composée d'un ensemble* A *et d'un ensemble de fonctions* $\{f_s : A^{ar(s)} \to A \mid s \in \Sigma\}$*. Dans une* Σ*-algèbre on a donc un ensemble et une fonction pour chaque symbole de la signature.*

Exemple 3.1.4 *Considérons la signature* $\Sigma = \{z^0, s^1\}$. *On peut construire une* Σ*-algèbre en prenant l'ensemble des nombres naturels avec la fonction constante* $f_z = 0$ *et la fonction unaire* $f_s(x) = x + 2$. *Une autre* Σ*-algèbre pourrait être l'ensemble des nombres réels avec la fonction constante* $g_z = 1$ *et la fonction unaire* $g_s(x) = 3 \cdot x$.

Remarque 3.1.5 *Si l'ensemble qui compose une* Σ*-algèbre est vide alors la signature* Σ *ne contient pas de symboles d'arité* 0.

Définition 3.1.6 (morphisme) *Soient* $(A, \{f_s \mid s \in \Sigma\})$ *et* $(B, \{g_s \mid s \in \Sigma\})$ *deux* Σ*-algèbres. On dit que la fonction* $h : A \to B$ *est un morphisme si elle commute avec les opérations de l'algèbre, à savoir pour tout* $s \in \Sigma$ *tel que* $ar(s) = n$ *et pour tout* $a_1, \ldots, a_n \in A$ *on a :*

$$h(f_s(a_1, \ldots, a_n)) = g_s(h(a_1), \ldots, h(a_n)) \ .$$

Exemple 3.1.7 *On reprend les deux* Σ*-algèbres de l'exemple 3.1.4. Le lecteur peut vérifier que la fonction* $h : \mathbf{N} \to \mathbf{R}$ *suivante est un morphisme :*

$$h(n) = \begin{cases} 3^k & \textit{si } n = 2 \cdot k \\ 0 & \textit{autrement.} \end{cases}$$

Il y a une façon *canonique* de construire une Σ-algèbre qu'on appelle Σ-algèbre *initiale*; cette dénomination est justifiée par la proposition 3.1.10.

Définition 3.1.8 (Σ-algèbre initiale) *Soit* (Σ, ar) *une signature. On définit les ensembles :*

$$\begin{array}{ll} T_0 & = \{s \in \Sigma \mid ar(s) = 0\}, \\ T_{n+1} & = T_n \cup \{(s, t_1, \ldots, t_m) \mid s \in \Sigma, ar(s) = m \geq 1, t_i \in T_n, i = 1, \ldots, m\}, \\ T_\Sigma & = \bigcup_{n \geq 0} T_n \ . \end{array}$$

Soit $s \in \Sigma$. *Si* $ar(s) = 0$ *on définit la (fonction) constante* $\underline{s} = s \in T_\Sigma$. *Et si* $ar(s) = m \geq 1$ *on définit une fonction* $\underline{s} : (T_\Sigma)^m \to T_\Sigma$ *par :*

$$\underline{s}(t_1, \ldots, t_m) = (s, t_1, \ldots, t_m) \ .$$

Exemple 3.1.9 *On explicite quelques éléments de l'ensemble* T_Σ *pour les signatures* Σ *introduites dans l'exemple 3.1.2.*

Signature	T_Σ
Nombres unaires	$\{z, (s, z), (s, (s, z)), (s, (s, (s, z))), \ldots\}$
Mots finis	$\{\epsilon, (a, \epsilon), \ldots, (z, \epsilon), (a, (a, \epsilon)), \ldots\}$
Arbres binaires	$\{nil, (b, nil, nil), (b, (b, nil, nil), nil), (b, nil, (b, nil, nil)), \ldots\}$
Expressions rationnelles	$\{\emptyset, \epsilon, a, \ldots, (+, \epsilon, \emptyset), \ldots, \}$
Formules propositionnelles	$\{x, \ldots, (\neg, x), \ldots, (\wedge, x, y), \ldots\}$

Proposition 3.1.10 (unicité morphisme de l'algèbre initiale) *Soit* $\underline{A} = (A, \{f_s \mid s \in \Sigma\})$ *une* Σ*-algèbre. Alors il existe un unique morphisme de la* Σ*-algèbre initiale dans* $\underline{A}$.

Idée de la preuve. D'après la définition 3.1.8 d'algèbre initiale, pour tout $x \in T_\Sigma$ on peut définir :

$$rang(x) = min\{n \in \mathbf{N} \mid x \in T_n\} \ .$$

On montre que pour tout $x \in T_\Sigma$, il y a une seule définition possible de $h(x)$. On procède par récurrence sur $rang(x)$. Si $rang(x) = 0$ alors $x = s \in \Sigma$, $ar(s) = 0$ et on doit avoir $h(\underline{s}) = h(s) = f_s$. Si $rang(x) = n + 1$ alors $x = (s, t_1, \ldots, t_m)$, $ar(s) = m \geq 1$, $rang(t_i) \leq n$ pour $i = 1, \ldots, m$ et on doit avoir :

$$h(\underline{s}(t_1, \ldots, t_m)) = h(s, t_1, \ldots, t_m) = f_s(h(t_1), \ldots, h(t_m)) \ .$$

□

Exemple 3.1.11 *Pour définir un morphisme sur une Σ-algèbre initiale il suffit de fixer l'interprétation des symboles de la signature. Considérons les Σ-algèbres initiales de l'exemple 3.1.9. On peut définir la fonction qui associe un nombre naturel à un nombre en représentation unaire en considérant l'ensemble des nombres naturels avec fonctions $f_z = 0$ et $f_s(x) = x+1$. La longueur d'un mot correspond à la Σ-algèbre sur les nombres naturels où $f_\epsilon = 0$ et $f_a(x) = \cdots = f_z(x) = x + 1$. De façon similaire, la fonction qui calcule la hauteur d'un arbre binaire correspond à la Σ-algèbre sur les nombres naturels où $f_{nil} = 0$ et $f_b(x, y) = 1 + max(x, y)$. Pour construire un morphisme qui associe un langage à chaque expression rationnelles, on procède comme expliqué dans la définition 1.3.2.*

On peut associer une valeur à chaque formule du calcul propositionnel en prenant l'ensemble des valeurs booléennes $\mathbf{2} = \{0, 1\}$. On suppose disposer d'une fonction $v : V \to \mathbf{2}$ des variables aux valeurs booléennes qui spécifie la valeur de chaque variable. Il reste à définir les fonctions NOT, AND et OR associées aux symboles $\neg$, $\wedge$ et $\vee$, respectivement. Ces fonctions sont spécifiées par les tableaux suivants :

x	$NOT(x)$
0	1
1	0

x	y	$AND(x,y)$
0	0	0
0	1	0
1	0	0
1	1	1

x	y	$OR(x,y)$
0	0	0
0	1	1
1	0	1
1	1	1

En changeant de Σ-algèbre, on peut trouver d'autres interprétations intéressantes des formules propositionnelles. Par exemple, on peut associer à chaque formule sa taille, à savoir le nombre de symboles qu'elle contient. Dans ce cas, on prend comme domaine d'interprétation l'ensemble des nombres naturels, on associe aux variables le nombre naturel 1 et on définit les fonctions $f_\neg(x) = x + 1$ et $f_\wedge(x, y) = f_\vee(x, y) = 1 + x + y$. Dans une autre direction, on peut fixer un ensemble E et interpréter les variables comme des sous-ensembles de E et les symboles $\neg$, $\wedge$ et $\vee$ comme le complémentaire, l'intersection et l'union, respectivement.

3.2 Syntaxe et sémantique

La *logique* est à l'origine une réflexion sur le discours (*logos*) et sur sa cohérence. En particulier, la logique *mathématique* s'intéresse à l'organisation et à la cohérence du discours mathématique et donc aux notions de *validité* et de *preuve*. Dans le *calcul propositionnel classique*, on dispose d'un certain nombre de *propositions* qui peuvent être vraies ou fausses et d'un certain nombre d'opérateurs qui permettent de combiner ces propositions.

On rappelle (section 3.1) que la syntaxe du calcul propositionnel est définie comme l' algèbre initiale sur la signature $\Sigma = V \cup \{\neg^1, \wedge^2, \vee^2\}$ où $V = \{x^0, y^0, \ldots\}$ est un ensemble

dénombrable de symboles de *variables* d'arité 0.[1] On appelle *formules* les éléments de T_Σ et on les dénote par les lettres $A, B, \ldots$ On choisit une variable x_0 (arbitraire mais fixée) et on utilise les abréviations suivantes :

$$\begin{array}{c} \neg A = (\neg, A), \qquad (A \wedge B) = (\wedge, A, B), \qquad (A \vee B) = (\vee, A, B), \\ \mathbf{1} = (x_0 \vee \neg x_0), \qquad \mathbf{0} = (x_0 \wedge \neg x_0)\ . \end{array} \tag{3.1}$$

Définition 3.2.1 (littéral) *Un* littéral *est une formule qui est une variable ou la négation d'une variable. Dans le premier cas on dit que le littéral est* positif *et dans le deuxième qu'il est* négatif. *On dénote un littéral avec* $\ell, \ell', \ldots$

Définition 3.2.2 (variables dans une formule) *L'ensemble* $var(A)$ *des variables présentes dans une formule* A *est défini par :*

$$var(x) = \{x\}, \quad var(\neg A) = var(A), \quad var(A \wedge B) = var(A \vee B) = var(A) \cup var(B).$$

Définition 3.2.3 (substitution) *Si* A, B *sont des formules et* x *est une variable alors on dénote par* $[B/x]A$ *la substitution de la variable* x *par la formule* B *dans la formule* A. *La fonction de substitution est définie sur* T_Σ *par :*

$$[B/x](y) = \begin{cases} B & \textit{si } y = x \\ y & \textit{autrement} \end{cases} \qquad [B/x](\neg A) = \neg[B/x]A,$$

$$[B/x](A \wedge A') = ([B/x]A \wedge [B/x]A')\ , \qquad [B/x](A \vee A') = ([B/x]A \vee [B/x]A')\ .$$

On verra dans la section 3.4 que les opérateurs $\neg$, $\vee$ et $\wedge$ suffisent à exprimer tous les autres. Cependant, un certain nombre d'opérateurs logiques sont utilisés assez souvent pour mériter un symbole spécifique.

Définition 3.2.4 (opérateurs dérivés)

$$\begin{array}{lll} (A \to B) & = (\neg A \vee B) & \textit{(implication)} \\ (A \leftrightarrow B) & = ((A \to B) \wedge (B \to A)) & \textit{(si et seulement si)} \\ (A \oplus B) & = ((A \wedge \neg B) \vee (\neg A \wedge B)) & \textit{(ou exclusif)} \\ (A \rightsquigarrow B, C) & = ((A \wedge B) \vee (\neg A \wedge C)) & \textit{(conditionnel).} \end{array}$$

L'interprétation standard des formules utilise les *valeurs booléennes* $\mathbf{2} = \{0, 1\}$ avec la convention que 0 correspond à 'faux' et 1 à 'vrai'. On rappelle (section 3.1) qu'une fois qu'on a fixé l'interprétation des variables et des symboles $\neg$, $\vee$ et $\wedge$, on a une fonction qui est définie de façon unique.

Définition 3.2.5 (interprétation) *L'interprétation d'une formule* A *par rapport à une affectation* $v : V \to \mathbf{2}$ *est la fonction (unique)* $[\![\_]\!]v$ *qui satisfait :*

$$\begin{array}{ll} [\![x]\!]v = v(x)\ , & [\![\neg A]\!]v = NOT([\![A]\!]v)\ , \\ [\![A \wedge B]\!]v = AND([\![A]\!]v, [\![B]\!]v)\ , & [\![A \vee B]\!]v = OR([\![A]\!]v, [\![B]\!]v)\ , \end{array}$$

où les fonctions NOT, AND, OR sont définies comme dans l'exemple 3.1.11.

1. La *syntaxe*, telle qu'on l'entend dans ce cours, est aussi appelée *syntaxe abstraite* dans le cadre de l'analyse syntaxique des langages. Un programme d'analyse syntaxique reçoit en entrée une suite de caractères et produit en sortie soit la syntaxe abstraite d'une phrase du langage soit un message d'erreur.

Définition 3.2.6 (mise à jour) *Si v est une affectation, x est une variable propositionnelle et $b \in \mathbf{2}$ est une valeur booléenne alors la mise à jour de v avec b pour x est définie par :*

$$v[b/x](y) = \begin{cases} b & \text{si } y = x \\ v(y) & \text{autrement.} \end{cases}$$

Notation On peut expliciter les valeurs d'une affectation en écrivant $[b_1/x_1, \ldots, b_n/x_n]$ où $x_i \neq x_j$ si $i \neq j$. Dans ce cas, il est entendu que les valeurs de l'affectation sur les variables différentes de $x_1, \ldots, x_n$ n'ont pas d'importance.

La proposition suivante met en relation la substitution au niveau syntaxique avec la mise à jour au niveau sémantique. Elle permet de remplacer les variables propositionnelles par des formules arbitraires.

Proposition 3.2.7 (substitution et mise à jour) *Soient A, B deux formules, x une variable et v une affectation. Alors :*

$$[\![[B/x]A]\!]v = [\![A]\!]v[[\![B]\!]v/x] \ .$$

Idée de la preuve. Par récurrence sur la *taille* (exemple 3.1.11) de A. □

On introduit maintenant 3 définitions fondamentales pour la suite du cours. On écrit $v \models A$ si $[\![A]\!]v = 1$ et $\models A$ si pour tout v, $v \models A$.

Définition 3.2.8 (validité) *Une formule A est valide (on dit aussi qu'elle est une* tautologie*) si $\models A$, c'est-à-dire si pour toute affectation v on a $v \models A$.*

Définition 3.2.9 (satisfaisabilité) *Une formule A est satisfaisable s'il existe une affectation v telle que $v \models A$.*

Définition 3.2.10 (équivalence logique) *Deux formules A et B sont logiquement équivalentes si pour toute affectation v on a $v \models A$ ssi $v \models B$. Dans ce cas on écrit $A \equiv B$.*

Par exemple, $\mathbf{1}$ est valide, x est satisfaisable mais pas valide, $\mathbf{0}$ n'est pas satisfaisable, x est équivalente à $x \wedge \mathbf{1}$ et n'est pas équivalente à $x \vee \mathbf{1}$. D'un point de vue mathématique, on peut prendre une de ces notions comme fondamentale et dériver les deux autres. [2] Même si les notions de validité, satisfaisabilité et équivalence logique sont d'une certaine façon interchangeables (voir exercice 23), chaque notion a ses propres méthodes algorithmiques.

3.3 Équivalence logique

Proposition 3.3.1 (équivalences remarquables) *Dans le calcul propositionnel, on dispose des équivalences logiques présentées dans la table 3.1.*

2. Quand on traite des formules, attention à ne pas confondre la relation $=$ (identité) avec la relation $\equiv$ (équivalence logique) ; la première est strictement contenue dans la deuxième.

$$
\begin{array}{ll}
(G1) & (x \vee \mathbf{0}) \equiv x, \quad (x \vee \mathbf{1}) \equiv \mathbf{1}, \quad (x \vee y) \equiv (y \vee x), \\
& ((x \vee y) \vee z) \equiv (x \vee (y \vee z)), \quad (x \vee x) \equiv x, \\
(G2) & (x \wedge \mathbf{0}) \equiv \mathbf{0}, \quad (x \wedge \mathbf{1}) \equiv x, \quad (x \wedge y) \equiv (y \wedge x), \\
& ((x \wedge y) \wedge z) \equiv (x \wedge (y \wedge z)), \quad (x \wedge x) \equiv x, \\
(G3) & (x \wedge y) \vee z \equiv (x \vee z) \wedge (y \vee z), \quad (x \vee y) \wedge z \equiv (x \wedge z) \vee (y \wedge z), \\
(G4) & \neg\neg x \equiv x, \quad \neg(x \vee y) \equiv (\neg x \wedge \neg y), \quad \neg(x \wedge y) \equiv (\neg x \vee \neg y)\ , \\
(G5) & (x \vee \neg x) \equiv \mathbf{1}, \quad (x \wedge \neg x) \equiv \mathbf{0}.
\end{array}
$$

TABLE 3.1 – Équivalences logiques

IDÉE DE LA PREUVE. En calculant les tables de vérité. □

Le groupe $(G1)$ dit que le $\vee$ est un opérateur commutatif, associatif et idempotent qui a $\mathbf{0}$ comme identité et $\mathbf{1}$ comme absorbant. Le groupe $(G2)$ dit que le $\wedge$ est aussi un opérateur commutatif, associatif et idempotent mais son identité est $\mathbf{1}$ et son absorbant est $\mathbf{0}$. Le groupe $(G3)$ permet de distribuer l'opérateur $\wedge$ ($\vee$) par rapport à l'opérateur $\vee$ ($\wedge$). Le groupe $(G4)$ affirme que la négation est un opérateur involutif et qu'il peut être distribué d'une certaine façon sur les opérateurs $\vee$ et $\wedge$ (lois de De Morgan). Avec ces équivalences on peut déduire $\neg\mathbf{0} \equiv \mathbf{1}$ et $\neg\mathbf{1} \equiv \mathbf{0}$. D'après la convention (3.1), les formules $\mathbf{1}$ et $\mathbf{0}$ dépendent d'une variable x_0 *fixée*. Le dernier groupe $(G5)$, nous permet de déduire, par exemple, que pour *toute* variable x, $(x \vee \neg x) \equiv \mathbf{1}$, aussi connue comme *loi du tiers exclu*.

Grâce à la proposition 3.2.7, on peut toujours remplacer une variable par une formule. Ainsi, si l'on veut montrer $(A \wedge A) \equiv A$ pour une formule A arbitraire, on fait appel à l'équivalence $(x \wedge x) \equiv x$ (groupe $(G2)$) et on note que pour toute affectation v :

$$[\![A \wedge A]\!]v \;\; = [\![[A/x](x \wedge x)]\!]v \;\; = [\![x \wedge x]\!]v[[\![A]\!]v/x] \;\; = [\![x]\!]v[[\![A]\!]v/x] \;\; = [\![A]\!]v\ .$$

Il est donc possible de pratiquer un *raisonnement équationnel* sur les formules du calcul propositionnel qui est similaire à celui que le lecteur a déjà pratiqué dans le cadre, par exemple, de la théorie des groupes. En effet, les équivalences de la table 3.1 forment la base d'une théorie équationnelle qu'on appelle *algèbre booléenne*.

3.4 Définissabilité et formes normales

Notation Si ℓ est un littéral, on prétendra parfois que $\neg\ell$ est aussi un littéral. Ceci est justifié par le caractère *involutif* de la négation, à savoir on peut toujours remplacer $\neg\neg x$ par x. Si $\{A_i \mid i \in I\}$ est une famille de formules indexées sur un ensemble fini I on peut écrire :

$$\textstyle\bigwedge\{A_i \mid i \in I\} \quad \text{ou} \quad \bigwedge_{i\in I} A_i, \qquad \text{et} \qquad \bigvee\{A_i \mid i \in I\} \quad \text{ou} \quad \bigvee_{i\in I} A_i\ .$$

Comme la disjonction et la conjonction sont associatives et commutatives, cette notation définit une formule unique à équivalence logique près. Par convention, si I est vide on a :

$$\textstyle\bigwedge \emptyset = \mathbf{1} \qquad \text{et} \qquad \bigvee \emptyset = \mathbf{0}\ . \tag{3.2}$$

Définition 3.4.1 (fonction définissable) *Soit A une formule et $x_1, \ldots, x_n$ une liste de variables distinctes telle que $var(A) \subseteq \{x_1, \ldots, x_n\}$. Alors la formule A définit une fonction $f_A : \mathbf{2}^n \to \mathbf{2}$ par :*

$$f_A(b_1, \ldots, b_n) = [\![A]\!][b_1/x_1, \ldots, b_n/x_n] \ .$$

Remarque 3.4.2 *Notez que la fonction f_A non seulement dépend de A mais aussi de la liste de variables $x_1, \ldots, x_n$. Par exemple, la formule x définit la première projection par rapport à la liste x, y et la deuxième projection par rapport à la liste y, x.*

Définition 3.4.3 (DNF) *On appelle* monôme *une conjonction de littéraux. Une formule est en forme normale disjonctive (DNF pour* Disjunctive Normal Form*) si elle est une disjonction de monômes.*

Définition 3.4.4 (CNF) *On appelle* clause *une disjonction de littéraux. Une formule est en forme normale conjonctive (CNF pour* Conjunctive Normal Form*) si elle est une conjonction de clauses.*

Proposition 3.4.5 (définissabilité par DNF) *Toute fonction $f : \mathbf{2}^n \to \mathbf{2}$, $n \geq 1$, est définissable par une formule A en forme normale disjonctive (DNF) telle que $var(A) \subseteq \{x_1, \ldots, x_n\}$.*

Idée de la preuve. On construit un tableau de vérité avec 2^n entrées. Si $f(b_1, \ldots, b_n) = 1$ avec $b_i \in \{0, 1\}$ alors on construit un monôme $(\ell_1 \wedge \cdots \wedge \ell_n)$ où $\ell_i = x_i$ si $b_i = 1$ et $\ell_i = \neg x_i$ autrement. La formule A est la disjonction de tous les monômes obtenus de cette façon. Par exemple, si $f(0,1) = f(1,0) = 1$ et $f(0,0) = f(1,1) = 0$ alors on obtient $A = (\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2)$. On note que si f est la fonction constante 0 alors on obtient une disjonction vide qui, par la convention (3.2), est $\mathbf{0}$. □

La formule DNF construite dans la proposition 3.4.5 est *unique* si l'on considère que conjonction et disjonction sont associatives et commutatives. Cependant, cette formule n'est pas forcement de taille *minimale* (considérez, par exemple, la fonction constante 1).

Remarque 3.4.6 *Les éléments d'un ensemble fini X peuvent être codés par les éléments de l'ensemble $\mathbf{2}^n$ pour n suffisamment grand (certains éléments peuvent avoir plusieurs codes). Toute fonction $f : \mathbf{2}^n \to \mathbf{2}^m$ se décompose en m fonctions $f_1 : \mathbf{2}^n \to \mathbf{2}, \ldots, f_m : \mathbf{2}^n \to \mathbf{2}$. Ainsi toute fonction $f : X \to Y$ où X et Y sont finis peut être définie, modulo codage, par un vecteur de formules du calcul propositionnel. Avec un peu de réflexion, tout objet fini peut être représenté par des formules du calcul propositionnel. Cette puissance de représentation explique en partie la grande variété d'applications possibles du calcul propositionnel.*

Tout ce qui a été dit de la forme DNF peut être transféré à la forme CNF 'par dualité'.

Corollaire 3.4.7 (définissabilité par CNF) *Toute fonction $f : \mathbf{2}^n \to \mathbf{2}$, $n \geq 1$, est définissable par une formule A en forme normale conjonctive (CNF) telle que $var(A) \subseteq \{x_1, \ldots, x_n\}$.*

IDÉE DE LA PREUVE. Par la proposition 3.4.5 on peut construire une formule A en forme normale disjonctive pour la fonction $NOT \circ f : \mathbf{2}^n \to \mathbf{2}$. Donc la formule $\neg A$ définit la fonction f. On applique maintenant les lois de De Morgan et on obtient :

$$\neg \bigvee_{i\in I}(\bigwedge_{j\in J_i} \ell_{i,j}) \equiv \bigwedge_{i\in I}(\neg(\bigwedge_{j\in J_i} \ell_{i,j})) \equiv \bigwedge_{i\in I}(\bigvee_{j\in J_i} (\neg\ell_{i,j})\ ) \equiv \bigwedge_{i\in I}(\bigvee_{j\in J_i} \ell'_{i,j}\ )\ ,$$

où $\ell'_{i,j} = \neg x_{i,j}$ si $\ell_{i,j} = x_{i,j}$ et $\ell'_{i,j} = x_{i,j}$ si $\ell_{i,j} = \neg x_{i,j}$. Bien sûr, on utilise ici l'équivalence logique $x \equiv \neg\neg x$. □

3.5 Compacité et conséquence logique

Définition 3.5.1 (ensemble satisfaisable) *Un ensemble (éventuellement infini) de formules T est satisfaisable s'il existe une affectation qui satisfait chaque formule dans T.*

Définition 3.5.2 (ensemble finiment satisfaisable) *Un ensemble T de formules est* finiment satisfaisable *si tout sous ensemble fini de T est satisfaisable.*

Il est immédiat de vérifier que si T est satisfaisable alors il est finiment satisfaisable ; on va montrer que la réciproque est aussi vraie. Cette propriété est connue comme *compacité.*[3]

Proposition 3.5.3 (compacité) *Si un ensemble de formules T est finiment satisfaisable alors il est satisfaisable.*

IDÉE DE LA PREUVE. Soit $x_1, x_2, \ldots$ une énumération des variables dans T et soit $T_n = \{A \in T \mid var(A) \subseteq \{x_1, \ldots, x_n\}\}$, pour $n \geq 0$.

1. On vérifie que $T_0 = \emptyset$, $T_n \subseteq T_{n+1}$ et $\bigcup_{n\geq 0} T_n = T$.
2. On prouve que T_n peut contenir au plus $2^{(2^n)}$ formules qui ne sont pas logiquement équivalentes deux à deux. Il y a donc un ensemble *fini* F_n tel que $F_n \subseteq T_n$ et F_n est satisfaisable ssi T_n est satisfaisable.
3. Par hypothèse, chaque sous ensemble fini de T est satisfaisable. On en déduit que :

$$\forall n\ \exists v_n\ (v_n \models T_n)\ . \tag{3.3}$$

4. Maintenant, on veut montrer qu'on peut *inverser les quantificateurs*, à savoir qu'il existe une affectation v qui satisfait tous les T_n :

$$\exists v\ \forall n\ (v \models T_n)\ . \tag{3.4}$$

Soient v_n, $n \geq 0$, les affectations de la condition 3.3. On va construire l'affectation v comme l'union d'affectations *partielles* p_i, $i \geq 0$, qui sont juste définies sur les variables $x_1, \ldots, x_i$. Si p est une affectation partielle, soit :

$$I(p) = \{n \in \mathbf{N} \mid v_n \models T_n \text{ et } p \subseteq v_n\}\ .$$

3. La notion de compacité est bien établie en topologie et dans les espaces métriques et la propriété en question peut être dérivee d'un théoreme de topologie sur les espaces compacts dû à Tykhonov.

On observe que si $I(p_i)$ est infini alors il est toujours possible de définir une affectation partielle p_{i+1} telle que $p_i \subset p_{i+1}$, et $I(p_{i+1})$ est infini aussi. En effet, il suffit de poser :

$$p_{i+1}(x_{i+1}) = \begin{cases} 1 & \text{si } \{n \mid n \in I(p_i) \text{ et } v_n(x_{i+1}) = 1\} \text{ infini} \\ 0 & \text{autrement} \end{cases}$$

Ici on utilise le fait élémentaire que si l'union de deux ensembles est infini alors au moins un des deux ensembles est infini aussi.

5. On a construit $v = \bigcup_{i \geq 0} p_i$ qui est une fonction totale sur les variables dans T et on va montrer que $v \models T$. Si $A \in T$ alors il existe j tel que $var(A) \subseteq \{x_1, \ldots, x_j\}$ et $A \in T_j$. On sait que $p_j \models A$ car il existe un n tel que $j \leq n$, $p_j \subseteq v_n$ et $v_n \models T_n$. □

Dans de nombreuses situations on a un ensemble de formules T et on aimerait savoir si la formule A est une *conséquence logique* de T.

Définition 3.5.4 (conséquence logique) *Soit T un ensemble de formules et A une formule. On dit que A est une* conséquence logique *de T et on écrit $T \models A$ si pour toute affectation v, si v satisfait chaque formule dans T alors v satisfait A.*

Remarque 3.5.5 *Si T est un ensemble* fini *de formules alors $T \models A$ ssi $\models (\bigwedge_{B \in T} B) \to A$. Donc dans ce cas la notion de conséquence logique se réduit à la notion de validité d'une formule.*

La propriété de compacité assure que si A est une conséquence logique de T alors il est conséquence logique d'un sous-ensemble *fini* de T.

Corollaire 3.5.6 (compacité et conséquence logique) *Si $T \models A$ alors il existe $T_0 \in \mathcal{P}_{fin}(T)$ tel que $T_0 \models A$.*

Idée de la preuve. Par contraposée. Supposons que pour tout $T_0 \in \mathcal{P}_{fin}(T)$ on a $T_0 \not\models A$. Alors pour tout $T_0 \in \mathcal{P}_{fin}(T)$ on a que $T_0 \cup \{\neg A\}$ est satisfaisable. Ceci implique que pour tout $T_0 \in \mathcal{P}_{fin}(T \cup \{\neg A\})$ on a que T_0 est satisfaisable. Par la proposition 3.5.3, on en déduit que $T \cup \{\neg A\}$ est satisfaisable et donc que $T \not\models A$. □

Un argument similaire permet de déduire la proposition 3.5.3 du corollaire 3.5.6. Il s'agit donc de deux formulations équivalentes de la propriété de compacité.

Exercices

Exercice 21 (morphisme) *Trouvez un autre morphisme entre les Σ-algèbres introduites dans l'exemple 3.1.4.*

Exercice 22 (définitions et Σ-algèbres initiales) *Explicitez les Σ-algèbres qui induisent au sens de la proposition 3.1.10, la fonction var de la définition 3.2.2 et la fonction $[B/x](\_)$ de la définition 3.2.3 (voir exemple 3.1.11).*

Exercice 23 (validité) *Soient A, B deux formules du calcul propositionnel. Montrez que :*

1. *A est valide ssi $\neg A$ n'est pas satisfaisable,*
2. *A est valide ssi $A \equiv \mathbf{1}$,*
3. *$A \equiv B$ ssi $A \leftrightarrow B$ est valide.*

Exercice 24 (équivalence logique et substitution) *Soient A, B, C, D des formules x une variable. Montrez que si $A \equiv B$ et $C \equiv D$ alors $[A/x]C \equiv [B/x]D$.*

Exercice 25 (raisonnement équationnel) *Utilisez un raisonnement équationnel pour déduire les (célèbres) lois suivantes :*

$$\begin{array}{ll} ((\neg y \to \neg x) \to (x \to y)) \equiv \mathbf{1} & \textit{(contraposée),} \\ (((x \wedge \neg y) \to \mathbf{0}) \to (x \to y)) \equiv \mathbf{1} & \textit{(réduction à l'absurde),} \\ (x \vee (x \wedge y)) \equiv x & \textit{(absorption).} \end{array}$$

Exercice 26 (équivalence incohérente) *Montrez que si on ajoute aux équivalences de la table 3.1 l'équivalence $(x \vee y) \equiv (x \oplus y)$ alors on peut dériver par un raisonnement équationnel $\mathbf{0} \equiv \mathbf{1}$.*

Exercice 27 (interprétation à 3 valeurs) *On considère une interprétation des formules sur un ensemble $\mathbf{3} = \{0, ?, 1\}$ qui est équipé d'un ordre total $<$ tel que $0 < ? < 1$. On interprète la formule $\mathbf{0}$ par 0, la formule $\mathbf{1}$ par 1, la conjonction $\wedge$ par la fonction binaire* minimum*, la disjonction $\vee$ par la fonction binaire* maximum *et la négation par la fonction unaire NOT_3 telle que $NOT_3(0) = 1$, $NOT_3(?) = NOT_3(1) = 0$. Certaines équivalences logiques de la table 3.1 ne sont plus vérifiées. Lesquelles ?*

Exercice 28 (équivalence et définissabilité) *Montrez que deux formules A et B sont équivalentes ssi elles définissent la même fonction sur une liste des variables dans $var(A) \cup var(B)$.*

Exercice 29 (construction CNF) *Proposez et justifiez une 'règle' pour construire une CNF à partir de la table de vérité d'une fonction booléenne.*

Exercice 30 (propriétés linéaires pour DNF et CNF) *Montrez que la satisfaisabilité d'une formule en DNF et la validité d'une formule en CNF peuvent être décidées en temps linéaire dans la taille de la formule.*

Exercice 31 (fonction de parité) *Soit* $pair(x_1, \ldots, x_n) = (\Sigma_{i=1,\ldots,n} x_i)$ *mod* 2 *la fonction qui calcule la parité d'un vecteur de bits. (1) Montrez que la formule DNF définissant cette fonction dérivée de la proposition 3.4.5 a une taille (exemple 3.1.11) exponentielle en* n*. (2) Soit* M *un monôme dans une DNF* A *qui définit la fonction pair. Montrez que toutes les variables* x_i *doivent paraître dans* M*. (3) Conclure que toute DNF* A *qui définit la fonction pair doit contenir* $2^{(n-1)}$ *monômes.*

Exercice 32 (compter les fonctions) *Est-ce possible de définir toutes les fonctions de type* $\mathbf{2}^n \to \mathbf{2}$*,* $n \geq 1$*, avec des formules dont la taille est au plus* n^3 *?*

Exercice 33 (De Morgan) *Montrez que toute formule est logiquement équivalente à une formule composée de négations et de conjonctions (ou de négations et de disjonctions).*

Exercice 34 (conditionnel) *On se focalise sur l'opérateur conditionnel (définition 3.2.4). Montrez que toute fonction* $f : \mathbf{2}^n \to \mathbf{2}$*,* $n \geq 1$*, est définissable par une formule qui utilise l'opérateur conditionnel et les formules* $\mathbf{0}$ *et* $\mathbf{1}$*.*

Exercice 35 (ou exclusif) *On considère l'*ou exclusif*, dénoté par le symbole* $\oplus$ *(définition 3.2.4). Montrez que : (1)* $\oplus$ *est associatif et commutatif, (2)* $x \oplus \mathbf{0} \equiv x$ *et* $x \oplus x \equiv \mathbf{0}$*, (3) toute fonction booléenne* $f : \mathbf{2}^n \to \mathbf{2}$*,* $n \geq 1$*, est définissable par une formule qui utilise* $\mathbf{1}$*,* $\wedge$ *et* $\oplus$*.*

Exercice 36 (nand et nor) *Les fonctions binaires NAND et NOR sont définies par :*

$$NAND(x, y) = NOT(AND(x, y)) \ , \quad NOR(x, y) = NOT(OR(x, y)) \ .$$

Montrez que toute fonction $f : \mathbf{2}^n \to \mathbf{2}$*,* $n \geq 1$*, s'exprime comme composition de la fonction NAND (ou de la fonction NOR). Montrez que les* 4 *fonctions unaires possibles n'ont pas cette propriété et que parmi les* 16 *fonctions binaires possibles il n'y en a pas d'autres qui ont cette propriété.*

Exercice 37 (le corps $\mathbf{Z}_2$) *L'ensemble* $\{0, 1\}$ *équipé avec l'addition et la multiplication modulo* 2 *est un* corps *qu'on dénote par* $\mathbf{Z}_2$*.*[4] *Notez que la multiplication modulo* 2 *coïncide avec la conjonction mais que l'addition modulo* 2 *coïncide avec le ou exclusif. Chaque polynôme en* n *variables à coefficients dans* $\mathbf{Z}_2$ *définit une fonction* $f : \mathbf{2}^n \to \mathbf{2}$*. Un polynôme* multilinéaire *est un polynôme dans lequel chaque variable peut apparaître avec degré au plus* 1*. Montrez que : (1) Chaque polynôme dans* $\mathbf{Z}_2$ *est équivalent à un polynôme multilinéaire. (2) Toute fonction* $f : \mathbf{2}^n \to \mathbf{2}$*,* $n \geq 1$ *est définissable par un polynôme multilinéaire sur* $\mathbf{Z}_2$ *en* n *variables.*

Exercice 38 (monotonie et définissabilité) *On peut ordonner l'ensemble* $\{0, 1\}$ *en supposant* $0 < 1$ *et étendre l'ordre à* $\mathbf{2}^n$ *par composantes, à savoir* $(x_1, \ldots, x_n) \leq (y_1, \ldots, y_n)$ *si* $x_i \leq y_i$ *pour* $i = 1, \ldots, n$*. Une fonction* monotone $f : \mathbf{2}^n \to \mathbf{2}$*,* $n \geq 0$*, est une fonction qui préserve l'ordre (non-strict). Montrez que les fonctions monotones sont exactement celles qui sont définissables par composition des fonctions binaires AND et OR et des constantes* 0 *et* 1*.*

4. On rappelle qu'un corps est un ensemble équipé d'une opération d'addition et d'une opération de multiplication qui satisfont des lois standards que le lecteur trouvera dans tout livre d'algèbre. Les nombres rationnels, les nombres réels et les nombres complexes avec les opérations usuelles d'addition et de multiplication sont trois exemples de corps *infinis*. Pour tout nombre premier p, l'ensemble des entiers modulo p avec addition et multiplication modulo p est un exemple de corps *fini*; et on peut même montrer qu'à isomorphisme près c'est le seul corps avec exactement p éléments !

Exercice 39 (coloriage et compacité) *On étudie une application amusante de la compacité. Soit $G = (N, A)$ un graphe non-dirigé (il peut être infini). On dit que G est k-coloriable s'il existe une fonction $c : N \to \{1, \ldots, k\}$ telle que $c(n) \neq c(n')$ si n et n' sont deux noeuds adjacents.*

1. *Pour $n \in N$ et $i \in \{1, \ldots, k\}$ on introduit une variable $x_{n,i}$. On suppose que $x_{n,i}$ vaut 1 ssi le noeud n a couleur i. Explicitez les ensembles de formules qui expriment les conditions suivantes :*
   — *Chaque noeud a au moins une couleur parmi $\{1, \ldots, k\}$.*
   — *Chaque noeud a au plus une couleur parmi $\{1, \ldots, k\}$.*
   — *Deux noeuds adjacents n'ont pas la même couleur.*
   Soit $T_{G,k}$ l'ensemble des formules qui expriment les 3 conditions.
2. *Montrez que $T_{G,k}$ est satisfaisable ssi G est k-coloriable.*
3. *On dit que G est finiment k-coloriable si tout sous-graphe fini de G est k-coloriable. Montrez que si G est finiment k-coloriable alors l'ensemble de formules $T_{G,k}$ est finiment satisfaisable.*
4. *Conclure que si un graphe est finiment k-coloriable alors il est k-coloriable.*

Chapitre 4

Logique du premier ordre

Le calcul propositionnel est une façon d'organiser le raisonnement autour d'assertions qui sont vraies ou fausses. En calcul propositionnel, on peut exprimer les propriétés d'une structure finie fixée mais on est incapable de traiter des situations où l'on veut parler d'une famille de structures (finies ou pas). Par exemple, en calcul propositionnel on peut écrire une formule qui décrit les propriétés d'un groupe avec n éléments. Si l'on veut décrire les propriétés d'un groupe avec $n+1$ éléments on doit changer la formule et on ne peut pas écrire une formule qui décrit les propriétés d'un groupe avec un nombre arbitraire d'éléments. On peut voir une formule du calcul propositionnel comme un *circuit* à savoir un programme spécialisé qui calcule sur des entrées de taille fixée. Dans ce chapitre, on va introduire une *extension* du calcul propositionnel qu'on appelle *logique du premier ordre* (ou *calcul des prédicats*) qui permet d'exprimer des programmes généraux qui calculent sur des entrées de taille arbitraire.

La consolidation du formalisme de la logique du premier ordre a pris un certain temps. Les premières idées remontent au *XIX* siècle avec Pierce, Frege,... et les résultats les plus importants ont été obtenus dans la première moitié du *XX* siècle par Russel, Hilbert, Herbrand, Lowenheim, Skolem, Gödel,... Dans cette première phase, le développement du sujet est strictement lié aux questions de formalisation et de fondation des mathématiques. Bien sûr le développement informatique est plus récent. Autour des années 1960, on commence à développer les premières techniques de preuve automatique (par exemple, la méthode de résolution [Rob65]) et ensuite on assiste à une diffusion des notions de la logique du premier ordre dans une multitude de directions. Par exemple, dans la formalisation de *logiques de la programmation* (logique de Floyd-Hoare [Hoa69],...), dans la conception de langages de *programmation logique* (PROLOG [CR93],...) et dans la conception de *langages de requête* pour les bases de données (DATALOG [GM78],...).

4.1 Syntaxe et sémantique

La logique du premier ordre a une syntaxe basée sur 3 niveaux : les *variables*, les *termes* et les *formules*.

Variables

Soit $V = \{x, y, z, \ldots\}$ un ensemble dénombrable de symboles d'arité 0. On appelle ces symboles *variables*. À noter que ces variables auront une interprétation bien différentes des

variables du calcul propositionnel. Dans la suite, on utilisera les symboles $p, q, \ldots$ pour dénoter des symboles propositionnels qui comprennent comme cas particulier les variables du calcul propositionnel.

Termes

Soit $\Sigma = \{f, g, \ldots\}$ un ensemble au plus dénombrable, disjoint de V et qui contient des symboles d'arité finie qu'on appelle *fonctions*. On appelle un symbole de fonction d'arité 0 une *constante* et on le désigne aussi avec les lettres $c, d, \ldots$ Si $V' \subseteq V$ est un sous-ensemble des variables alors on désigne par $T_\Sigma(V')$ l'algèbre initiale (définition 3.1.8) construite sur l'ensemble $\Sigma \cup V'$. On appelle *termes* les éléments de cet ensemble et on les désigne par $t, s, \ldots$ En particulier, si $V' = \emptyset$ alors les éléments de $T_\Sigma(\emptyset)$ sont les termes clos (c'est à dire ils ne contiennent pas de variables). Pour éviter des situations pathologiques, on fera l'hypothèse que Σ contient toujours au moins un symbole d'arité 0 et donc $T_\Sigma(\emptyset)$ est toujours non-vide. Si $f \in \Sigma$, $ar(f) = n$ et $t_i \in T_\Sigma(V)$ pour $i = 1, \ldots, n$ alors on utilisera la notation habituelle $f(t_1, \ldots, t_n)$ pour désigner le terme $(f, t_1, \ldots, t_n)$. L'opération de substitution d'un terme t pour une variable x dans un terme s est dénotée par $[t/x]s$ et suit la définition 3.2.3.

Formules

Soit $\Pi = \{p, q, \ldots\}$ un ensemble au plus dénombrable, disjoint de $T_\Sigma(V)$ et qui contient des symboles d'arité finie qu'on appelle *prédicats*. On remarque au passage que les *variables* du calcul propositionnel correspondent en logique du premier ordre à des prédicats d'arité 0. Si $p \in \Pi$, $ar(p) = n$ et $t_i \in T_\Sigma(V)$ pour $i = 1, \ldots, n$ alors on dit que $(p, t_1, \ldots, t_n)$ est une *formule atomique*. Soit At l'ensemble des formules atomiques où l'on considère que chaque formule a arité 0. L'ensemble des formules est l'algèbre initiale associée à l'ensemble de symboles :

$$At \cup \{\neg^1, \vee^2, \wedge^2, (\forall x)^1, (\exists x)^1 \mid x \in V\} \ .$$

Donc on construit les formules à partir des formules atomiques en utilisant les opérateurs du calcul propositionnel ainsi que des nouveaux opérateurs unaires qu'on appelle *quantificateurs universels* $(\forall x)$ et quantificateurs existentiels $(\exists x)$. Comme pour le calcul propositionnel, on distingue une syntaxe abstraite et une syntaxe concrète :

Syntaxe abstraite	Syntaxe concrète
$(p, t_1, \ldots, t_n)$	$p(t_1, \ldots, t_n)$
$(\neg, A)$	$\neg A$
$(\wedge, A, B)$	$(A \wedge B)$
$(\vee, A, B)$	$(A \vee B)$
$(\forall x, A)$	$\forall x.A$
$(\exists x, A)$	$\exists x.A$.

On écrira aussi $Qx_1, \ldots, x_n.A$ pour $Qx_1 \ldots Qx_n.A$, où $Q \in \{\forall, \exists\}$.

Variables libres, substitution et renommage

Dans une formule $Qx.A$, où $Q \in \{\forall, \exists\}$, le quantificateur Q rend la variable x liée (on dit aussi muette) dans la formule A. Les variables liées peuvent être renommées à condition d'éviter des collisions avec d'autres variables. Par exemple, la formule $\exists x.p(x, y)$ peut être renommée en $\exists z.p(z, y)$; par contre, le renommage en $\exists y.p(y, y)$ n'est pas correct. Le lecteur

a déjà rencontré cette notion de renommage, par exemple, dans le cadre du calcul intégral où l'on sait que $\int(x^2+y)\mathrm{d}x$ peut se renommer en $\int(z^2+y)\mathrm{d}z$ mais pas en $\int(y^2+y)\mathrm{d}y$. La situation se complique un peu quand on peut imbriquer les quantificateurs ou les intégrales et dans la suite on prendra le temps de définir exactement les notions de substitution et de renommage.

Définition 4.1.1 (variables libres) *Si t est un terme alors $var(t)$ est l'ensemble des variables qui paraissent dans le terme. Si A est une formule alors $vl(A)$ est l'ensemble des variables qui paraissent libres dans la formule et qui est défini comme suit où $op \in \{\wedge, \vee\}$ et $Q \in \{\forall, \exists\}$:*

$$\begin{array}{ll} vl(p(t_1,\ldots,t_n)) & = var(t_1)\cup\cdots\cup var(t_n) \\ vl(\neg A) & = vl(A) \\ vl(A\ op\ B) & = vl(A)\cup vl(B) \\ vl(Qx.A) & = vl(A)\backslash\{x\}. \end{array}$$

Exemple 4.1.2 *Soit $A = (\forall x.p(x,y)) \wedge q(x)$. Alors $vl(A) = \{x, y\}$ et on remarquera que l'occurrence libre de x est celle dans $q(x)$ alors que dans $p(x, y)$ la variable x est liée par le quantificateur.*

Définition 4.1.3 (substitution) *La substitution $[t/x]A$ d'un terme t pour une variable x dans une formule A est définie comme suit où $op \in \{\wedge, \vee\}$ et $Q \in \{\forall, \exists\}$:*

$$\begin{array}{ll} [t/x]p(t_1,\ldots,t_n) & = p([t/x]t_1,\ldots,[t/x]t_n) \\ [t/x](\neg A) & = \neg[t/x]A \\ [t/x](A\ op\ B) & = ([t/x]A\ op\ [t/x]B) \\ [t/x](Qy.A) & = \left\{\begin{array}{ll} Qy.A & \text{si } x \notin vl(Qy.A) \\ Qy.[t/x]A & \text{sinon et } y \notin var(t) \\ Qz.[t/x][z/y]A & \text{sinon et } z \text{ est la première variable} \\ & \text{telle que } z \notin var(t)\cup vl(Qy.A). \end{array}\right. \end{array}$$

Exemple 4.1.4 *Si on substitue le terme $t = f(x)$ pour y dans la formule A de l'exemple 4.1.2, on doit renommer la quantification $\forall x$ en $\forall z$, par exemple, et on obtient :*

$$[t/y]A = (\forall z.p(z, f(x))) \wedge q(x)\ .$$

Remarque 4.1.5 *La définition de la substitution est par récurrence sur la taille de la formule en observant que la taille de $[z/y]A$ est égale à la taille de A. Si on veut être pédant il faudrait dire qu'on définit d'abord la substitution d'une variable pour une variable, qu'on observe que cette substitution n'augmente pas la taille de la formule et qu'on définit ensuite la substitution d'un terme qui n'est pas une variable. On remarque aussi que dans le dernier cas on choisit la 'première variable' par rapport à une énumération fixée ; ceci permet de voir la substitution comme une fonction.*

Parfois, sans la manipulation syntaxique de formules, on a besoin de *renommer* les variables liées. D'un point de vue formel, on dérive la notion de renommage à partir de celle de substitution.

Définition 4.1.6 (renommage) *On définit une relation $\equiv$ sur les formules comme la plus petite relation d'équivalence (réflexive, symétrique et transitive) telle que pour tout $Q \in \{\forall, \exists\}$, pour toute variable x et formule A :*

$$Qx.A \equiv Qy.[y/x]A \qquad \text{si } y \notin vl(A)$$

et qui est stable par rapport aux opérateurs logiques. A savoir, si $A \equiv A'$ et $B \equiv B'$ alors pour tout $op \in \{\wedge, \vee\}$, $Q \in \{\forall, \exists\}$ et pour toute variable x : (i) $\neg A \equiv \neg A'$, (ii) $AopB \equiv A'opB'$ et (iii) $Qx.A \equiv Qx.A'$.

Exemple 4.1.7 *Si on prend la formule A de l'exemple 4.1.2, on a : $A \equiv (\forall z.p(z, y)) \wedge q(x)$.*

D'un point de vue sémantique, si $A \equiv B$ alors A et B reçoivent exactement la même interprétation (à suivre).

Définition 4.1.8 (interprétation) *Soient Σ un ensemble de symboles de fonction et Π un ensemble de symboles de prédicats. Une interprétation I, relative à Σ et Π, est définie par :*

- *un ensemble non-vide D^I qu'on appelle domaine d'interprétation,*
- *pour chaque symbole de fonction $f \in \Sigma$, d'arité n, une fonction $f^I : (D^I)^n \to D^I$,*
- *pour chaque symbole de prédicat $p \in \Pi$, d'arité n, un prédicat $p^I : (D^I)^n \to \{0, 1\}$.*

On remarquera qu'en ce qui concerne les symboles de fonction, une interprétation est rien d'autre qu'une Σ-algèbre (définition 3.1.3). Ce qu'on ajoute est l'interprétation des prédicats qui sont des fonctions dans un ensemble de valeurs de vérité $\{0, 1\}$.

Exemple 4.1.9 *On suppose un symbole de constante c, un symbole de fonction binaire f et un symbole de prédicat unaire p. Alors une interprétation I est un ensemble non vide D^I, une fonction $f^I : D^I \times D^I \to D^I$ et une fonction $p^I : D^I \to \{0, 1\}$ (ou de façon équivalente un sous-ensemble de D^I.*

On interprète les opérateurs logiques $\neg$, $\wedge$ et $\vee$ avec les fonctions NOT, AND et OR ; exactement comme dans le calcul propositionnel. Notez que cette interprétation ne dépend pas de l'interprétation I. Par contre, l'interprétation des quantificateurs dépend de l'interprétation I et plus précisément du domaine d'interprétation D^I. Pour toute interprétation I, on définit deux fonctions $\forall^I$ et $\exists^I$ telles que pour toute fonction $h : D^I \to \{0, 1\}$ on a :

$$\forall^I(h) = \begin{cases} 1 & \text{si } h^{-1}(1) = D^I \\ 0 & \text{autrement} \end{cases} \qquad \exists^I(h) = \begin{cases} 1 & \text{si } h^{-1}(1) \neq \emptyset \\ 0 & \text{autrement.} \end{cases}$$

Définition 4.1.10 (interprétation des termes) *Soit I une interprétation. Toute affectation $v : V \to D^I$ détermine une $\Sigma \cup V$-algèbre et donc une façon unique d'associer à un terme un élément de D^I qu'on peut expliciter avec les équations suivantes :*

$$\begin{aligned} [\![x]\!]Iv &= v(x) \\ [\![f(t_1, \ldots, t_n)]\!]Iv &= f^I([\![t_1]\!]Iv, \ldots, [\![t_n]\!]Iv) \ . \end{aligned}$$

Définition 4.1.11 (interprétation des formules) *Soit I une interprétation et $v : V \to D^I$ une affectation. Alors on peut associer une valeur booléenne $[\![A]\!]Iv \in \mathbf{2}$ unique à chaque*

formule A :

$$\begin{array}{lll}
[\![p(t_1,\ldots,t_n)]\!]Iv & = p^I([\![t_1]\!]Iv,\ldots,[\![t_n]\!]Iv) & \\
[\![\neg A]\!]Iv & = NOT([\![A]\!]Iv) & \\
[\![A \wedge B]\!]Iv & = AND([\![A]\!]Iv,[\![B]\!]Iv) & \\
[\![A \vee B]\!]Iv & = OR([\![A]\!]Iv,[\![B]\!]Iv) & \\
[\![\forall x.A]\!]Iv & = \forall^I(h) & \text{où : } h(d) = [\![A]\!]Iv[d/x] \\
[\![\exists x.A]\!]Iv & = \exists^I(h) & \text{idem.}
\end{array}$$

Exemple 4.1.12 *Soit $A = \forall x.\exists y.p(x,y)$ et soit I une interprétation. Alors pour toute affectation $v : V \to \{0,1\}$ on a :*

$$\begin{array}{lll}
[\![A]\!]Iv = 1 & ssi & \text{pour tout } d\ [\![\exists y.p(x,y)]\!]Iv[d/x] = 1 \\
& ssi & \text{pour tout } d, \text{ existe } d'\ [\![p(x,y)]\!]Iv[d/x,d'/y] = 1 \\
& ssi & \text{pour tout } d, \text{ existe } d'\ p^I(d,d') = 1\ .
\end{array}$$

On remarque que pour interpréter une formule quantifiée $Qx.A$, avec $Q \in \{\forall,\exists\}$ par rapport à une affectation v on interprète la sous-formule A par rapport à *toutes* les affectations de la forme $v[d/x]$ (il peut y en avoir une infinité). On remarque aussi que pour une formule close, l'interprétation ne dépend pas de l'affectation.

On dit que l'interprétation I est un modèle pour la formule A, et on écrit $I \models A$, si pour toute affectation v, $[\![A]\!]Iv = 1$. Par extension, on dit que I est un modèle pour un ensemble de formules T, et on écrit $I \models T$, si elle est un modèle pour chaque formule dans l'ensemble.

Il est facile de porter les notions de formule valide, de formule satisfaisable et de formules logiquement équivalentes (définitions 3.2.8, 3.2.9 et 3.2.10) du calcul propositionnel au premier ordre.

Définition 4.1.13 (formule valide) *On dit qu'une formule A est* valide *si pour toute interprétation I, et toute affectation $v : V \to D^I$, on a que $[\![A]\!]Iv = 1$.*

Définition 4.1.14 (formule satisfaisable) *De façon duale une formule A est* satisfaisable *s'il existe une interprétation I et une affectation $v : V \to D^I$ telle que $[\![A]\!]Iv = 1$.*

Définition 4.1.15 (formules équivalentes) *On dit que deux formules A et B sont équivalentes si pour toute interprétation I et affectation $v : V \to D^I$ on a que $[\![A]\!]Iv = [\![B]\!]Iv$.*

Remarque 4.1.16 *Le problème de savoir si une formule du premier ordre est valide semble horriblement compliqué. En calcul propositionnel, il s'agit de vérifier un nombre fini (mais exponentiel) de cas. Au premier ordre, on doit regarder tous les ensembles non-vides avec une interprétation des symboles de fonction et des symboles de prédicat. On va voir qu'en réalité il est possible de se réduire à un domaine d'interprétation qui est constitué des termes clos. Ce domaine peut être dénombrable mais au moins il est fixé et le seul paramètre variable sera l'interprétation des symboles de prédicat. Des considérations similaires s'appliquent aux questions de la satisfaisabilité et de l'équivalence.*

Traitement de l'égalité

Parmi les prédicats, on introduit souvent un prédicat d'égalité $=$ (un prédicat binaire qu'on va écrire en notation infixe). Dans ce contexte, on distingue les deux approches suivantes.

1. On axiomatise le fait que $=^I$, l'interprétation de $=$, est une relation réflexive, symétrique et transitive :

$$\forall x.x = x \ , \quad \forall x, y.x = y \to y = x \ , \quad \forall x, y, z.(x = y \wedge y = z) \to x = z \ . \tag{4.1}$$

   De plus, on demande que deux éléments égaux satisfassent les mêmes propriétés. Par exemple, si p est un prédicat unaire et f un symbole de fonction unaire on écrira :

$$\forall x, y.x = y \to (p(x) \leftrightarrow p(y)) \ , \quad \forall x, y.x = y \to f(x) = f(y) \ . \tag{4.2}$$

   A noter qu'il est possible de construire des interprétations I pour ces axiomes dans lesquelles $=^I$ n'est pas la relation identité (voir exercice 44).

2. Pour toute interprétation I, on suppose que :

$$=^I = \{(d, d) \mid d \in D^I\} \ .$$

Dans la première approche, le prédicat d'égalité est un prédicat parmi d'autres et les résultats disponibles pour la logique du premier ordre s'appliquent immédiatement. Par contre, il faut ajouter un certain nombre d'axiomes et l'interprétation de $=^I$ n'est pas forcement l'identité. Pour la deuxième approche, on peut formuler des considérations complémentaires : le prédicat $=^I$ est l'identité et on n'a pas besoin d'axiomes additionnels, par contre il faut un traitement spécial pour la logique du premier ordre avec égalité.

Exemple 4.1.17 *On veut décrire des structures algébriques qu'on appelle monoïdes. On a une opération binaire $\cdot$ qui est associative avec une identité 1 à gauche et à droite. Comme pour $=$, on va utiliser une notation infixe pour l'opération binaire. On a donc :*

$$\forall x, y, z.(x \cdot y) \cdot z = x \cdot (y \cdot z) \ , \quad \forall x.x \cdot 1 = x \ , \quad \forall x.1 \cdot x = x \ .$$

Dans la première approche, on doit ajouter les conditions (4.1) et adapter les conditions (4.2), à savoir :

$$\forall x, y, z, w.(x = z \wedge y = w) \to x \cdot y = z \cdot w \ .$$

4.2 Simplification de la structure d'une formule

On s'intéresse à des transformations qui permettent de simplifier la structure d'une formule.

Forme préfixe

Il est toujours possible de transformer une formule en une formule *équivalente* dans laquelle les quantificateurs précèdent les autres opérateurs logiques.

Définition 4.2.1 (forme prefixe) *Une formule est en forme préfixe si elle a la forme :*

$$Q_1 x_1 \cdots Q_n x_n.B$$

où $Q_i \in \{\forall, \exists\}$ et B ne contient pas de quantificateurs.

Proposition 4.2.2 (transformation en forme prefixe) *On peut transformer de façon efficace toute formule A en une formule équivalente B qui est en forme préfixe.*

Idée de la preuve. On suppose que $op \in \{\wedge, \vee\}$. On suppose aussi que $Q \in \{\forall, \exists\}$ et, par convention, si $Q = \forall$ alors $\overline{Q} = \exists$ et si $Q = \exists$ alors $\overline{Q} = \forall$. Avec ces conventions, la relation de réécriture $\rightsquigarrow$ est la plus petite relation binaire sur les formules telle que :

$$\begin{array}{lcll} \neg Qx.A & \rightsquigarrow & \overline{Q}x.\neg A & \\ (Qx.A)\ op\ B & \rightsquigarrow & Qy.([y/x]A\ op\ B) & \text{si } y \notin vl(Qx.A) \cup vl(B) \\ A\ op\ (Qx.B) & \rightsquigarrow & Qy.(A\ op\ [y/x]B) & \text{si } y \notin vl(Qx.B) \cup vl(A) \\ Qx.A & \rightsquigarrow & Qx.B & \text{si } A \rightsquigarrow B. \end{array}$$

On dit que la formule A est en *forme normale* si elle ne peut pas se réduire. On vérifie que :

1. Si A est en forme normale alors A est en forme préfixe.
2. Si A n'est pas en forme normale alors on a un algorithme efficace pour calculer un B tel que $A \rightsquigarrow B$.
3. Si $A \rightsquigarrow B$ alors $A \equiv B$ (A est logiquement équivalente à B).
4. Toute séquence de réécriture $A_1 \rightsquigarrow A_2 \rightsquigarrow \cdots$ termine et la taille des formules obtenues est linéaire dans la taille de la formule initiale A_1.

Pour obtenir une forme préfixe équivalente à une formule A, il suffit d'appliquer les règles dans un ordre arbitraire jusqu'à arriver à une forme normale. La forme normale n'est pas forcement unique mais toutes les formules obtenues par réécriture sont équivalentes à la formule de départ. □

Exemple 4.2.3 *Soit $A = \forall x.(\exists y.p(x,y) \wedge \forall y.(q(x,y) \vee r(x)))$. On a :*

$$A \ \rightsquigarrow\ \forall x.\exists y.(p(x,y) \wedge \forall y.(q(x,y) \vee r(x))) \ \rightsquigarrow\ \forall x.\exists y.\forall z.(p(x,y) \wedge (q(x,z) \vee r(x)))\ .$$

Cas sans quantificateurs

Il se trouve que si A est une formule sans quantificateurs alors on peut construire une formule B du calcul propositionnel qui est valide ssi A est valide.

Proposition 4.2.4 (transformation propositionnelle) *Soit A une formule sans quantificateurs. Si on remplace chaque formule atomique $p(t_1, \ldots, t_n)$ par une variable propositionnelle $x_{p(t_1,\ldots,t_n)}$ différente on obtient une formule propositionnelle B qui est valide (au sens propositionnel) ssi A est valide.*

Idée de la preuve. Soit A un formule sans quantificateurs et soient $p_1(\mathbf{t}_1), \ldots, p_n(\mathbf{t}_n)$ les formules atomiques syntaxiquement différentes qu'on trouve dans A. La formule B est obtenue en remplaçant chaque occurrence de la formule atomique $p_i(\mathbf{t}_i)$ par une variables propositionnelle x_i, pour $i = 1, \ldots, n$ et $x_i \neq x_j$ si $i \neq j$.

On montre d'abord que si A est valide alors B est valide. On fixe une affectation $u : \{x_1, \ldots, x_n\} \to \mathbf{2}$ arbitraire. On prend une interprétation I dont le domaine D^I est l'algèbre initiale $T_\Sigma(V)$ où $V = vl(A)$ et une affectation v telle que $v(x) = x$. Tout terme est donc interprété par lui même. L'interprétation des prédicats $p_1, \ldots, p_n$ est dirigée par l'affectation u, à savoir on pose $\mathbf{t} \in p_j^I$ ssi $\mathbf{t} = \mathbf{t}_j$, $p_j(\mathbf{t}_j)$ est une sous-formule atomique de A et $u(x_j) = 1$.

Il suit que $[\![p_j(\mathbf{t}_j)]\!]Iv = 1$ ssi $u(x_j) = 1$. Ensuite, on procède par récurrence sur la structure de A (qui n'a pas de quantificateurs) pour conclure que $[\![A]\!]Iv = [\![B]\!]u$.

Dans l'autre direction, supposons B valide et montrons que dans ce cas A est valide. Soit donc I une interprétation et $v : V \to D^I$. On utilise I et v pour construire une affectation u telle que : $u(x_j) = 1$ ssi $[\![p_j(\mathbf{t}_j)]\!]Iv = 1$. Comme dans le cas précédent, on procède par récurrence sur la structure de A. □

Exemple 4.2.5 *La formule $(p(x, c) \wedge q(c)) \vee \neg q(x)$ est valide ssi $(x_1 \wedge x_2) \vee \neg x_3$ est valide.*

Corollaire 4.2.6 (décidabilité sans quantificateurs) *L'ensemble des formules sans quantificateurs qui sont valides est décidable.*

Élimination d'alternances de quantificateurs

En logique du premier ordre, on dispose de symboles de fonctions et de symboles de prédicats ce qui est pratique pour la modélisation mais parfois un peu redondant. Par exemple, si on a une relation d'identité $=$, alors on peut remplacer un symbole fonctionnel f à n arguments par un prédicat p_f à $n + 1$ arguments qui se comporte de façon fonctionnelle, à savoir on pose :

$$\begin{array}{l}\forall x_1, \ldots, x_n.\exists y.\ p_f(x_1, \ldots, x_n, y) \wedge \\ \forall x_1, \ldots, x_n, y, y'.\ (p_f(x_1, \ldots, x_n, y) \ \wedge\ p_f(x_1, \ldots, x_n, y') \to y = y')\ .\end{array}$$

Ensuite, si on a une formule A qui contient un terme $t = f(t_1, \ldots, t_n)$ où f ne parait pas dans les termes $t_1, \ldots, t_n$, on peut la remplacer par :

$$\exists y.(A' \wedge p_f(t_1, \ldots, t_n, y))\ ,$$

où A' est la formule obtenue de A en remplaçant le terme t par y. En itérant cette transformation, on élimine le symbole de fonction f.

Dans cet exemple, pour éliminer un symbole de fonction on introduit des quantifications. On va voir qu'on peut aussi procéder dans la direction inverse. Plus précisément, on peut réduire la *satisfaisabilité* d'une formule A à la satisfaisabilité d'une formule en forme préfixe qui contient seulement des *quantificateurs universels*. On appelle cette transformation *skolemisation*.

Proposition 4.2.7 (skolemisation)

1. *Une formule $\forall x_1 \ldots \forall x_n \exists y.A$, $n \geq 0$ est satisfaisable ssi la formule $\forall x_1 \ldots \forall x_1[f(x_1, \ldots, x_n)/y]A$ est satisfaisable où f est un nouveau symbole de fonction (si $n = 0$ alors f est une constante).*
2. *Pour toute formule A, on peut construire une formule B de la forme $\forall x_1 \ldots \forall x_n.C$ telle que C est sans quantificateurs et A est satisfaisable ssi B est satisfaisable.*

Idée de la preuve. (1) ($\Rightarrow$) Soit I une interprétation qui satisfait $\forall x_1 \ldots \forall x_n \exists y.A$. On étend l'interprétation au nouveau symbole de fonction f en posant pour $d_1, \ldots, d_n \in D^I$:

$$f^I(d_1, \ldots, d_n) = d$$

où d est choisi parmi les éléments de D^I tels que $I, [d_1/x_1, \ldots, d_n/x_n, d/y] \models A$. [1]

($\Leftarrow$) Pour tout $d_1, \ldots, d_n \in D^I$, on prend $f^I(d_1, \ldots, d_n)$.

(2) En itérant l'application de (1), jusqu'à obtenir une formule sans quantificateurs existentiels. □

Il y a une version duale de ce résultat : on peut réduire la *validité* d'une formule A à la validité d'une formule en forme préfixe qui contient seulement des quantificateurs existentiels. On appelle cette transformation *herbrandisation* et la preuve du résultat est similaire à celle de la proposition 4.2.7.

Proposition 4.2.8 (herbrandisation) *1. Une formule $\exists x_1 \ldots \exists x_n \forall y.A$, $n \geq 0$ est valide ssi la formule $\exists x_1 \ldots \exists x_1 [f(x_1, \ldots, x_n)/y]A$ est valide où f est un nouveau symbole de fonction (si $n = 0$ alors f est une constante).*

2. Pour toute formule A, on peut construire une formule B de la forme $\exists x_1 \ldots \exists x_n.C$ telle que C est sans quantificateurs et A est valide ssi B est valide.

Exemple 4.2.9 *Soit B la formule en forme préfixe obtenue dans l'exemple 4.2.3. Pour skolemiser, on introduit un nouveau symbole de fonction unaire f et on obtient :*

$$\forall x. \forall z. (p(x, f(x)) \wedge (q(x, z) \vee r(x))) \ .$$

Pour herbrandiser, on introduit une constante c et un symbole de fonction f et on obtient :

$$\exists y. (p(c, y) \wedge (q(c, f(y)) \vee r(c))) \ .$$

Contrairement à la transformation en forme préfixe, les transformations pour skolemiser *ou* herbrandiser *une formule ne préservent pas l'équivalence logique. Par exemple, $\exists x.p(x, f(x))$ est l'hebrandisation de $\exists x. \forall y.p(x, y)$ mais il n'est pas difficile de trouver une interprétation I dans laquelle la première formule est vraie mais pas la seconde.*

4.3 Interprétations d'Herbrand

Dans la section précédente, on a réduit la question de la validité d'une formule à la question de la validité d'une formule de la forme $\exists x_1, \ldots, x_n.A$, où A est sans quantificateurs. Il se trouve que pour ces formules on peut se restreindre à des interprétations particulières qu'on appelle interprétations d'Herbrand.

Définition 4.3.1 (interprétation d'Herbrand) *Soient Σ un ensemble de symboles de fonction et Π un ensemble de symboles de prédicats. Une interprétation de Herbrand H relative à Σ et Π est une interprétation (définition 4.1.8) telle que : (i) $D^H = T_\Sigma$, (ii) pour tout $f \in \Sigma$, d'arité n, $f^H(d_1, \ldots, d_n) = f(d_1, \ldots, d_n)$ et (iii) pour tout $p \in \Pi$, d'arité n, p^H est un prédicat sur $(T_\Sigma)^n$.*

1. Cet argument suppose un axiome de la théorie des ensembles qu'on appelle l'*axiome du choix* et qui stipule que si on a une famille d'ensembles non vides $\{X_i \mid i \in I\}$ alors on peut construire une fonction $f : I \to \bigcup_{i \in I} X_i$ telle que $f(i) \in X_i$. C'est un résultat fondamental [Coh08] qu'on peut aussi bien concevoir des théories dans lequel l'axiome est valide et des théories dans lequel il ne l'est pas ; on appelle ce phénomène l'*indépendance de l'axiome du choix.*

Dans une interprétation d'Herbrand, le domaine et les symboles de fonction correspondent toujours à l'algèbre initiale construite sur les symboles de fonction. Le seul degré de liberté concerne les symboles de prédicats qui sont interprétés comme des relations sur l'algèbre initiale. On remarque que le domaine d'une interprétation d'Herbrand est toujours *infini* sauf si Σ contient seulement un nombre fini de symboles de constante.

Proposition 4.3.2 (validité formule existentielle) *Une formule close $\exists x_1 \ldots \exists x_n.A$, où A est sans quantificateurs, est valide ssi elle est valide dans toute interprétation de Herbrand.*

Idée de la preuve. ($\Rightarrow$) Immédiat, car une interprétation de Herbrand est une interprétation. ($\Leftarrow$) Soit A une formule sans quantificateurs. On va montrer la contraposée, à savoir : si $I \models \forall x_1, \ldots, x_n.A$ alors il y a une interprétation de Herbrand H telle que $H \models \forall x_1, \ldots, x_n.A$. Soient $t_1, t_2, \ldots$ des termes clos, soit des éléments de l'univers d'Herbrand. L'interprétation dans H des prédicats est fixée par :

$$(t_1, \ldots, t_m) \in p^H \text{ ssi } ([\![t_1]\!]I, \ldots, [\![t_m]\!]I) \in p^I \ .$$

Il suit que si B est une formule close sans quantificateurs alors : $H \models B$ ssi $I \models B$. Maintenant, on remarque :

- $I \models \forall x_1 \ldots x_n.A$ implique
- pour tout $t_1, \ldots, t_n$ termes clos, $I \models [t_1/x_1, \ldots, t_n/x_n]A$ implique
- pour tout $t_1, \ldots, t_n$ termes clos, $H \models [t_1/x_1, \ldots, t_n/x_n]A$ implique
- $H \models \forall x_1 \ldots x_n.A$. □

Exemple 4.3.3 *Soient $\Sigma = \{c^0, s^1\}$, $\Pi = \{p^1\}$ et A la formule :*

$$(\forall x.p(x) \to p(s(x))) \to (\neg p(c) \vee \exists x.p(x)) \ ,$$

qui est logiquement équivalente à la formule existentielle :

$$\exists x.(\neg p(c) \vee (p(x) \wedge \neg p(s(x))) \vee p(x)) \ .$$

Pour savoir si cette formule est valide il suffit de considérer les interprétations I où D^I est l'algèbre initiale sur $\{c^0, s^1\}$ et p^I est un sous-ensemble de $\{c, s(c), s(s(c)), \ldots\}$. Si p^I est vide la formule est satisfaite car $c \notin p^I$ et si p^I est non-vide la formule est satisfaite aussi. Donc A est valide.

En combinant les interprétations d'Herbrand avec la propriété de compacité du calcul propositionnel (proposition 3.5.3), on montre que pour toute formule du premier ordre A on peut construire une suite de formules du calcul propositionnel $A_0, A_1, \ldots$ telle que A est valide ssi une des formules de la suite est valide. Il s'agit d'une forme simple d'un résultat connu comme *théorème d'Herbrand*.

Proposition 4.3.4 ([Her30]) *Une formule close $\exists \mathbf{x}.A$, A sans quantificateurs, est valide ssi on peut trouver des vecteurs de termes clos $\mathbf{t}_1, \ldots, \mathbf{t}_n$ tels que $[\mathbf{t}_1/\mathbf{x}]A \vee \cdots \vee [\mathbf{t}_n/\mathbf{x}]A$ est valide.*

Idée de la preuve. On obtient le résultat en 4 étapes.

1. Une formule close $\exists \mathbf{x}.A$, A sans quantificateurs, est valide ssi elle est valide dans toute interprétation de Herbrand H. Ceci revient à dire que pour toute interprétation H, on peut trouver un vecteur de termes clos $\mathbf{t}$ tel que $[\![[\mathbf{t}/\mathbf{x}]A]\!]H = 1$.
2. Considérons l'ensemble T suivant constitué de formules closes et sans quantificateurs :

$$T = \{\neg[\mathbf{t}/\mathbf{x}]A \mid \mathbf{t} \text{ vecteur termes clos}\} .$$

   Cet ensemble n'est pas satisfaisable. Pour le prouver on procède par contradiction. Si T est satisfaisable alors il est satisfaisable par une interprétation de Herbrand. En suivant la proposition 4.2.4, une telle interprétation consiste à affecter des valeurs de vérité aux formules atomiques closes qui paraissent dans T de façon telle que l'interprétation (propositionnelle) de toutes les formules $[\mathbf{t}/\mathbf{x}]A$ est fausse. Mais par le point 1., on sait que pour toute affectation (ou interprétation) il doit y avoir au moins un vecteur de termes $\mathbf{t}$ tel que l'interprétation de $[\mathbf{t}/\mathbf{x}]A$ est vraie.
3. Si dans T on remplace chaque formule atomique $p(\mathbf{t})$ par une variable propositionnelle $x_{p(\mathbf{t})}$, on peut voir T comme un ensemble de formules propositionnelles et invoquer la propriété de compacité (proposition 3.5.3) pour affirmer qu'il doit exister un ensemble fini $T_0 = \{\neg[\mathbf{t_1}/\mathbf{x}]A, \ldots, \neg[\mathbf{t_n}/\mathbf{x}]A\} \subseteq T$ qui n'est pas satisfaisable.
4. Donc la formule suivante doit être valide : $[\mathbf{t_1}/\mathbf{x}]A \vee \cdots \vee [\mathbf{t_n}/\mathbf{x}]A$. □

Exemple 4.3.5 *La formule $\exists x.((p(a) \vee p(b)) \rightarrow p(x))$ est valide. Pour le vérifier, soit I une interprétation de Herbrand. On a $D^I = \{a, b\}$ et on a 4 interprétations possibles pour p^I. Il est important de noter que dans ce cas on ne peut pas remplacer x par un seul terme. En effet, il y a deux termes possibles, à savoir a et b, et dans les deux cas on obtient une formule qui n'est pas valide. La proposition 4.3.4 est quand même correcte car la disjonction suivante est valide : $(p(a) \vee p(b) \rightarrow p(a)) \vee (p(a) \vee p(b) \rightarrow p(b))$.*

Corollaire 4.3.6 (semi-décidabilité) *L'ensemble des formules valides est* semi-décidable.

Idée de la preuve. Soit A une formule. Au besoin, on quantifie universellement les variables libres. On calcule une formule logiquement équivalente en forme préfixe B. On dérive une formule de la forme $\exists \mathbf{x}.C$, C sans quantificateurs, qui est valide ssi B est valide. On applique la proposition 4.3.4 à la formule $\exists \mathbf{x}.C$. □

4.4 Calculabilité et validité

Par le corollaire 4.3.6, on sait que l'ensemble des formules valides est semi-décidable. Dans cette section, on va montrer qu'on ne peut pas faire mieux : l'ensemble des formules valides n'est pas décidable [Chu36].

Proposition 4.4.1 (indécidabilité formules valides) *On peut réduire le problème de l'arrêt d'un programme au problème de la validité d'une formule de la logique du premier ordre.*

Idée de la preuve. Pour simplifier l'argument, on suppose qu'on a une machine à deux compteurs M (section 2.3). Soit $\Sigma = \{c^0, s^1\}$ et soit T_Σ l'algèbre initiale (les nombres naturels en représentation unaire). Une configuration de la machine est un triplet (q, n, m) où q est une instruction et $n, m \in T_\Sigma$. Pour chaque instruction q on suppose un symbole de prédicat

binaire p_q. On décrit chaque instruction de la machine par une formule. Par exemple, si dans l'état q la machine incrémente le deuxième compteur et va dans l'état q' on écrit :

$$\forall x, y.(p_q(x, y) \to p_{q'}(x, s(y))) \ .$$

Soit A_M la conjonction de toutes les formules associées aux instructions de la machine M. On prouve par récurrence sur la longueur du calcul que :

$$(q, n, m) \vdash_M^* (q', n', m') \text{ implique } \models p_q(n, m) \wedge A_M \to p_{q'}(n', m') \ . \tag{4.3}$$

D'autre part, soit (q_0, n_0, m_0) la configuration initiale de la machine. On veut prouver que :

$$\models p_{q_0}(n_0, m_0) \wedge A_M \to p_q(n, m) \text{ implique } (p_0, n_0, m_0) \vdash_M^* (q, n, m) \ . \tag{4.4}$$

On définit une interprétation de Herbrand H où l'on pose :

$$(n, m) \in p_q^H \text{ ssi } (q_0, n_0, m_0) \vdash_M^* (q, n, m) \ .$$

Maintenant, on vérifie que :

$$H \models p_{q_0}(n_0, m_0) \text{ et } H \models A_M \ ,$$

et donc par hypothèse $H \models p_q(n, m)$, soit $(p_0, n_0, m_0) \vdash_M^* (q, n, m)$. Sans perte de généralité, on peut supposer que les machines s'arrêtent en mettant les deux compteurs à zéro et en allant dans un état final q_F. Dans ce cas, en combinant (4.3) et (4.4), on a que : $(q_0, n_0, m_0) \vdash_M^* (q_F, c, c)$ ssi la formule $p_{q_0}(n_0, m_0) \wedge A_M \to p_{q_a}(c, c)$ est valide. On a donc réduit le problème de l'arrêt au problème de la validité d'une formule. □

On vient de voir que l'ensemble des formules valides est semi-décidable mais pas décidable. Il suit que l'ensemble des formules satisfaisables est le complémentaire d'un ensemble semi-décidable et il n'est pas décidable non plus.

Dans la suite on va évoquer deux résultats un peu plus avancés qui suggèrent qu'il n'est pas vraiment possible de contourner cette difficulté. Le premier résultat affirme que la situation n'est guère meilleure si on se limite aux interprétations avec un domaine d'interprétation *fini* [Tra50].

Proposition 4.4.2 (interprétations finies) *L'ensemble des formules valides dans les interprétations finies n'est pas semi-décidable.*

Idée de la preuve. L'idée de départ est que pour toute machine à deux compteurs (ou MdT) M, on peut construire une formule A_M telle que :

$$M \text{ termine ssi } \exists\, I \text{ finie } \ I \models A_M \ .$$

Il suit que M diverge ssi pour toute interprétation finie I, on a que $I \models \neg A_M$. Donc on peut réduire le problème de la divergence de toute machine M au problème de la validité dans toutes les interprétations finies. On trouvera les détails de la construction de la formule A_M, par exemple, dans [EF95]. □

Échaudés par les propositions 4.4.1 et 4.4.2, on peut se dire que ce qui nous intéresse vraiment est l'ensemble des formules vraies dans des interprétations "naturelles". Considérons donc les formules construites sur une signature qui comprend deux fonctions binaires $+$ et $*$ et un prédicat binaire $=$. On appelle les formules dans ce langage les *formules de l'arithmétique*. On interprète les formules de l'arithmétique dans l'ensemble $\mathbf{N}$ des nombres naturels (de façon équivalente on pourrait prendre l'ensemble des entiers) avec les opérations d'addition et de multiplication et le prédicat d'égalité. Le lecteur conviendra qu'il s'agit bien d'une interprétation "naturelle" ; peut être même la mère de toutes les interprétations. Ce qui suit est une première étape vers un résultat célébrissime de la logique qu'on appelle *théorème d'incomplétude de Gödel* [G̈31] .

Proposition 4.4.3 (indécidabilité de l'arithmétique) *L'ensemble des formules de l'arithmétique vraies dans l'interprétation* $\mathbf{N}$ *n'est pas décidable.*

Idée de la preuve. Pour toute machine à deux compteurs M et configuration initiale (q_0, n, m) on construit une formule A qui est vraie dans l'interprétation $\mathbf{N}$ ssi la machine accepte.

(1) Une configuration d'une machine à deux compteurs (q, n, m) est un triplet de nombres dont le premier varie dans un ensemble fini, par exemple $0 \leq q \leq k-1$, où k est le nombre d'états de la machine. On peut écrire une formule (de l'arithmétique) $\mathit{config}(x)$ qui affirme que x est la codification d'un triplet (q, n, m) avec $0 \leq q \leq k-1$. Pour ce faire, on peut définir $x \leq y$ comme une abréviation pour $\exists z.y = x + z$ et on peut utiliser les fonctions pour coder les couples de nombres étudiées dans l'exercice 16(1) qui utilisent seulement les opérations d'addition et de multiplication. On peut aussi écrire des formules $\mathit{init}(x)$ et $\mathit{term}(x)$ qui affirment que x est la codification d'un triplet (q, n, m) qui correspond à la configuration initiale et terminale, respectivement.

(2) Les règles de transition de la machine induisent une relation binaire sur les configurations. On peut écrire une formule $\mathit{step}(x, y)$ qui affirme que x et y sont les codifications de deux configurations et que d'après les règles de la machine, on peut aller de la première à la deuxième.

(3) Le crux est d'arriver à écrire une formule $\mathit{calcul}(x, y)$ qui affirme que x est la codification d'une suite $x_0, \ldots, x_y$ de longueur $y+1$ telle que $\mathit{step}(x_0, x_1), \ldots, \mathit{step}(x_{y-1}, x_y)$. En première approximation, la formule A recherchée prend la forme :

$$\exists y.(\mathit{calcul}(x, y) \wedge \mathit{init}(x_0) \wedge \mathit{term}(x_y)) \ , \tag{4.5}$$

à savoir il y a une suite de configurations de longueur $y+1$ dont la première est la configuration initiale, la dernière est la configuration terminale et qui constitue une séquence légitime d'après les règles de calcul de la machine. A noter que l'approche qui consisterait à dire que $x = \langle \cdots \langle x_0, x_1 \rangle, \ldots, x_y \rangle$ en utilisant les fonctions de codification de l'exercice 16(1) ne marche pas car dans ce cas la taille de la formule dépendrait de y ; ce qu'on veut est une seule formule dont la taille ne dépend pas de y. L'astuce est de faire appel à des notions élémentaires d'arithmétique modulaire. Considérons la formule suivante introduite par Gödel :

$$(x \equiv a) \mod (1 + (1 + i) * b) \tag{4.6}$$

Il est facile de vérifier qu'il s'agit d'une formule de l'arithmétique car on peut définir les notions de quotient et de reste à partir de l'addition et de la multiplication. En suivant Gödel, on

abrège la formule comme $\beta(a, b, i, x)$. Si on fixe a et b, on peut voir la formule comme une fonction qui associe à chaque i la valeur $a \mod (1 + (1 + i) * b)$. Donc si on s'intéresse à des séquences de nombres de longueur $y + 1$ on pourrait fixer a et b et ensuite, en faisant varier i entre 0 et y, obtenir $y + 1$ valeurs $a \mod (1 + b), \ldots, a \mod (1 + (y + 1) * b$. Ce qu'il faut montrer est que toute suite de nombres $n_0, \ldots, n_y$ peut être obtenue pour un certain choix de a et b.

(4) Soit donc $n_0, \ldots, n_y$ une suite de nombres. On veut montrer qu'on peut choisir a et b pour que $\beta(a, b, i, x)$ soit équivalente à $x = n_i$ pour $i = 0, \ldots, y$. On va utiliser un résultat élémentaire d'arithmétique modulaire connu comme *théorème des restes chinois*.

> Soient $m_0, \ldots, m_y$ des nombres entiers positifs premiers entre eux et soient $n_0, \ldots, n_y$ des nombres entiers. Alors le *système de congruences* :
>
> $$(a \equiv n_i) \mod m_i, i \in \{0, \ldots, y\} \tag{4.7}$$
>
> a une *solution* qui est *unique modulo* $M = m_0 \cdots m_y$.

Pour appliquer ce résultat à la fonction β, on prend $m_i = (1 + (1 + i) * b)$ et on choisit b de façon telle que $n_i < m_i$ et les m_i sont premiers entre eux pour $i = 0, \ldots, y$. Un choix qui fait l'affaire est de prendre le factoriel du plus grand élément parmi les n_i et y, à savoir :

$$b = (max(y, n_0, \ldots, n_y))! \tag{4.8}$$

Il est facile de vérifier qu'il n'existe pas un nombre premier p qui divise m_i et m_j pour $i \neq j$. Maintenant, on prend a comme la solution du système de congruences (4.7) et on obtient que pour ce choix de a et b la formule $\beta(a, b, i, x)$ est équivalente à dire que $x = n_i$.

(5) On peut donc conclure qu'en faisant varier a, b, y sur les nombres naturels on peut obtenir toutes les suites de nombres $n_0, \ldots, n_y$ et qu'en faisant varier i entre 0 et y on peut accéder à tous les éléments de la suite. Plus précisément, on peut maintenant raffiner la formule A évoquée dans (4.5) comme suit :

$$\begin{aligned} \exists a, b, y. \quad & \exists x.(\beta(a, b, 0, x) \wedge init(x)) \quad \wedge \\ & \exists x.(\beta(a, b, y, x) \wedge term(x)) \quad \wedge \\ & \forall i.(0 \leq i < y \rightarrow \exists x, x'.(\beta(a, b, i, x) \wedge \beta(a, b, i + 1, x') \wedge step(x, x'))) \ . \end{aligned}$$

□

Remarque 4.4.4 *La représentation du calcul d'une machine à deux compteurs qu'on vient de décrire implique des formules logiques avec un nombre borné de quantificateurs ; en effet il s'agit essentiellement d'une formule de la forme* $\exists \mathbf{x}.A$*, où* A *n'a pas de quantificateurs. Si on permet des alternances de quantificateurs, on peut décrire des problèmes toujours plus difficiles qu'on peut classifier dans une hiérarchie appelée* hiérarchie arithmétique*. Les problèmes semi-décidables et les problèmes dont le complémentaire est semi-décidable ne représentent que le premier étage de ce bâtiment qui en contient une infinité !*

Exercices

Exercice 40 (dualité $\forall, \exists$) *Vérifiez que pour toute interprétation I et fonction $h : D^I \to \{0,1\}$ on a : $NOT(\exists^I(h)) = (\forall^I(NOT \circ h))$.*

Exercice 41 (autour de la validité) *Soit A une formule telle que $vl(A) \subseteq \{x_1, \ldots, x_n\}$. Montrez que :*

1. *A est valide ssi sa clôture universelle $\forall x_1. \cdots \forall x_n.A$ est valide,*
2. *A est satisfaisable ssi sa clôture existentielle $\exists x_1. \cdots \exists x_n.A$ est satisfaisable,*
3. *A est valide ssi $\neg A$ n'est pas satisfaisable,*
4. *A est équivalente à une formule B ssi la formule $A \leftrightarrow B$ est valide.*

Exercice 42 (équivalences logiques) *Parmi les équivalences logiques suivantes, certaines sont fausses ; lesquelles ?*

$$\begin{array}{llll} (1) & \forall x.A \wedge \forall x.B \equiv \forall x.(A \wedge B) & (2) & \forall x.A \vee \forall x.B \equiv \forall x.(A \vee B) \\ (3) & \exists x.A \wedge \exists x.B \equiv \exists x.(A \wedge B) & (4) & \exists x.A \vee \exists x.B \equiv \exists x.(A \vee B) \\ (5) & \forall x.\exists y.p(x,y) \equiv \exists y.\forall x.p(x,y) & (6) & (\exists x.p(x)) \to p(y) \equiv \forall x.(p(x) \to p(y)) \ . \end{array}$$

Exercice 43 (interprétation infinie) *Considérez la formule A suivante :*

$$\forall x.p(x, f(x)) \wedge \forall x.\neg p(x,x) \wedge \forall x, y, z.(p(x,y) \wedge p(y,z) \to p(x,z)) \ .$$

Montrez que tous les modèles de A ont un domaine d'interprétation infini.

Exercice 44 (égalité(s))

1. *Pour tout $n \geq 1$, trouvez une formule A_n qui est satisfaisable et dont tous les modèles ont au moins n éléments.*
2. *Soit A une formule satisfaisable par une interprétation avec n éléments. Montrez que dans ce cas A est satisfaisable aussi par une interprétation infinie.*
3. *Considérez la formule E suivante qui décrit les propriétés de réflexivité, symétrie et transitivité d'un prédicat d'égalité e :*

   $$\forall x.e(x,x) \wedge \forall x, y.(e(x,y) \to e(y,x)) \wedge \forall x, y, z.(e(x,y) \wedge e(y,z) \to e(x,z))$$

   Soit A la formule qui dit que tous les éléments "sont égaux" : $\forall x, y.e(x,y)$. Montrez que la formule $E \wedge A$ admet un modèle infini (SIC).
4. *On suppose maintenant disposer d'un prédicat binaire $=$ qui est interprété par la relation identité. Montrez que dans ce cas, pour tout $n \geq 1$ on peut écrire une formule A_n qui est satisfaisable et dont les modèles ont au plus n éléments.*

Exercice 45 (théories) *La notion de* conséquence logique *introduite pour le calcul propositionnel (définition 3.5.4) se généralise au calcul des prédicats. Soit T un ensemble (fini ou dénombrable) de formules closes et soit A une formule close. On écrit $T \models A$ si toute interprétation qui satisfait les formules dans T satisfait aussi A. Si T est un ensemble de formules on dénote ses conséquences logiques par :*

$$Cons(T) = \{A \mid T \models A\} \ .$$

Clairement, $Cons(T) \supseteq T$ et on dit que T est une théorie *si $Cons(T) = T$. Une théorie est :*

— cohérente *s'il existe une formule A telle que $A \notin T$,*

— complète *si pour tout A ou bien $A \in T$ ou bien $\neg A \in T$,*
— *(finiment)* axiomatisable *s'il existe $T' \subseteq T$ tel que T' est décidable (fini) et $Cons(T') = T$.*

On rappelle que si T est fini alors $T \models A$ est semi-décidable (corollaire 4.3.6). Par ailleurs, la propriété de compacité est valide aussi dans le calcul des prédicats. En particulier on a (cf. corollaire 3.5.6) :

$$T \models A \text{ ssi } \exists T_0 \in \mathcal{P}_{fin}(T)\ \ T_0 \models A.$$

En utilisant ces résultats, montrez que :

1. *Si T est semi-décidable alors $Cons(T)$ est semi-décidable.*
2. *Si T est axiomatisable et complète alors $Cons(T)$ est décidable.*
3. *Pour toute interprétation, l'ensemble des formules vraies dans l'interprétation est une théorie cohérente et complète.* [2]

Exercice 46 (forme prefixe, skolemisée, herbrandisée) *Transformez la formule suivante en forme préfixe :*

$$\forall x.(\exists y.p(x, g(y, f(x))) \vee \neg q(z)) \vee \neg \forall x.r(x, y)\ ,$$

Ensuite, calculez la forme skolemisée et la forme herbrandisée.

Exercice 47 (Herbrand)

1. *Trouvez une formule $\forall \mathbf{x}.A$, A sans quantificateurs, qui n'est pas valide mais qui est vraie dans toute interprétation d'Herbrand.*
2. *On suppose les signatures $\Sigma = \{c^0, f^1, g^1\}$ et $\Pi = \{p^1, q^2\}$. Considérez la formule $\exists x, y, z.A$ sur ces signatures où :*

   $$A = \neg p(x) \vee (\neg q(x, f(x)) \wedge p(x)) \vee q(g(y), z)\ .$$

   Vérifiez que la formule est valide et ensuite trouvez des termes clos t_1, t_2, t_3 tels que $[t_1/x, t_2/y, t_3/z]A$ est valide.
3. *Maintenant on suppose les signatures $\Sigma = \{a^0, b^0, f^1, g^2\}$ et $\Pi = \{p^3\}$. Considérez la formule $\exists x, y, z.A$ sur ces signatures où :*

   $$A = \neg p(x, a, g(x, b)) \vee p(f(y), z, g(f(a), b))$$

   A nouveau, vérifiez que la formule est valide et ensuite trouvez des termes clos t_1, t_2, t_3 tels que $[t_1/x, t_2/y, t_3/z]A$ est valide.

Exercice 48 (interprétation finie) *Donnez un exemple d'une formule qui n'est pas valide mais qui est vraie dant toute interprétation avec un domaine fini.*

Exercice 49 (représentation du calcul) *Considèrez la machine M à 2 compteurs suivante qui termine dans l'état $q_F = q_6$ si les deux compteurs contiennent la même valeur et boucle autrement. L'instruction $dec_i \to q$ décremente le compteur i et saute à l'instruction q et l'instruction $zero_i \to q, q'$ saute à l'instruction q si le compteur i est zéro et à l'instruction q' autrement.*

2. Une formalisation de l'arithmétique connue comme arithmétique de Peano [Pea89] est un exemple de théorie axiomatisable qui est ni finiment axiomatisable ni complète (ce qui est attendu par la proposition 4.4.3).

$$\begin{array}{ll} q_0: & zero_1 \to q_4, q_1 \\ q_1: & dec_1 \to q_2 \\ q_2: & zero_2 \to q_5, q_3 \\ q_3: & dec_2 \to q_0 \\ q_4: & zero_2 \to q_6, q_5 \\ q_5 & dec_1 \to q_5 \end{array}$$

1. *En suivant la preuve de la proposition 4.4.1, explicitez la formule A_M qui décrit le calcul de la machine.*
2. *On se place maintenant dans le cadre de la proposition 4.4.3. Pour la machine M ci-dessus écrire :*
   — *les formules $config(x)$ $init(x)$ et $term(x)$ évoquées dans la première étape de la preuve de la proposition.*
   — *la formule $step(x, y)$ dont il est question dans la deuxième étape de la preuve de la proposition.*

Exercice 50 (monadique) *A la lumière de la proposition 4.4.1, on peut se demander si on peut trouver des fragments décidables de la logique du premier ordre avec quantificateurs. Si on permet une alternance arbitraire de quantificateurs alors une façon de procéder et d'exclure les fonctions et de se limiter à des prédicats monadiques (ou unaires). On dit qu'une formule est* monadique *si elle contient seulement des prédicats monadiques et des constantes (symboles de fonction d'arité* 0*). On fixe n prédicats monadiques $p_1, \ldots, p_n$ et on considère des formules qui peuvent contenir seulement ces prédicats. Soit I une interprétation. On définit le* spectre *$S(d)$ d'un élément $d \in D^I$ par :*

$$S(d) = \{i \mid d \in p_i^I\} \ .$$

La notion de spectre induit une relation d'équivalence *$\sim_I$ sur D^I par :*

$$d \sim_I d' \text{ si } S(d) = S(d') \ .$$

1. *Pour toute interprétation I, définissez une interprétation quotient finie $I/\sim$ et montrez que pour toute formule A, on a $I \models A$ ssi $I/\sim \models A$.*
2. *Montrez que si A est satisfaisable alors elle est satisfaisable par une interprétation dont le domaine contient au plus 2^n éléments.*
3. *Conclure que l'ensemble des formules monadiques valides est décidable.*

Exercice 51 (Presburger) *On considère les formules sur une signature qui contient juste une fonction binaire $+$ et un prédicat binaire $=$ (on exclut donc la multiplication). Comme dans la proposition 4.4.3, on fixe comme interprétation $\mathbf{N}$ l'ensemble des nombre naturels avec addition et égalité. Par rapport à cette interprétation, considérons une formule A et une liste de variables $x_1, \ldots, x_m$ sans répétition qui contient les variables libres dans A. On associe à la formule A un sous-ensemble de $\mathbf{N}^m$:*

$$[\![A]\!]\mathbf{N} = \{(n_1, \ldots, n_m) \in \mathbf{N}^m \mid [\![A]\!]\mathbf{N}, [n_1/x_1, \ldots, n_m/x_m] = 1\} \ .$$

On dit aussi que la formule A définit un sous-ensemble de $\mathbf{N}^m$. On va voir que modulo un codage naturel des tuples de nombres comme des mots, cet ensemble est reconnaissable par un automate fini. Un corollaire de cette propriété est que l'ensemble des formules closes vraies

dans l'interprétation $\mathbf{N}$ *est décidable : il suffit de construire l'automate qui reconnaît le langage défini pas la formule et de vérifier que ce langage n'est pas vide.* [3]

1. *Construisez des formules qui définissent les ensembles suivants par rapport à une liste de variables adéquate :*

   (1) $\{n \mid n = 0\}$, (2) $\{(n, m) \mid n \leq m\}$, (3) $\{n \mid n = 1\}$,
   (4) $\{n \mid (k \mid n)\}$, (5) $\{(m, n, p) \mid p = max(m, n)\}$.

2. *Montrez que pour toute formule A sans quantificateurs et liste de variables adéquate on peut définir une formule B qui peut avoir des quantificateurs, qui définit le même ensemble et dont les seules sous-formules atomiques ont la forme $x + y = z$ ou $x = y$.*
3. *On explicite maintenant le codage des vecteurs de nombres comme mots. Un automate qui reconnaît des ensembles dans $\mathbf{N}^m$ a comme alphabet $\Sigma = \mathbf{2}^m$. En particulier, si $m = 0$ alors $\mathbf{N}^0 = \{()\}$ est l'ensemble qui contient la tuple vide et l'alphabet associé $\mathbf{2}^0$ contient un seul symbol, disons $*$. Un mot $w \in (2^m)^*$ peut être vu comme une matrice dans laquelle chaque ligne correspond à la représentation binaire d'un nombre. Par exemple, le mot :*

   $$\begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

   correspond au vecteur $(3, 7) \in \mathbf{N}^2$. Si $m \geq 1$ on peut prendre la convention que le mot vide ϵ représente la tuple $(0, \ldots, 0)$ qui contient m fois 0 et si $m = 0$ alors on peut prendre la convention que tout mot sur l'alphabet $\{*\}$ (y compris le mot vide) représente la tuple $()$.
   Construisez les automates qui reconnaissent les ensembles définis par les formules $x = y$ et $x = y + z$ par rapport aux listes x, y et x, y, z respectivement (on peut s'appuyer sur l'exercice 4.4).
4. *Supposons qu'on dispose de l'automate M pour une formule par rapport à la liste de variables x, y et qu'on souhaite construire un automate M' pour la même formule par rapport à la liste de variables x, z, y. On appelle cette opération* cylindrification. *Montrez qu'on peut obtenir M en gardant les états de l'automate M et en associant à chaque transition de M deux transitions dans M'.*
5. *Soient M_1 et M_2 les automates pour les formules A_1 et A_2 par rapport à la même liste de variables. Expliquez comment construire les automates pour $\neg A_1$, $A_1 \wedge A_2$ et $A_1 \vee A_2$.*
6. *Soit M un automate pour la formule A par rapport à la liste $x_1, \ldots, x_n$. Montrez qu'on peut construire un automate M' pour la formule $\exists x_i.A$ par rapport à la liste $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$. On appelle cette opération* projection. *Suggestion : comme pour la cylindrification l'automate M' garde les états de l'automate M.*
7. *Montrez qu'on peut traiter la quantification universelle en utilisant l'équivalence logique $\forall x.A \equiv \neg\exists x.\neg A$. Par exemple, appliquez votre méthode à la formule (fausse !) $\exists x.\forall y.x = y$.*

3. Ce fragment décidable de l'arithmétique est connu comme arithmétique de Presburger ; à noter que Presburger [Pre29] a utilisé une technique de preuve basée sur l'élimination des quantificateurs qui est différente de celle discutée dans l'exercice. Il se trouve qu'en tout cas les algorithmes obtenus ne sont pas très efficaces ; il a été prouvé [FR74] que tout algorithme qui décide si une formule de cette arithmétique est vraie prend un temps au moins doublement exponentiel dans la taille de la formule.

8. *Conclure que pour toute formule A et liste de variables, on peut construire un automate M qui reconnaît le langage défini par la formule et qu'on peut donc décider si la formule A est vraie dans l'interprétation* $\mathbf{N}$.

Chapitre 5

Classes de complexité : P et NP

On suppose le lecteur familier avec la notion de complexité asymptotique. Il s'agit d'une mesure qui n'est pas très sensible aux détails de la mise en oeuvre et qui permet d'avoir une première estimation de l'efficacité d'un algorithme. Une grande partie des problèmes qui sont considérés dans un cours standard d'algorithmique peuvent être résolus par un algorithme qui prend un temps polynomial dans la taille de l'entrée. Par exemple : le tri, la solution d'un système d'équations linéaires, l'inversion d'une matrice, la recherche d'un élément dans un arbre, la recherche de la plus longue sous-séquence commune de deux mots, la connectivité d'un graphe, la recherche des plus courts chemins dans un graphe,... On peut regrouper tous ces problèmes dans une classe de complexité qu'on dénote par P.

En pratique, on rencontre souvent des problèmes pour lesquels on ne connaît pas d'algorithme polynomial *déterministe* mais qui sont décidables par un algorithme polynomial *non-déterministe*. On appelle NP la classe qui contient tous ces problèmes ; le N est là pour nous rappeler qu'on a un algorithme polynomial *non-déterministe*. Une façon équivalente de décrire la classe NP est de dire qu'il s'agit de la classes de problèmes pour lesquels on dispose d'un algorithme polynomial *déterministe* pour vérifier la validité d'une solution du problème. En d'autres termes, la classe NP contient des problèmes pour lesquels la recherche d'une solution peut être difficile mais dont on sait au moins vérifier la validité d'une solution de façon efficace. On peut noter qu'on est confronté à ce type de situation chaque fois qu'on cherche à prouver un théorème : trouver une preuve semble difficile mais la vérifier est en principe une tâche beaucoup plus simple.

Par définition, la classe P est contenue dans la classe NP et le problème de savoir si cette inclusion est stricte est un problème naturel de la théorie de la complexité et il est probablement le problème ouvert le plus médiatisé de l'informatique fondamentale. [1] A défaut de résoudre ce problème, on présente une *méthode* qui repose sur les notions de réduction polynomiale et de NP-complétude. Le but de la méthode est de montrer que si le problème qui nous intéresse admet un algorithme polynomial déterministe alors *tous* les problèmes de la classe NP admettent aussi un algorithme polynomial déterministe. En d'autres termes, si on arrive à appliquer la méthode, on peut conclure que le problème qui nous intéresse est bien parmi les plus difficiles de la classe NP.

1. Le problème P *vs.* NP est cité parmi les "7 problèmes mathématiques du troisième millénaire" par la *Clay Foundation* à coté de *l'hypothèse de Riemann*, *la conjecture de Poincaré*, la *résolution des équations de Navier-Stokes*,... La preuve de la conjecture de Poincaré a été annoncée récemment, il ne reste donc que 6 problèmes...

5.1 Temps de calcul polynomial

Un pas de calcul d'une MdT est une opération élémentaire qui demande un effort de calcul borné : il s'agit de consulter un tableau fini, d'écrire un symbole et de déplacer d'une position la tête de lecture. Il semble donc raisonnable de mesurer le *temps de calcul* d'une MdT simplement comme le nombre de pas de calcul nécessaires pour arriver à un état final.

Définition 5.1.1 (temps de calcul) *Soit M une MdT qui termine sur toute entrée. La fonction de coût (en temps) de M est une fonction $c : \mathbf{N} \to \mathbf{N}$ où $c(n)$ est le nombre maximal de pas de calcul nécessaires à la machine pour terminer sur une entrée de taille au plus n (la taille d'un mot est sa longueur).*

Souvent on s'intéresse seulement à l'ordre de grandeur de la complexité. On rappelle la notation O et la notion de complexité asymptotique que le lecteur a déjà rencontré dans les cours d'algorithmique.

Définition 5.1.2 (notation O) *Soient $f, g : \mathbf{N} \to \mathbf{N}$ deux fonctions sur les nombres naturels. On dit que f est $O(g)$ s'ils existent $n_0, k \in \mathbf{N}$ tels que pour tout $n \geq n_0$, $f(n) \leq k \cdot g(n)$.*

En d'autres termes, f est $O(g)$ si presque partout f est dominée par g à une constante multiplicative près.

Définition 5.1.3 (complexité asymptotique) *Soit $g : \mathbf{N} \to \mathbf{N}$ une fonction sur les nombres naturels et M une MdT. On dit que M est $O(g)$ si la fonction de coût c de M est $O(g)$.*

Par exemple, dire qu'une machine M est $O(n)$ veut dire qu'ils existent $n_0, k \in \mathbf{N}$ tels que pour toute entrée w de taille $n \geq n_0$ le temps de calcul de M sur l'entrée w est au plus kn. On s'intéresse aux problèmes décidables en temps polynomial (déterministe ou non-déterministe).

Définition 5.1.4 (classes P et NP) *P (NP) est la classe des langages qui sont décidables par une MdT déterministe (non-déterministe) en temps $O(n^k)$, pour un certain k (pour chaque langage on doit fixer un k qui ne dépend pas de la taille de l'entrée).*

Il suit de la définition que tout problème dans P est aussi dans NP. Les classes P et NP sont suffisamment robustes pour ne pas être affectées par des modifications raisonnables du modèle de calcul. Par exemple, ces classes ne dépendent pas du fait que les MdT disposent de un ou de plusieurs rubans (voir proposition 2.3.1). On peut même enrichir le modèle de calcul en supposant que la machine dispose d'une mémoire illimitée en accès direct (RAM pour *random access memory*). Dans une telle machine l'accès à une cellule de mémoire est effectué en $O(1)$. On peut démontrer qu'une MdT déterministe peut simuler une machine avec RAM avec une dégradation polynomiale des performances, c'est-à-dire qu'il y a un (petit) nombre k tel que si la machine avec RAM a complexité $O(t(n))$ la MdT qui la simule a complexité $O(t(n)^k)$.

Exemple 5.1.5 (problèmes dans P) *Voici deux problèmes qui admettent un algorithme polynomial (déterministe) qui n'est pas du tout trivial et dont la découverte a été une percée scientifique très significative.*

Programmation linéaire *Le problème de savoir si un système d'équations linéaires à coefficients rationnels a une solution est bien sûr dans* P*. Par exemple, on peut utiliser la méthode d'élimination de Gauss. Si on remplace les équations par des inégalités le problème devient considérablement plus compliqué. Le problème est équivalent à un problème célèbre connu comme problème de* programmation linéaire *à savoir le problème de minimiser (ou maximiser) une fonction linéaire à n arguments sujette à des inégalités linéaires.*

Depuis les années 1950, une méthode de choix pour ce problème consiste à utiliser l'algorithme du simplexe [Dan48] qui est souvent efficace en pratique mais qui peut prendre un temps exponentiel. C'est seulement depuis [Kha79] qu'on sait que le problème est dans P *et depuis [Kar84] qu'on a développé une nouvelle famille d'algorithmes (dits méthodes de points intérieurs) qui sont dans* P *et qui sont compétitifs avec l'algorithme du simplexe en pratique.*

Primalité *On suppose une représentation des nombres, par exemple, en base* 10*. Depuis [AKS04], on sait que l'ensemble des nombres premiers est dans* P*. Avant ce travail, on disposait seulement d'algorithmes polynomiaux* probabilistes *(voir, par exemple, [Mil76, Rab80]). Ces algorithmes ont une très petite probabilité de se tromper ; plus précisément il peuvent déclarer premier un nombre qui ne l'est pas. En pratique, la probabilité est négligeable et les algorithmes probabilistes sont plus efficaces que l'algorithme polynomial déterministe.*

Exemple 5.1.6 (problèmes ouverts) *Voici trois problèmes pour lesquels on ne connaît pas d'algorithme polynomial déterministe mais on n'a pas perdu espoir d'en trouver...*

Identité de polynômes *On ne connaît pas d'algorithme polynomial (déterministe) pour savoir si deux expressions polynomiales à plusieurs variables définissent le même polynôme. Par exemple, l'algorithme devrait reconnaître le fait que ces deux polynômes sont identiques :* $(x-y)(x+y) = x^2-y^2$*. Notez que les polynômes peuvent être présentés comme produit de polynômes et qu'en général la stratégie qui consiste à normaliser les polynômes comme somme de monômes et à les comparer prend un temps exponentiel. Par exemple, considérez le polynôme* $\Pi_{i=1,\ldots,n}(x_i - x_{i-1})$ *qui a une taille linéaire en n mais dont l'expansion s'exprime comme la somme de* 2^n *monômes de degré n. On peut contourner ces difficultés en utilisant un algorithme probabiliste. L'algorithme utilise le fait qu'on peut évaluer de façon efficace un polynôme dans un point sans passer par son expansion comme somme de monômes. Comme pour le problème de la primalité, cet algorithme à une petite probabilité de se tromper : plus précisément il peut déclarer deux polynômes identiques alors qu'ils sont différents.*

Factorisation *On ne connaît pas d'algorithme polynomial pour calculer les facteurs premiers d'un nombre naturel* $n \geq 2$*. De façon équivalente, on ne connaît pas un algorithme polynomial qui prend en entrée un couple* (n, b) *et détermine si* $2 \leq b < n$ *et n a un diviseur compris entre* 2 *et b. On abrège ce problème comme FACT.*

Isomorphisme de graphes *On ne connaît pas d'algorithme polynomial pour savoir si deux graphes finis* $G_1 = (N_1, A_1)$ *et* $G_2 = (N_2, A_2)$ *sont isomorphes, c'est-à-dire s'il existe une bijection des noeuds* $\theta : N_1 \to N_2$ *compatible avec les arêtes, à savoir :* $(i, j) \in A_1$ *ssi* $(\theta(i), \theta(j)) \in A_2$*.*

Exemple 5.1.7 (problèmes NP-complets) *Voici 3 problèmes qui sont dans la classe* NP *et qui ont une propriété dite de* NP*-complétude qui sera discutée dans la section 5.2 : si on trouve un algorithme polynomial déterministe pour un de ces problèmes alors on a un algorithme polynomial pour tous les problèmes dans* NP*.*

SAT *Le problème SAT consiste à déterminer si une formule du calcul propositionnel est satisfaisable. Ce problème est dans* NP *car il suffit de deviner une affectation et de vérifier.*

HAMILTON *Soit $G = (V, E)$ un graphe non-dirigé. Le problème du* parcours hamiltonien *consiste à déterminer s'il existe un parcours du graphe qui contient chaque sommet du graphe une et une seule fois. Un algorithme dans* NP *qui répond à la question devine une permutation des sommets et vérifie si elle correspond à un parcours dans le graphe.*

TSP *Soit V un ensemble de villes et d une fonction qui associe à chaque paire de villes (v, v') la distance $d(v, v') \geq 0$ pour aller de v à v'. Le problème du* voyageur de commerce *(aussi connu comme TSP pour* Travelling Salesman Problem*) consiste à déterminer s'il existe un parcours qui traverse chaque ville exactement une fois dont la longueur est inférieure à b. En d'autres termes, dans TSP on considère un graphe non-dirigé, complet (chaque couple de noeuds est connecté par une arête) et avec une fonction de coût sur les arêtes et on cherche à déterminer si le graphe contient un parcours hamiltonien dont le coût est inférieur à b. Un algorithme dans* NP *qui répond à la question devine une permutation des villes et vérifie si la somme des distances est inférieure à b. On note que le problème TSP est aussi formulé comme un problème d'*optimisation *où l'on cherche à minimiser la longueur d'un parcours fermé. Il est souvent facile de dériver un algorithme pour l'optimisation de l'algorithme de décision. Par exemple, dans le cas en question on peut procéder par recherche dichotomique.*

NP : la classe des problèmes faciles à vérifier

On va discuter une présentation alternative de la classe NP évoquée dans l'introduction du chapitre qui ne fait pas appel à la notion de non-déterminisme. Considérons d'abord le problème *SAT* qui consiste à déterminer si une formule du calcul propositionnel est satisfaisable. Pour répondre à cette question, on peut explorer les 2^n affectations possibles d'une formule avec n variables. On peut remarquer qu'il est facile de vérifier si une formule A est satisfaisable, en effet il suffit d'exhiber une affectation v et de vérifier que $[\![A]\!]v = 1$ car :

$$SAT = \{A \mid \exists\, v \;\; [\![A]\!]v = 1\} \; .$$

On peut voir l'affectation v comme un *certificat* (ou témoin, ou preuve) de la propriété que A est satisfaisable ; il peut être difficile de trouver un certificat mais il est certainement facile de vérifier si on a un certificat. En généralisant cette remarque, l'idée est de caractériser la classe NP comme la classe des problèmes qui admettent des certificats vérifiables en temps polynomial.

Définition 5.1.8 (vérificateur polynomial) *Un vérificateur pour un langage L est un algorithme de décision V tel que :*

$$L = \{w \mid \exists\, c \;\; V \text{ accepte } \langle w, c\rangle\} \; .$$

Un vérificateur V est polynomial s'il exécute en temps polynomial en $|w|$.

Remarque 5.1.9 *Le fait que le vérificateur V est polynomial en $|w|$ implique qu'il peut examiner seulement une portion polynomiale du certificat c ; en pratique on peut toujours supposer que le certificat c a une taille polynomiale en $|w|$.*

Proposition 5.1.10 (caractérisation NP) *La classe* NP *contient exactement les langages qui ont un vérificateur polynomial.*

Idée de la preuve. Soit L un langage qui dispose d'un vérificateur polynomial V et soit $p(n)$ le polynôme en question. La MdT non-déterministe qui accepte L se comporte de la façon suivante sur une entrée w de taille n :

— elle sélectionne de façon non déterministe un mot c de longueur au plus $p(n)$,
— elle simule V sur $\langle w, c\rangle$,
— elle accepte ssi une des branches accepte.

D'autre part, supposons que L soit accepté par une MdT non-déterministe N de complexité polynomiale. Un vérificateur polynomial V pour L se comporte de la façon suivante sur une entrée $\langle w, c\rangle$:

— il simule l'exécution de N sur une entrée w en traitant c comme une codification des choix non-déterministes effectués par N à chaque étape,
— il accepte si la branche déterminée par c accepte. □

5.2 Réductions polynomiales et NP-complétude

Faute de pouvoir démontrer que les problèmes dans l'exemple 5.1.7 sont ou ne sont pas dans P, on va essayer de les comparer. A cette fin, on reprend la notion de *réduction* entre problèmes (définition 2.4.4) en ajoutant la contrainte que la réduction est calculable en temps polynomiale (déterministe).

Définition 5.2.1 (réduction polynomiale) *Soient L, L' deux langages sur un alphabet Σ. On dit que L se réduit à L' en* temps polynomial *et on écrit $L \leq_P L'$ s'il existe une fonction récursive $f : \Sigma^* \to \Sigma^*$ calculable en temps polynomial telle que : $w \in L$ ssi $w \in L'$.*

Exemple 5.2.2 *Il y a une réduction polynomiale du problème du parcours hamiltonien au problème du voyageur de commerce. L'ensemble des noeuds correspond à l'ensemble des villes. La distance d est définie par :*

$$d(v, v') = \begin{cases} 1 & \textit{si } \{v, v'\} \textit{ arête} \\ 2 & \textit{autrement.} \end{cases}$$

La constante b est égale au nombre des villes moins 1. Maintenant, on remarque :

— *s'il existe un parcours de longueur b alors ce parcours ne peut contenir que des chemins entre villes de longueur 1 ; donc ce parcours correspond à un parcours hamiltonien,*
— *inversement, s'il y a un parcours hamiltonien alors la réponse au problème du voyageur de commerce est positive.*

Définition 5.2.3 (NP-complétude) *Un problème L (langage) est* NP*-complet s'il est dans* NP *et si tout problème L' dans* NP *admet une réduction polynomiale à L.*

Dans un certain sens les problèmes NP-complets sont les plus durs. Si on trouve un algorithme polynomial pour un problème NP-complet alors on a un algorithme polynomial pour tous les problèmes de la classe NP. Un fait remarquable est que plusieurs problèmes naturels sont NP-complets. Certains problèmes comme l'*isomorphisme de graphes* et la *factorisation* (exemple 5.1.6) résistent à une classification. En l'état de nos connaissances, il est possible que ces problèmes soient ni NP-complets ni dans P (plus sur ce point dans la prochaine section 5.3).

Pour montrer l'existence d'un problème NP-complet, on va reprendre un refrain de ce cours : on peut utiliser la logique pour représenter le calcul ! Dans la proposition 4.4.1, on a représenté le calcul d'une machine M sur une entrée w par des formules de la *logique du premier ordre*. Dans la suite, une nouveauté essentielle est qu'on va représenter un calcul *polynomial* d'une machine non-déterministe N sur une entrée w. Un tel calcul peut être vu comme une matrice de taille polynomiale en $|w|$ et le contenu des éléments de la matrice peut être décrit exactement par une formule du calcul *propositionnel* dont la taille est aussi polynomiale en $|w|$.

Proposition 5.2.4 ([Coo71, Lev73]) *Le problème SAT est* NP*-complet.*

Idée de la preuve. Soit L un langage décidé par une MdT M non déterministe polynomiale en temps $p(n)$. Donc $w \in L$ ssi à partir de la configuration initiale $q_0 w$ la machine M peut arriver à l'état q_a. On décrit une réduction polynomiale qui associe à chaque mot w une formule en CNF A_w qui est satisfaisable si et seulement si $w \in L$. L'idée est que la formule A_w va décrire tous les calculs possibles (M est non-déterministe !) de la machine M sur l'entrée w. La remarque fondamentale est qu'un calcul d'une machine de Turing en temps $p(n)$ sur un mot w de taille n peut être représenté par un tableau de taille $p(n) \times p(n)$ dont la case de coordonnées (i, j) contient la valeur du ruban au temps i et à la position j. Si le calcul termine avant $p(n)$ on peut toujours recopier le ruban jusqu'au temps $p(n)$.

On peut associer à chaque case (i, j) et à chaque symbole a une variable propositionnelle $x_{i,j,a}$ avec l'idée que $x_{i,j,a} = 1$ si et seulement si la case (i, j) contient le symbole a.

Ensuite on peut construire des formules (de taille polynomiale en n) qui assurent que :

— Exactement un symbole est dans chaque case.
— Les cases $(1, j)$ correspondent à la configuration initiale.
— Chaque case $(i+1, j)$ est obtenue des cases $(i, j-1), (i, j), (i, j+1)$ selon les règles de la Machine.
— La configuration finale accepte.

On donne un peu plus de détails dans l'exemple suivant. □

Exemple 5.2.5 *On construit une CNF qui correspond au calcul de la MdT M dans l'exemple 2.1.4 sur l'entrée aab.*[2] *Le calcul de la MdT pourrait être :*

	1	2	3	4	5
1	q_0	a	a	b	⊔
2	a	q_0	a	b	⊔
3	a	a	q_0	b	⊔
4	a	a	b	q_1	⊔
5	a	a	b	⊔	q_a

2. Il s'agit d'un cas très spécial car la MdT en question se comporte comme un automate fini déterministe. Cependant les idées se généralisent.

Pour représenter le calcul on introduit les variables $x_{i,j,u}$ où $i,j \in \{1,\ldots,5\}$ et $u \in \{a,b,q_0,q_1,q_a\}$. La configuration initiale est spécifiée par :

$$A_{init} = x_{1,1,q_0} \wedge x_{1,2,a} \wedge x_{1,3,a} \wedge x_{1,4,b} \wedge x_{1,5,\sqcup} \ .$$

On doit imposer la contrainte que à chaque instant exactement un symbole est présent à chaque position. Par exemple, pour l'instant i à la position j on écrira :

$$A_{i,j} = (x_{i,j,a} \vee \cdots \vee x_{i,j,q_a}) \wedge (\neg x_{i,j,a} \vee \neg x_{i,j,b}) \wedge \cdots \wedge (\neg x_{i,j,q_1} \vee \neg x_{i,j,q_a}) \ .$$

L'objectif est d'arriver à une configuration qui contient l'état q_a. Cela revient à demander :

$$A_{accept} = \bigvee_{1 \leq i,j \leq 5} x_{i,j,q_a} \ .$$

Enfin on doit décrire les 'règles de calcul' de la machine M. Par exemple, on pourrait exprimer $\delta(q_0,a) = (q_0,a,R)$ par la conjonction de formules de la forme :

$$(x_{i-1,j-1,q_0} \wedge x_{i-1,j,a}) \rightarrow (x_{i,j-1,a} \wedge x_{i,j,q_0}) \ .$$

Il est possible de procéder d'une façon plus systématique. Une propriété intéressante des MdT est qu'à chaque instant le calcul est localisé dans une région de taille bornée. Si $w_1 q w_2 \vdash_M w_1' q' w_2'$ la différence entre les deux configurations est localisée dans une région de taille 3 qui comprend l'état et les deux symboles contiguës. L'idée est alors de regarder toutes les fenêtres de largeur 3 et de hauteur 2 dans le tableau qui représente le calcul (il y en a un nombre polynomial) et de s'assurer que le contenu de chaque fenêtre est conforme aux règles de la machine.

La formule en question peut être exprimée en CNF. Par exemple, on pourrait avoir une formule de la forme :

$$(x_1 \wedge x_2) \rightarrow ((y_1 \wedge y_2) \vee (w_1 \wedge w_2)) \ ,$$

pour dire que si deux cases contiennent les symboles a_1, a_2 (variables x_1, x_2) alors deux autres cases contiennent ou bien les symboles b_1, b_2 (variables y_1, y_2) ou bien les symboles c_1, c_2 (variables w_1, w_2). C'est un simple exercice de réécrire une telle formule en CNF. Notez que la disjonction à droite de l'implication permet d'exprimer le non-déterminisme du calcul.

Remarque 5.2.6 (espace polynomial) *Une autre façon de mesurer la complexité du calcul d'une MdT est de compter l'espace, c'est-à-dire le nombre de cellules du ruban qu'elle utilise. La classe* PSPACE *(*NPSPACE*) est la classe des problèmes qui peuvent être résolus par une MdT déterministe (non-déterministe) en utilisant un espace polynomial dans la taille de l'entrée. Pour utiliser de l'espace il faut du temps ; cette remarque élémentaire implique que* (N)P $\subseteq$ (N)PSPACE*. Par ailleurs, il n'est pas très difficile de montrer que* PSPACE *est égal à* NPSPACE*. On en déduit immédiatement que :*

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} = \mathsf{NPSPACE} \ ,$$

mais on ne sait pas si une de ces inclusions est stricte !

5.3 Classification de quelques problèmes remarquables

Dans cette section, on considère la classification de quelques problèmes remarquables.

CLIQUE

Soit G un graphe non-dirigé. Un k-clique est un ensemble de k noeuds de G qui ont la propriété que chaque couple de noeuds est connectée par une arête. Le langage CLIQUE est composé de couples $\langle G, k\rangle$ tels que : (i) G est le codage d'un graphe, (ii) k est un nombre naturel et (iii) G contient comme sous-graphe un k-clique.

Proposition 5.3.1 (CLIQUE) *Le problème CLIQUE est* NP*-complet.*

Idée de la preuve. On montre d'abord que le problème est dans NP. Un certificat pour le problème est un ensemble de k noeuds et la vérification consiste à contrôler que tous les noeuds dans l'ensemble sont connectés dans G.

Le problème *3-SAT* est l'ensemble des CNF satisfaisables dont les clauses contiennent exactement 3 littéraux. L'exercice 53 vous demande de prouver que le problème *SAT* a une réduction polynomiale au problème *3-SAT*. Pour montrer la NP-complétude de *CLIQUE*, il suffit donc de montrer que le problème *3-SAT* admet une réduction polynomiale à *CLIQUE*.

Soit donc A une formule qui contient k clauses composées de 3 littéraux. On construit un graphe G_A qui contient k groupes de 3 noeuds, un pour chaque clause. Chaque noeud dans un groupe est étiqueté exactement par un littéral de la clause correspondante. Par exemple, si la clause est $(x \vee \neg y \vee z)$ alors on aura un groupe de 3 noeuds étiquetés avec x, $\neg y$ et z. Les arêtes du graphe G_A sont spécifiées de la façon suivante : deux noeuds sont toujours connectés sauf s'ils sont dans le même groupe ou s'il sont étiquetés par un littéral et sa négation. Il reste à vérifier que la formule A est satisfaisable ssi le graphe G_A a un k-clique. □

Programmation entière

Le problème *ILP* est la version du problème de programmation linéaire évoqué dans l'exemple 5.1.5 dans laquelle les variables varient sur les *entiers* plutôt que sur les *rationnels*. Il est aisé de trouver une réduction polynomiale de *3-SAT* à *ILP*, mais il est beaucoup plus délicat de montrer que *ILP* est dans NP.

Proposition 5.3.2 (ILP) *Le problème de savoir si un système d'inégalités linéaires a une solution entière est* NP*-complet.*

Idée de la preuve. On présente une réduction de *3-SAT* à *ILP*. Un problème *3-SAT* est un système de contraintes de la forme :

$$\ell_1 \vee \ell_2 \vee \ell_3$$

où ℓ_i sont des littéraux Un problème *ILP* est un système de contraintes de la forme :

$$a_1 x_1 + \cdots + a_n x_n \geq b$$

où a_i, x_i, b varient sur les nombres entiers. Avec les contraintes $x_i \geq 0$ et $-x_i \geq -1$ on exprime la condition que $x_i \in \{0, 1\}$. La disjonction $\vee$ est remplacée par le $+$ et la négation de la variable x devient $(1 - x)$. Ainsi, la CNF :

$$(\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$$

devient le système d'inégalités suivantes :

$$\begin{cases} x_i & \geq & 0 & i \in \{1,2,3\} \\ -x_i & \geq & -1 & i \in \{1,2,3\} \\ (1-x_1)+x_2+(1-x_3) & \geq & 1 & \\ x_1+(1-x_2)+x_3 & \geq & 1\ . & \end{cases}$$

Et on vérifie que la CNF est satisfaisable ssi le système d'inégalités associées a une solution entière (la condition que les variables sont entières est essentielle, si les variables varient sur les nombres rationnels la réduction ne marche pas).

La partie la plus compliquée de la preuve consiste à montrer que le problème est dans NP. Pour ce faire il faut montrer que si une solution existe alors on peut en trouver une dont la taille est polynomiale dans la taille du système. Il s'agit d'un argument assez technique et on réfère le lecteur à [PS82][chapitre 13] pour les détails. Le point de départ est que les solutions rationnelles si elles existent, s'expriment par le biais de la règle de Cramer comme des déterminants de matrices liées au système et que la taille des nombres nécessaires à exprimer ces déterminants est *polynomiale* dans la taille des nombres utilisés dans le système à résoudre. □

Factorisation

Il n'est pas toujours possible d'appliquer la méthode esquissée pour le problème *CLIQUE*. Par exemple, le problème de la factorisation déjà évoqué dans l'exemple 5.1.6 est remarquable par son intérêt pratique et aussi pour avoir une position particulière dans la classification des problèmes en classes de complexité. Soit co-NP l'ensemble des langages dont le complémentaire est dans la classe NP.

Proposition 5.3.3 (factorisation) *Le problème FACT est dans* NP *et dans* co-NP.

Idée de la preuve. Pour montrer que le problème *FACT* est dans NP, il suffit d'utiliser comme certificat un nombre d tel que $2 \leq d \leq b$ et $d \mid n$. Pour montrer que le problème *FACT* est dans co-NP, il faut trouver un certificat du fait que pour tout d tel que $2 \leq d \leq b < n$, d ne divise pas n. On remarque que la factorisation $p_1 \cdots p_k$ de n en nombres premiers est un tel certificat. En effet, on vérifie que : (i) $p_i > b$ et p_i est premier pour $i = 1, \ldots, k$ et (ii) $p_1 \cdots p_k = n$. Ceci suffit car $2 \leq d \leq b$ et d divise n ssi d est les produit d'un sous-ensemble des nombres premiers qui constituent la factorisation de n. □

Remarque 5.3.4 *On ne sait pas si le problème de la factorisation est* NP*-complet ; si c'était le cas on aurait que* NP*=co-*NP *! On sait [Sho94] que le problème est dans* BQP *à savoir la classe des problèmes solubles en temps polynomial par un algorithme* probabiliste *qui tourne sur un ordinateur* quantique*. A noter qu'on ne sait même pas si* BQP *est contenu dans* NP *et qu'on rencontre toujours des difficultés considérables à construire des ordinateurs quantiques qui passent à l'échelle.*

Exercices

Exercice 52 (notation O) *Montrez que :*

1. *Si f est $O(g)$ et g est $O(h)$ alors f est $O(h)$.*
2. *$6n^3 + 2n^2 + 20n + 45$ est $O(n^3)$.*
3. *Il y a une MdT M qui décide le langage $L = \{w\sharp w \mid w \in \{0,1\}^*\}$ en $O(n^2)$.*

Exercice 53 (*3-SAT*) *Une formule est en 3-CNF si elle est en CNF et chaque clause (disjonction de littéraux) comporte exactement 3 littéraux. On rappelle que le problème 3-SAT consiste à déterminer si une formule en 3-CNF est satisfaisable. Montrez que :*

1. *3-SAT est dans* NP.[3]
2. *Soit A une formule en CNF qui ne contient pas la clause vide. On peut construire en temps linéaire une formule B en CNF avec les propriétés suivantes :*
   - *A est satisfaisable ssi B est satisfaisable,*
   - *toutes les clauses dans B ont 3 littéraux et*
   - *la taille de B est linéaire dans la taille de A.*

Exercice 54 (problèmes dans NP) *Montrer que les problèmes suivants sont dans* NP *(voir exemple 5.1.6) :*

- *l'isomorphisme de graphes,*
- *l'identité de deux polynômes en une variable sur le corps des nombres rationnels.*

Exercice 55 (réduction polynomiale) *Montrez que la notion de réduction polynomiale est transitive : $L_1 \leq_P L_2$ et $L_2 \leq_P L_3$ implique $L_1 \leq_P L_3$.*

Exercice 56 (*CLIQUE* $\leq_P$ *SAT*) *La proposition 5.3.1 affirme que le problème CLIQUE est* NP*-complet. Comme le problème SAT est* NP*-complet aussi (proposition 5.2.4) on sait à priori qu'il y a une réduction polynomiale de CLIQUE à SAT. Le but de cet exercice est d'en expliciter une qui est plus compréhensible que celle donnée implicitement par la preuve de la proposition 5.2.4.*

Si n est le nombre de noeuds du graphe et k la taille de la clique alors on introduit kn variables propositionnelles $x_{i,j}$, $i = 1, \ldots, k$, $j = 1, \ldots, n$ avec l'idée que :

$$x_{i,j} = 1 \text{ ssi } j \text{ est le } i\text{-ème noeud de la clique.}$$

Ici on suppose que les noeuds de la clique on été numérotés de 1 à k. Formalisez par une formule du calcul propositionnel les propriétés suivantes :

1. *Pour $i = 1, \ldots, k$, il y a un i-ème noeud de la clique.*
2. *Pour $i = 1, \ldots, k$ et $1 \leq j_1 < j_2 \leq n$, j_1 et j_2 ne peuvent pas être le i-ème noeud de la clique.*
3. *Pour $1 \leq i_1 < i_2 \leq k$ et $1 \leq j_1 < j_2 \leq n$ si $\{j_1, j_2\} \notin A$ alors j_1 et j_2 ne peuvent pas être dans la clique en même temps.*
4. *Combien de clauses contient la formule qui formalise la propriété que le graphe contient une k-clique ?*

Exercice 57 (mini-Sudoku) *Par souci de simplicité, on considère un mini-Sudoku de dimension 4. Par exemple :*

3. Si on permet au plus 2 littéraux par clause alors le problème de la satisfaction est dans P.

1			2
4			
	2	4	

Il s'agit maintenant d'exprimer les règles du jeu *et les* conditions initiales *avec une formule CNF A ; on doit avoir : A satisfaisable ssi le jeu a une solution. Pour $i, j, k \in \{1, 2, 3, 4\}$, on introduit les variables $x_{i,j,k}$ (soit 64 variables) avec l'idée que :*

$$x_{i,j,k} = 1 \text{ ssi la cellule } (i, j) \text{ de la grille contient le chiffre } k.$$

Exprimez les contraintes suivantes comme CNF :

1. *Chaque cellule contient au moins un chiffre :*
2. *Chaque chiffre apparaît au plus une fois dans chaque ligne, chaque colonne et chaque région.*
3. *Les conditions initiales du jeu.*

Exercice 58 (coloriage et emploi du temps) *Considérez les deux problèmes suivants.*

Problème du coloriage. *Étant donné un graphe $G = (N, A)$ fini, non-dirigé et un nombre naturel $k \geq 2$ déterminer s'il existe une fonction $c : N \to \{1, \ldots, k\}$ telle que si i, j sont deux noeuds adjacents alors $c(i) \neq c(j)$ (on peut voir les valeurs $\{1, \ldots, k\}$ comme des couleurs qu'on affecte aux noeuds, d'où le nom du problème qu'on a déjà rencontré dans sa version infinie dans l'exercice 39).*

Problème de l'emploi du temps. *Soient :*

- *$E = \{1, \ldots, n\}$, $n \geq 2$, un ensemble d'étudiants,*
- *$C = \{1, \ldots, m\}$, $m \geq 2$, un ensemble de cours,*
- *$P = \{1, \ldots, p\}$, $p \geq 2$, un ensemble de plages horaires et*
- *R une relations binaire R telle que $(i, j) \in R$ si et seulement si l'étudiant i suit le cours j.*

Déterminer s'il existe une fonction emploi du temps $edt : C \to P$ telle que si un étudiant suit deux cours différents $j \neq j'$ alors $edt(j) \neq edt(j')$.

Montrez, en explicitant la réduction si possible, que :

1. *le problème de l'emploi du temps se réduit en temps polynomial au problème du coloriage,*
2. *un jeu du Sudoku (exercice 57) peut se formuler comme un problème de coloriage,*
3. *le problème du coloriage se réduit en temps polynomial au problème SAT.*

Exercice 59 (HAMILTON $\leq_P$ SAT) *Montrez, sans utiliser la proposition 5.2.4, qu'il y a une réduction polynomiale du problème du parcours hamiltonien (exemple 5.1.7) au problème SAT.*

Bibliographie

[Ack28] Wilhelm Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematischen Annalen*, 99 :118–133, 1928.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2) :781–793, 2004.

[Ard61] Dean N. Arden. Delayed-logic and finite-state machines. In *2nd Symposium on Switching Circuit Theory and Logical Design*, pages 133–151. IEEE Computer Society, 1961.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8) :677–691, 1986.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2) :345–363, 1936.

[Coh08] Paul Cohen. *Set theory and the continuum hypothesis*. Dover, 2008. Première publication en 1966.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proc. 3rd ACM Symposium on Theory of Computing*, pages 151–158. ACM, 1971.

[CR93] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *History of Programming Languages Conference (HOPL-II), Preprints, Cambridge, Massachusetts, USA, April 20-23, 1993*, pages 37–52. ACM, 1993.

[Dan48] George Dantzig. Programming in a linear structure. Technical report, United States Air Force, Washington DC., 1948.

[EF95] Heinz-Dieter Ebbinghaus and Jörg Flum. *Finite model theory*. Springer, 1995.

[FR74] Michael Fischer and Michael Rabin. Super-exponential complexity of Presburger arithmetic. In *Proceedings of the SIAM-AMS Symposium in Applied Mathematics, 7*, pages 27–41, 1974.

[G$\ddot{3}$1] Kurt Gödel. Über formal unentscheidbar Sätze der Principia Mathematica und verwandter Systeme, I. *Monatshefte für Mathematik und Physik*, 38 :173–198, 1931.

[GM78] Hervé Gallaire and Jack Minker, editors. *Logic and Data Bases, Symposium on Logic and Data Bases, Centre d'études et de recherches de Toulouse, France, 1977*, Advances in Data Base Theory. Plemum Press, 1978.

[Her30] Jacques Herbrand. Recherches sur la théorie de la démonstration. *Travaux de la société des Sciences et des Lettres de Varsovie, Class III, Sciences Mathématiques et Physiques*, 33, 1930.

[Hil00] David Hilbert. Mathematische Probleme. *Göttinger Nachrichten*, pages 253–297, 1900.

[Hoa69] Charles A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, 1969.

[Kar84] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4) :373–396, 1984.

[Kha79] Leonid Khachiyan. A polynomial algorithm in linear programming. *Akademiia Nauk SSSR. Doklady*, 244 :1093–1096, 1979.

[Kle56] Stephen Kleene. Representation of events in nerve nets and finite automata. *Annals of mathematical studies, Princeton University Press*, 34 :3–41, 1956.

[Lev73] Leonid Levin. Universal'nye perebornye zadachi (Universal search problems). *Problemy Peredachi Informatsii (Problems of Information Transmission)*, 9(3) :115–116, 1973.

[Mat93] Yuri Matiyasevich. *Hilbert's tenth problem*. MIT-Press, 1993. Disponible aussi en Russe et en Français.

[Mil76] Gary L. Miller. Riemann's hypothesis and tests for primality. *J. Comput. Syst. Sci.*, 13(3) :300–317, 1976.

[Ner58] Anil Nerode. Linear automaton transformations. *Proceedings of the American mathematical society*, 9(4) :541–544, 1958.

[Pea89] Giuseppe Peano. *Arithmetices Principia nova methode exposita.* Université de Turin, 1889. Traduction en anglais dans Jean van Heijenoort, A Source Book in Mathematical Logic, 1967.

[Per90] Dominique Perrin. Finite automata. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B : Formal Models and Semantics*, pages 1–57. Elsevier and MIT Press, 1990.

[Pre29] Mojżesz Presburger. Über die Vollständigkeit eines gewissen Systems der arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*, pages 92–101, 1929.

[PS82] Christos Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization : algorithms and complexity.* Prentice-Hall, 1982.

[Rab80] Michael O. Rabin. Probabilistic algorithms in finite fields. *SIAM J. Comput.*, 9(2) :273–280, 1980.

[Ric53] Henry Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American mathematical society*, 74(2) :358–366, 1953.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1) :23–41, 1965.

[RS59] Michael O. Rabin and Dana S. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2) :114–125, 1959.

[Sho94] Peter Shor. Algorithms for quantum computation : discrete logarithms and factoring. In *Proceedings 35th IEEE Symp. on Foundations of Computer Science*, pages 124–134, 1994.

[Tar48] Alfred Tarski. A decision method for elementary algebra and geometry. Technical report, Rand Corporation, R-109, 1948.

[Tra50] Boris Trakhtenbrot. The impossibility of an algorithm for the decidability problem on finite classes. *Proceedings of the USSR academy of science*, 70(4) :569–572, 1950. En russe.

Index

Annexe A

Travaux pratiques

A.1 Analyse lexicale

Un fichier texte est une suite de caractères et la première tâche nécessaire à son analyse consiste à en extraire et classifier les *unités lexicales* qu'il contient. Par exemple, s'il s'agit d'un texte dans une langue naturelle ou dans un langage de programmation on veut extraire la liste des mots qu'il contient avec éventuellement une information sur leur *typologie*, à savoir pour une langue naturelle on veut connaître la catégorie grammaticale du mot (nom, adjectif, verbe,...) et pour un langage de programmation on veut savoir s'il s'agit d'un identificateur, d'un mot réservé, d'un opérateur arithmétique ou logique,...

Dans ce travail, on vise à utiliser un logiciel appelé flex (ou lex qui est son prédécesseur) qui permet de déclarer :

— la liste des unités lexicales dans une notation proche des expressions rationnelles et
— la liste des 'actions' associées à chaque unité lexicale.

Ensuite, flex construit automatiquement un programme C qui va reconnaître les unités lexicales et exécuter les actions associées. En particulier, un tel programme peut prendre un fichier en entrée et produire en sortie le flot des unités lexicales contenues dans le fichier ou un message d'erreur. On trouvera plus d'informations, par exemple, dans `https://en.wikipedia.org/wiki/Flex_(lexical_analyser_generator)`.

A noter que des logiciels équivalents ont été développés pour des langages comme Java, OCaml, python,... Tous ces logiciels se basent sur la notion de langage rationnel et sur les constructions associées étudiées dans le chapitre 1, notamment la transformation d'une expression rationnelle en AFN, la déterminisation de l'AFN, la minimisation de l'AFD,...

Voici une petite spécification flex qu'on écrit dans un fichier qu'on va appeler, par exemple, tp.lex.

```
  /* (1) C Declarations */
%{
#include <stdio.h>
  void action(char *token, char *value){
    printf("%s\t%s\n",token,value);
    return; }
%}
%option noyywrap
/* (2) Regular expressions declarations */
LETTRE   [a-zA-Z]
CHIFFRE  [0-9]
MOT      {LETTRE}+
NOMBRE   {CHIFFRE}+
BLANC    " "|"\t"|"\n"
/* (3) List of regular expressions and associated actions */
%%
"if"                action("if", yytext);
{NOMBRE}            action("nombre", yytext);
{MOT}               action("mot", yytext);
{BLANC}
.                   action("inconnu", yytext);
%%
/* (4) Call to yylex */
int main( void ) {
        yylex() ;
        return 0;
}
```

Pour compiler la spécification, on exécute flex tp.lex. Le résultat est un fichier lex.yy.c qu'on va compiler avec cc lex.yy.c -o tp. Si on prépare un fichier input et ensuite on exécute ./tp < input on voit à l'écran la suite des unités lexicales trouvées par la spécification dont on va bientôt expliquer la structure. Par exemple, si le fichier input est le suivant :

```
   if  then
while  (  < do  ;

  uNmOt3232
   IF 9090907
    @
```

on obtient :

```
if if
mot then
mot while
inconnu (
inconnu <
mot do
inconnu ;
mot uNmOt
nombre 3232
mot IF
nombre 9090907
inconnu @
```

La spécification dans le fichier tp.lex est divisée en 4 parties dont on suggère la fonction en commentaire (entre /* */).

1. Dans la partie entre %{ et %}, on peut inclure et déclarer des fonctions C qui seront utilisées dans la suite.
2. La commande %noyywrap est là juste pour dire qu'on veut lire un seul fichier. Ensuite on a une liste de noms (LETTRE, CHIFFRE,...) qu'on associe à des expressions rationnelles. La notation [a-zA-Z] permet de traiter d'un coup toutes les lettres (ASCII) et de même la notation [0-9] traite tous les chiffres. On utilise $*$ pour l'itération, $+$ pour l'itération positive et $|$ pour l'union.
3. Dans la partie entre %% et %% on a une liste d'expressions rationnelles suivies par une action. Notez que dans l'analyse du texte, on considère les expressions rationnelles et les actions associées dans l'ordre. Par exemple, ceci permet de reconnaître if en tant qu'un mot particulier (réservé) plutôt que comme un mot quelconque. On notera aussi que la variable yytext est une variable prédéfinie de flex qui pointe à la chaîne de caractères qu'on vient de reconnaître.
4. Dans la dernière partie, on a une fonction main qui appelle une fonction prédéfinie de flex qui s'appelle yylex et qui va démarrer l'analyse lexicale du fichier passé en entrée.

Exercice 1 *Votre première tâche est d'écrire un analyseur lexicale pour un (très) petit langage de programmation qu'on va appeler* Imp*. Dans ce langage :*

— *un identificateur* Imp *est une suite alpha-numérique qui commence par une lettre et*
— *un nombre entier est une suite non-vide de chiffres décimales.*

Par ailleurs on peut utiliser les symboles $+$, $-$, $($ *et* $)$ *pour écrire des expressions arithmétiques, le symbole* $<$ *pour les comparer et les symboles* $=, \{, \}$ *et les mots* skip, if, then, else, while *et* do *pour écrire des commandes. Par exemple, avec le fichier* input *suivant :*

```
n = 1
x = 1
while ( x <5)
    n=x*n
  x=x+1
```

on pourrait avoir la sortie suivante :

```
identificateur n
assign =
nombre 1
identificateur x
assign =
nombre 1
while while
lpar (
identificateur x
less <
nombre 5
rpar )
identificateur n
assign =
identificateur x
inconnu *
identificateur n
identificateur x
assign =
identificateur x
plus +
nombre 1
```

Exercice 2 *Certains langages comme* python, *considèrent que les seuls espacements admissibles en début de ligne sont des tabulations ou* 4 *espaces consécutifs et que leur nombre est significatif. Modifiez votre analyseur lexicale pour qu'il traite cette situation. En reprenant l'exemple précédent, voici une sortie attendue.*

```
identificateur n
assign =
nombre 1
identificateur x
assign =
nombre 1
while while
lpar (
identificateur x
less <
nombre 5
rpar )
tabulation
identificateur n
assign =
identificateur x
```

```
inconnu *
identificateur n
espace illegal
identificateur x
assign =
identificateur x
plus +
nombre 1
```

Remarque Les logiciels d'*analyse lexicale* comme flex sont couplés avec des logiciels d'*analyse syntaxique* comme bison (un succésseur de yacc) pour construire une représentation *arborescente* d'un programme (la *syntaxe abstraite* mentionnée dans la section 3.2). A partir de cette représentation arborescente, il est possible d'effectuer un certain nombre d'*analyses* et ensuite soit d'*interpréter* le code source soit de le *compiler* vers un code objet directement exécutable par la machine. Comme évoqué dans la préface, des contraintes de temps nous imposent de faire l'impasse sur les *langages algébriques* et sur leur application à l'analyse syntaxique. Dans ce travail il est question uniquement d'*analyse lexicale*.

A.2 Diagrammes de décision binaire

Soit $\mathbf{2} = \{0, 1\}$ l'ensemble des valeurs binaires. Un *diagramme de décision binaire* (qu'on abrège en *BDD*) est une structure de données arborescente (plus précisement un graphe acyclique) pour réprésenter les *fonctions booléennes* de la forme $f : \mathbf{2}^m \to \mathbf{2}$, pour $m \geq 0$. On peut aussi voir un BDD comme une sorte d'automate fini déterministe spécialisé pour reconnaître les mots $b_1 \cdots b_n$ tels que $f(b_1, \ldots, b_n) = 1$, où $b_i \in \mathbf{2}$, pour $i = 1, \ldots, n$. En on peut aussi voir un BDD comme un circuit combinatoire à base de multiplexeurs (la version matériel de l'*if-then-else*). Les BDD sont largement utilisés dans le domaine de la vérification de circuits [Bry86]. Dans cet exercice on se concentre sur le premier point de vue avec l'objectif de concevoir un algorithme qui permet de minimiser la taille du BDD.

Soit $V = \{x_m, \ldots, x_1\}$, $m \geq 0$, un ensemble fini de variables avec un ordre total $x_m < x_{m-1} < \ldots < x_1$. Par convention, on suppose aussi que $x_1 < 0$ et $x_1 < 1$. Un BDD par rapport à cet ordre est un *graphe dirigé* avec un ensemble fini de noeuds N, un ensemble d'arêtes $A \subseteq N \times N$ et qui satisfait les propriétés suivantes :

- Un noeud $n \in N$ est désigné comme noeud *racine* et tout noeud est accessible depuis la racine.
- Chaque noeud n a une *étiquette* $v(n) \in V \cup \{0, 1\}$.
- Si $v(n) \in V$ alors le noeud n a deux arêtes sortantes vers les noeuds qu'on désigne par $b(n)$ et $h(n)$.
- Si $v(n) \in \{0, 1\}$ alors le noeud n n'a pas d'arête sortante.
- Il y a au plus un noeud étiqueté par 0 et au plus un noeud étiqueté par 1.
- Pour tout noeud n, si $v(n) \in V$ alors $v(n) < v(b(n))$ et $v(n) < v(h(n))$ (on traverse les noeuds par ordre croissant des étiquettes). Notez que cette condition force l'*acyclicité* du graphe.

On associe à un BDD β une fonction unique $f_\beta : \mathbf{2}^m \to \mathbf{2}$ qui prend en argument un vecteur de m valeurs binaires et retourne une valeur binaire. Pour calculer la sortie de $f_\beta(c_m, \ldots, c_1)$, $c_i \in \mathbf{2}$ pour $i = 1, \ldots, m$, on se place au noeud racine de β et on progresse dans le BDD jusqu'à arriver à un noeud étiqueté par 0 ou 1. La valeur de l'étiquette est alors la sortie de la fonction. La règle de progression est que si on se trouve dans le noeud n et $v(n) = x_i$ alors on se déplace vers $b(n)$ si $c_i = 0$ (b pour bas) et vers $h(n)$ (h pour haut) si $c_i = 1$. Par construction, le chemin existe et est unique.

Exercice 3 *Dessinez un BDD avec 9 noeuds qui définit la fonction $f : \mathbf{2}^3 \to \mathbf{2}$ suivante :*

$c_3c_2c_1$	000	001	010	011	100	101	110	111
$f(c_3, c_2, c_1)$	0	0	0	1	0	1	0	1

Vous allez suivre les conventions suivantes : la racine est en haut et les arêtes d'un noeud n à un noeud $b(n)$ ($h(n)$) sont des lignes pointillées (continues). Notez qu'en allant de la racine vers les feuilles on rencontre des étiquettes croissantes d'après l'ordre défini ci dessus.

On va représenter un noeud d'un BDD par des valeurs de type :

```
struct node {int label; struct node * b; struct node * h;};
```

une variable x_i, $i = m, \ldots, 1$ est représentée par l'entier négatif $-i$ et une valeur binaire 0 ou 1 par les entiers 0 et 1 respectivement (avec ces conventions, l'ordre sur les entiers coïncide avec l'ordre défini sur les étiquettes). Un BDD sera représenté par un pointeur à la racine du graphe.

Exercice 4 *Programmez une fonction* alloc_node *d'en tête* struct node ∗ alloc_node(int x) *qui alloue un* struct node *avec* malloc*, initialise le champ* label *à* x *et les champs* b *et* h *à* NULL *et retourne un pointeur au noeud.*

A partir de maintenant, vous ferez l'hypothèse que l'expression rand()%n donne un entier dans $\{0, \ldots, n-1\}$ avec probabilité uniforme et dans un temps constant $O(1)$. Comment peut-on choisir une fonction $f : \mathbf{2}^m \to \mathbf{2}$, $m \geq 0$ avec probabilité uniforme ?

Exercice 5 *Programmez une fonction d'en tête* struct node ∗ gen_bdd(int m) *qui prend en argument un entier non-négatif* m *et retourne un pointeur à un BDD qui représente une fonction* $f : \mathbf{2}^m \to \mathbf{2}$ *choisie avec probabilité uniforme.*

Une fonction $f : \mathbf{2}^m \to \mathbf{2}$, $m \geq 0$ est *symétrique* si elle est invariante par permutation de ses arguments, c'est à dire pour toute permutation $\sigma : \{1, \ldots, m\} \to \{1, \ldots, m\}$ et pour tout $c_m, \ldots, c_1 \in \mathbf{2}$ on a $f(c_m, \ldots, c_1) = f(c_{\sigma(m)}, \ldots, c_{\sigma(1)})$.

Exercice 6 *Montrez qu'une fonction* $f : \mathbf{2}^m \to \mathbf{2}$ *est symétrique si et seulement si il y a une fonction* $g : \{0, \ldots, m\} \to \mathbf{2}$ *telle que* $f(c_m, \ldots, c_1) = g(\Sigma_{i=m,\ldots,1} c_i)$.

Exercice 7 *Programmez une fonction d'en tête* struct node ∗ gen_sbdd(int m) *qui prend en argument un entier non-négatif* $m \geq 0$ *et retourne un pointeur à un BDD qui représente une fonction* symétrique $f : \mathbf{2}^m \to \mathbf{2}$ *choisie avec probabilité uniforme.*

Soit β un BDD. La *simplification (S1)* consiste à trouver un noeud n dans β tel que $v(n) \in V$ et $b(n) = h(n) = n'$ et à : (i) rédiriger vers n' toutes les arêtes vers n, et (ii) éliminer le noeud n. Le nouveau BDD obtenu définit toujours la même fonction.

Exercice 8 *Programmez une fonction d'en tête* struct node ∗ simplify1(struct node ∗ bdd) *qui prend en argument le pointeur vers un BDD* β *et retourne un pointeur vers un BDD qui définit la même fonction et dans lequel la simplification (S1) ne s'applique pas. Avant de programmer la fonction, vous expliquerez son fonctionnement sur l'exemple de la question 1 et vous analyserez sa complexité asymptotique en temps.*

Soit β un BDD. La *simplification (S2)* consiste à trouver deux noeuds différents n, n' dans β tels que $v(n) = v(n') \in V$, $b(n) = b(n')$ et $h(n) = h(n')$ et à : (i) rédiriger vers n toutes les arêtes vers n' et (ii) éliminer le noeud n'. Un *BDD réduit* est un BDD où les simplifications (S1) et (S2) sont impossibles.

Exercice 9 *Donnez une borne supérieure au nombre de noeuds d'un BDD réduit qui représente une fonction symétrique* $f : \mathbf{2}^m \to \mathbf{2}$.

Exercice 10 *Programmez une fonction d'en tête* struct node ∗ simplify(struct node ∗ bdd) *qui prend en argument un pointeur vers un BDD* β *et retourne un pointeur vers un BDD* réduit *qui définit la même fonction. Avant de programmer la fonction, vous expliquerez son fonctionnement sur l'exemple de la question 1 et vous analyserez sa complexité asymptotique en temps. Pour répondre à cette question il peut être utile de disposer d'une table de hachage et sachiez qu'il est possible de résoudre le problème en traversant le BDD une seule fois.*

A.3 Spécification et vérification de programmes

Dans ce TP, on considère le problème de spécifier et vérifier une fonction du langage C. Pour ce faire, on va choisir une terrain de jeu particulièrement favorable où on peut juste voir une formule de l'arithmétique étudiée dans le chapitre 4 comme une description de l'état de la fonction. De plus, on fera les hypothèses suivantes :

— la fonction reçoit des arguments de type int et retourne une valeur de type int,
— le corps de la fonction est constitué de déclarations de variables entières, d'affectations, branchements et boucles while,
— on suppose qu'on dispose d'un espace illimité pour mémoriser les entiers (on ignore les débordements).

Exercice 11 *Installez le logiciel* frama-c *en suivant les instructions disponibles à l'adresse* `https://frama-c.com/install-aluminium-20160501.html`. *Sur une distribution Debian/Ubuntu,* sudo apt-get install frama-c *fait l'affaire !*

Sur le site `https://frama-c.com` on trouvera une documentation assez complète et dans `https://stackoverflow.com/tags/frama-c/`, des discussions plus ou moins intéressantes. En principe, on peut faire ce TP sans lire la documentation ou les discussions.

Spécification et Test

Considérons la fonction suivante qui calcule un polynôme sur les entiers avec une assertion qui affirme que le polynôme est toujours positif.

```
int poly(int x){
  int b=4431;
  int r=b*b;
  int s=x-2*b;
  r=r+x*s;
  assert(r>0);
  return r;
}
```

Exercice 12 *Écrire un programme qui va tester la fonction* poly *sur* 100 *entiers tirés de façon aléatoire avec probabilité uniforme dans l'intervalle* $[-10000, 10000]$. *Votre programme trouve-t-il (souvent) une erreur ?*

Spécification et Vérification

On va maintenant remplacer la ligne assert(r>0) par la ligne //@ assert(r>0). En frama-c, les commentaires qui commencent par //@ sont interprétés comme des *spécifications* sur une ligne. Si on veut écrire une spécification sur plusieurs lignes on commence par /*@ et on termine comme d'habitude avec */.

Le logiciel frama-c prend très au sérieux ces spécifications et cherche à prouver qu'elle sont satisfaites dans *toute exécution* possible. Si on lance la commande frama-c -wp poly.c où poly.c contient la fonction avec la spécification, on reçoit en retour un rapport d'échec. On peut avoir la même chose, en plus joli, avec une interface graphique en lançant la commande frama-c-gui -wp poly.c. [1]

1. Le logiciel frama-c peut effectuer plusieurs types d'analyses. Dans ce TP, on est concerné juste avec celle qu'on appelle *weakest precondition*, d'où l'utilisation de l'option -wp en ligne de commande.

Exercice 13 *Bien sûr il est faux que le polynôme est toujours positif mais* frama-c *ne le sait pas ! Ce qu'il sait est qu'il n'arrive pas à prouver l'assertion. Remplacez dans la fonction la ligne* r=r+x*s ; *par la ligne* r=r+x*s+1 ; *(ceci rend l'assertion vraie). Si vous demandez à* frama-c *de prouver la nouvelle assertion vous devriez avoir à nouveau un échec. Que faire ? Donnez un tuyau à* frama-c *en ajoutant l'assertion* //@ assert r==(x-b)*(x-b)+1 ; *juste avant l'assertion* //@ assert(r>0) ; *. Maintenant,* frama-c *devrait être capable de vérifier que le tuyau est valide et en plus que le polynôme est toujours positif.*

Spécifier des fonctions (sans boucles)

Pour toute fonction, on peut introduire des annotations pour décrire les valeurs attendues en entrée et la relation entre les valeurs en entrée et la valeur en sortie. Ces spécifications s'écrivent dans le langage de l'arithmétique en logique du premier ordre, ce qui veut dire qu'on dispose des opérations arithmétiques $(+, -, *, /, \%, \ldots)$, des prédicats de comparaison $(<, <=, \ldots)$, des opérateurs logiques propositionnels $(!, \&\&, ||, \ldots)$ et des quantificateurs universel (\forall) et existentiel (\exists). Par exemple, voici une spécification qui dit que si on multiplie deux nombres entiers positifs a et b alors le résultat, dénoté par \result, est positif et a est un de ses diviseurs.

```
/*@ requires a>0 && b>0;
    ensures (\result > 0) && (\exists integer c; a*c==\result); */
int mul(int a, int b){
  return a*b;
}
```

Exercice 14 *Considérez la fonction suivante :*

```
int troisieme(int a, int b){
  return 6-a-b;
}
```

Spécifiez et vérifiez la propriété suivante : si $a, b \in \{1, 2, 3\}$ *et* $a \neq b$ *alors la fonction rend la valeur dans* $\{1, 2, 3\} \backslash \{a, b\}$.

Un *multiplexeur* est un circuit combinatoire avec 3 entrées et une sortie qui réalise la fonction conditionnelle $a \rightsquigarrow b, c$ sur les booléens (définition 3.2.4). On rappelle qu'avec la fonction conditionnelle et les constantes 0 et 1, on peut représenter toutes les fonctions booléennes (exercice 34). A tout circuit combinatoire construit avec des multiplexeurs, on peut associer une fonction C qui simule sont comportement :

- les arguments de la fonction correspondent aux entrées du circuit,
- on associe une variable différente à la sortie de chaque multiplexeur,
- on simule chaque multiplexeur avec entrées a, b, c et sortie d avec une affectation de la forme :
  `d=a?b:c;`
- comme le graphe associé à un circuit combinatoire est acyclique il est toujours possible de faire un tri topologique des multiplexeurs et d'exécuter les branchements associés dans un ordre qui respecte la propagation du signal dans le circuit.

Par exemple, la fonction suivante correspond à un circuit combinatoire à base de multiplexeurs qui calcule la fonction booléenne *AND* :

```
/*@ requires (0<=a && a<=1 && 0<=b && b<=1);
    ensures (\result == (a && b)); */

int and(int a, int b){
    int c,d;
    c=b?1:0;
    d=a?c:0;
    return d;
}
```

Exercice 15 *1. Construisez un circuit combinatoire à base de multiplexeurs qui calcule la fonction majorité à* 3 *arguments. La fonction booléenne majorité retourne* 1 *ssi au moins* 2 *arguments sont* 1*.*

2. Traduisez votre circuit combinatoire en une fonction C *et spécifiez son comportement avec* requires *et* ensures*.*

3. Vérifiez votre spécification.

Fonctions avec boucles : assigns et variant

On va maintenant s'intéresser à des fonctions qui contiennent des boucles while. En pratique, la preuve de ces programmes devient vite délicate et il faut aider frama-c autant que possible. Une façon de l'aider est de lui dire quelles variables peuvent être modifiée par la boucle. Par exemple, on ajoute une spécification loop assigns x pour dire que la boucle modifie seulement la variable x.

Un deuxième problème concerne la terminaison des boucles. Il suit des résultats développés dans le chapitre 2, que ce problème est indécidable et que donc une automation complète est impossible. Le logiciel frama-c connaît une stratégie de preuve élémentaire qui consiste à définir pour chaque boucle while, une expression numérique qui dépend des variables utilisées dans la boucle. Si l'expression numérique s'évalue en un nombre naturel et si le système peut montrer qu'à chaque itération la valeur diminue alors on a une preuve que la boucle termine. En effet, si la boucle ne terminait pas alors on aurait une suite infinie strictement descendante dans l'ensemble des nombres naturels ce qui est impossible ! Pour spécifier l'expression numérique dont la valeur est censée diminuer, on utilise le mot réservé loop variant comme dans l'exemple suivant dans lequel on calcule la somme des premiers n nombres. Le système vérifie que la quantité $n - i$ diminue à chaque itération.

```
int somme(int n){
   int s=0;
   int i=0;
   /*@ loop assigns s,i;
       loop variant  (n-i); */
   while (i<=n){
     s=s+i;
     i=i+1;
   }
   return s;
}
```

Exercice 16 *Prouvez la terminaison de la fonction suivante (donc de la boucle qu'elle contient) en faisant les hypothèses les plus faibles possibles sur les entrées.*

```
int q(int a,int b){
  int r=a;
  int q=0;
  while (r>=b){
      r=r-b;
      q=q+1;
  }
  return q;
}
```

Fonctions avec boucles : invariant

L'objectif est de caractériser l'état dans lequel le programme peut se trouver si jamais il sort de la boucle. La notion clef qui permet de traiter ce problème est celle d'*invariant*. Considérons une boucle de la forme :

$$\mathsf{while}\ (b)\{S\} \tag{A.1}$$

où b est une condition logique et S le corps de la boucle. Un invariant est une prédicat I qui dépend des variables de la boucle tel que si avant l'exécution de S les variables satisfont I et la condition logique b alors après l'exécution de S, les variables satisfont toujours la propriété I. En frama-C, on utilise l'annotation loop invariant I ; pour décrire les invariants d'une boucle. Voici une fonction qui calcule le nombre d'itérations nécessaires pour que la suite de Syracuse avec valeur initiale n arrive à 1. Avec frama-c, on peut vérifier l'invariant $(\mathsf{n} >= 1)\&\&(\mathsf{t} >= 0)$. Notez que l'invariant ne dit rien sur la terminaison de la boucle en question (et pour cause, il s'agit d'un problème ouvert !).

```
/*@ requires (n>=1);
    ensures (\result >=0); */
int syracuse(int n){
  int t=0;
  /*@ loop assigns t, n;
      loop invariant (n>=1)&&(t>=0); */
  while (n>1){
    t++;
    if (n%2==0){n=n/2;}
    else {n=3*n+1;}
  }
return t;
}
```

Exercice 17 *Considérez le programme suivant qui cherche à calculer l'approximation par défaut de la racine carrée d'un nombre positif. Proposez une spécification formelle et vérifiez-la.*

```
int racine(int n){
   int r=1;
   while (r*r<=n){
     r=r+1;
   }
   return (r-1);
}
```

A.4 Réduction à *SAT*

L'objectif de ce travail est l'utilisation pratique de la notion de *réduction polynomiale* (définition 5.2.1). On s'intéresse au problème du *calcul du nombre chromatique d'un graphe* et plutôt que développer un algorithme *ad hoc*, on va utiliser un programme appelé MiniSat qui prend en entrée une formule CNF du calcul propositionnel (représentée au format DIMACS) et calcule, si elle existe, une affectation qui la satisfait. Le programme MiniSat englobe un certain nombre d'heuristiques et d'optimisations. Le pari est qu'en ce qui concerne (i) le temps de développement et (ii) l'efficacité du programme obtenu il est avantageux de passer par MiniSat plutôt que de programmer directement une solution du problème. Comme étapes préliminaires il faut :

- Télé-charger et installer MiniSat `http://minisat.se/` sur votre machine. Sur la mienne, `sudo apt-get minisat` fait l'affaire.
- Comprendre le format DIMACS. Voici un exemple :
  ```
  c Ceci est un commentaire
  p cnf 5 3
  1 -5 4 0
  -1 5 3 4 0
  -3 -4 0
  ```
  Les lignes qui commencent par `c` sont des commentaires et elles sont facultatives. La ligne `p cnf 5 3` déclare une `cnf` avec `5` variables et `3` clauses. Chaque clause est décrite par une ligne d'entiers séparés par des espaces qui termine par un `0`. Par exemple, la ligne `1 -5 4 0` correspond à la clause $x_1 \vee \neg x_5 \vee x_4$.
- Lire la documentation de MiniSat. La première chose à comprendre est qu'on peut appeler MiniSat en lui passant en argument le nom d'un fichier qui contient une formule *CNF* au format DIMACS et le nom d'un fichier dans lequel MiniSat doit écrire le résultat. Le résultat peut être de deux types :
  - formule satisfaisable et une affectation qui le prouve (un certificat),
  - formule non-satisfaisable.

  Par exemple, si la formule au format DIMACS est celle décrite ci-dessus, le résultat pourrait être le suivant :
  ```
  SAT
  -1 -2 -3 -4 -5 0
  ```
  et si on ajoute des clauses de façon à rendre la formule non-satisfaisable le résultat sera le suivant :
  ```
  UNSAT
  ```

Exercice 18 *Pour développer votre solution, il est fortement conseillé d'utiliser un* langage de script*. Le choix est libre, pour ma part dans mes tests j'ai utilisé le langage* python*. Dans un premier temps, il s'agit d'écrire un* script *qui automatise les tâches suivantes :*

- *A partir d'un graphe non-dirigé $G = (N, A)$ et d'un nombre k générer un fichier qui contient la description d'une formule du calcul propositionnel au format* DIMACS *qui est satisfaisable ssi le graphe G admet une k-coloration.*
- *Appeler* MiniSat *en lui passant le fichier au format* DIMACS*.*
- *Récupérer la réponse de* MiniSat *et en cas de résultat positif, vérifier que l'affectation fournie correspond bien à une k-coloration du graphe et l'afficher.*

Exercice 19 *Dans un deuxième temps, on ajoutera :*

— *Une fonction qui génère des graphes de test. Par exemple, on peut générer les graphes qui correspondent au jeu du Sudoku (exercice 57) ou à un problème d'emploi du temps (exercice 58).*
— *Un script qui calcule le nombre chromatique du graphe généré, par exemple, par une* recherche dichotomique *; il s'agira donc d'itérer un certain nombre de fois la génération d'une CNF au format* DIMACS*, l'appel de* MiniSat *et l'analyse de la réponse de* MiniSat *jusqu'à déterminer le nombre minimum de couleurs qui permet de colorier le graphe.*